QuantLib Notes

Patrick Hénaff
Euria
Université de Bretagne Occidentale

January 1, 2011

Contents

The purpose of this note is to present some aspects of the QuantLib library, and in particular to explore how the entities specific to quantitative finance are implemented in QuantLib.

We focus on the representation of market data in QuantLib and on the maintenance of consistency among various calculated results.

1 Introduction

At a high level, the pricing of a financial claim can be expressed as:

$$P = E^Q(f(X)) \tag{1}$$

where:

- X represents a vector of random variables, observed at future dates determined by the nature of the claim.
- $f()$ represents the cash flow function, discounted to the pricing date
- Q represents the (risk-neutral) probability measure

In short, the price P is the discounted expected value of future cash flows.
This equation leads us to identify three types of entities:

- The financial contract, which defines the cash flow schedule represented by the function $f()$
- The stochastic processes followed by the risk factors X
- The calculation method used to estimate the expected value $E^Q()$.

Each one of these entities is represented by a hierarchy of classes in QuantLib. In the following sections, we focus on the representation of market data (the vector X) and one aspect of the calculation process, which minimizes the amount of calculation while enforcing consistency between the market data and the calculated values.

The text is illustrated by code segments from QuantLib, executed from `python/sage`. The QuantLib classes are wrapped one-to-one into python classes. Using the `python` syntax for illustration enables to abstract from the C++ implementation details to focus on the underlying design.

This document is produced with the `sagetex` package, which allows one to combine code and text, and produce in the final document the result of on-the-fly execution of QuantLib code.

2 Modeling Market Data

Modeling market data and calculated values is a key element of a successful design, as this is where one finds the largest diversity of concrete cases.

We will use an European option for illustrative purpose. How should one represent S_0, the value of the underlying asset on pricing date? The naive solution:

```
double S0 = 100.0
```

is not a good idea for at least two reasons:

- market data items are not observed in isolation, but organized in sets that must be kept consistent.
- In general, there are many ways of representing the same information (e.g. a yield curve), and it is important to define a standard interface to the data, independently of the implementations details.

Each aspect is now described in more details.

2.1 Maintaining Consistency in Market Data

To introduce the topic, let's consider some examples of relationships between market data:

- The price of asset A is defined as the price of asset B plus a constant spread: :

$$P_A(t) = P_B(t) + .1$$

- A price is defined as a function of two other prices:

$$P_C(t) = .75P_A(t) + .25P_B(t)$$

This leads us to identify the abstract notion of `Quote`, and of various concrete sub-classes:

- SimpleQuote: a numerical value
- DerivedQuote: a "Quote" modified by a unary function
- CompositeQuote: a "Quote", function of two other quotes.

The Quote class and its derived classes provide a framework for defining the dependency relationships between quotes. This is best explained by the following example:

Let's first define two simple quotes:

```
sage: m1 = SimpleQuote(float(1.0))

sage: m2 = SimpleQuote(float(2.0))

sage: 'm1: %3.2f m2: %3.2f' % (m1.value(), m2.value())

                    m1:  1.00 m2:  2.00
```

then, define a derived Quote $m_3 = m_1 + 1$

```
sage: def plus1(x):
....:     return (x+1)
```

```
sage: m3 = DerivedQuote(QuoteHandle(m1), plus1)

sage: 'm3: %3.2f' % m3.value()

                              m3:  2.00
```

Quote m_3 is re-computed whenever `value()` is called. Thus, m_1 and m_3 always remain consistent.

The derived class `CompositeQuote` provide a way to create quotes that are defined by formula:

```
sage: def WeightedMean(x, y):
....:     return (3.0*x+y)/4.0


sage: m4 = CompositeQuote(QuoteHandle(m1), QuoteHandle(m2), WeightedMean)

sage: 'm1: %3.2f m2: %3.2f m4: %3.2f' % (m1.value(), m2.value(), m4.value())

                  m1:  1.00 m2:  2.00 m4:  1.25
```

Again, the CompositeQuote m_4 is re-computed at each call to `.value()`. If input data have not changed, this re-calculation is obviously superfluous. We will see later that QuantLib provides a sophisticated scheme to avoid such redundant calculations. That complex machinery involves a new entity (the financial instrument), which is discussed in section 3.

2.2 Data Encapsulation

There are many ways to define a yield curve. At one extreme, one can use a constant rate for all maturities, but in general a yield curve is constructed from a set of market quotes: over-night rate, deposit rates, Euro-currency prices, swap rates, etc. Regardless of the construction method and the input data, one would like to obtain, for example, the discount factor corresponding to a given future payment date. The notion of discount factor is well defined, regardless of representation of the curve and the market data used for constructing it. Of course the calculation details will be specific to each curve type. In QuantLib, you can generically query a curve for a discount factor, as illustrated by the following example:

We first construct a simple, constant yield curve:

```
sage: dtToday = Date(int(5), November, int(2010))

sage: Settings.instance().evaluationDate = dtToday

sage: settlementDays = int(2)

sage: dtSettlement = dtToday + settlementDays

sage: dtHorizon = dtToday + int(180)
```

```
sage: yc1 = FlatForward(dtSettlement, float(0.06), Actual365Fixed())

sage: '''Discount factor for %s: %f''' % (dtHorizon, yc1.discount(dtHorizon))

        Discount factor for May 4th, 2011:  0.971164
```

The discount factor is computed by:

$$e^{-.06\frac{178}{365}} = .9711$$

A more realistic example involves the construction of a piece-wise constant yield curve from deposit rates:

```
sage: deposits = { Period(int(1),Weeks): float(0.0382),
....:              Period(int(1),Months): float(0.0372),
....:              Period(int(3),Months): float(0.0363),
....:              Period(int(6),Months): float(0.0353),
....:              Period(int(9),Months): float(0.0348),
....:              Period(int(1),Years): float(0.0345) }


sage: calendar = TARGET()

sage: dayCounter = Actual360()

sage: depositHelpers = [ DepositRateHelper(QuoteHandle(SimpleQuote(deposits[p])),
....:                                      p, settlementDays,
....:                                      calendar, ModifiedFollowing,
....:                                      False, dayCounter)
....:                    for p in deposits.keys()]


sage: yc2 = PiecewiseFlatForward(dtSettlement, depositHelpers, Actual360())

sage: '''Discount factor for %s: %f''' % (dtHorizon, yc2.discount(dtHorizon))

        Discount factor for May 4th, 2011:  0.982814
```


To emphasize, the relevant aspect of this example is that in both cases, the discount factor is obtained with the same syntax. The objects `yc1` and `yc2` belong to different classes, all derived from the virtual class `YieldTermStructure`, where the interface `discount` is defined.

3 Pricing

In this section, we provide a step-by-step description of the calculation of of the net present value of a financial claim. The QuantLib architecture is complex, but achieves two major goals:

- It separates the three major entities involved in a financial calculation, as discussed earlier: the contract, the risk factors and their dynamics, and finally the calculation method for the expected discounted value. This clear separation provides flexibility, and allows for some level of composition between the various building blocks.
- It implements a scheme for maintaining consistency between calculated values while minimizing the amount of redundant computations.

3.1 Calculation Steps

Let's consider again the pricing of a European option. The corresponding code may be found in the "Example" folder of the QuantLib distribution.

An European option claim is constructing by assembling a payoff and exercise conditions:

```
sage: exercise = EuropeanExercise(Date(int(17),May,int(1999)))

sage: payoff = PlainVanillaPayoff(Option.Call, float(8.0))

sage: option = VanillaOption(payoff, exercise)
```

For illustrative purpose, market data is simplified to the extreme: in particular, all curves are flat:


```
sage: todaysDate = Date(int(15),May,int(1998))

sage: Settings.instance().evaluationDate = todaysDate

sage: settlementDate = Date(int(17),May,int(1998))

sage: riskFreeRate = FlatForward(settlementDate, float(0.05), Actual365Fixed())

sage: dividendYield = FlatForward(settlementDate, float(0.05), Actual365Fixed())

sage: underlying = SimpleQuote(float(7.0))

sage: volatility = BlackConstantVol(todaysDate, TARGET(), float(0.10), Actual365Fixed())
```

Next, we specify the dynamic of the underlying asset: a log-normal process. The calculation method for the discounted expected value will be the Black-Scholes formula.

```
u = QuoteHandle(underlying)
d = YieldTermStructureHandle(dividendYield)
r = YieldTermStructureHandle(riskFreeRate)
v = BlackVolTermStructureHandle(volatility)
process = BlackScholesMertonProcess(u,d,r,v)
priceur = AnalyticEuropeanEngine(process)
option.setPricingEngine(priceur)
```

The net present value is finally obtained by:

```
sage: prixAnalytique = option.NPV()

sage: '''prix: %5.3f''' % prixAnalytique

                          prix:  0.030
```

The flexibility provided by this architecture is illustrated next. In order to price the option with a binomial tree, one just need to replace the integration method associated with the claim:

```
sage: timeSteps = int(801)
sage: option.setPricingEngine(BinomialVanillaEngine(process,"crr",timeStep
sage: prixCRR = option.NPV()
```

Another call to `NPV()`, with no change to market data, will not trigger a recalculation. The method will directly return the computed value, which has been kept in cache. This is a fundamental aspect of the QuantLib architecture, which is now described.

3.2 Optimization of Calculations

This section describes the scheme which is implemented in QuantLib to avoid redundant calculations, while maintaining consistency among calculated data.

The calculation sequence can be summarized as follows:

- The method `NPV()` is implemented in the `Instrument` class, from which `VanillaOption` is derived:

Listing 1: Implementation de NPV()

```
inline Real Instrument::NPV() const {
    calculate();
    QL_REQUIRE(NPV_ != Null<Real>(), "NPV not provided");
    return NPV_;
}
```

- The method `calculate()` is also implemented in the `Instrument` class:

Listing 2: Instrument::calculate()

```
inline void Instrument::calculate() const {
    if (isExpired()) {
        setupExpired();
        calculated_ = true;
    } else {
        LazyObject::calculate();
    }
}
```

It is in `LazyObject` that we find the logic that prevents redundant calculations. The flag `calculated_` signals whether the data found in cache is up-to-date or not (next section describes the calculation of this flag).

Listing 3: LazyObject::calculate()

```
inline void LazyObject::calculate() const {
    if (!calculated_ && !frozen_) {
        calculated_ = true;   // prevent infinite recursion in
                              // case of bootstrapping
        try {
            performCalculations();
        } catch (...) {
            calculated_ = false;
            throw;
        }
    }
}
```

- If the data is not up-to-date, it is next recalculated by the method `performCalculations()`, implemented in the `Instrument` class:

Listing 4: Instrument::performCalculations()

```
inline void Instrument::performCalculations() const {
    QL_REQUIRE(engine_, "null pricing engine");
    engine_->reset();
    setupArguments(engine_->getArguments());
    engine_->getArguments()->validate();
    engine_->calculate();
    fetchResults(engine_->getResults());
}
```


The gathering of input arguments to the calculation algorithm is performed by `setupArguments()`), and the extraction of results is performed by `fetchResults()`. This careful design isolates the logic of the calculation algorithm from the representation of input data, and is described in details in chapter 2 of the document "Implementing QuantLib".

- The calculation per se is carried out in the pricing object "engine". Assume that the pricing object is of the kind `AnalyticEuropeanEngine`. The essential part of the implementation is:

Listing 5: AnalyticEuropeanEngine::calculate()

```
void AnalyticEuropeanEngine::calculate() const {

    Real variance =
        process_->blackVolatility()->blackVariance(
                    arguments_.exercise->lastDate(),
                    payoff->strike());
    DiscountFactor dividendDiscount =
```

```
            process_->dividendYield()->discount(
                                arguments_.exercise->lastDate());
        DiscountFactor riskFreeDiscount =
            process_->riskFreeRate()->discount(
                                arguments_.exercise->lastDate());
        Real spot = process_->stateVariable()->value();
        Real forwardPrice = spot * dividendDiscount / riskFreeDiscount;

        BlackCalculator black(payoff, forwardPrice, std::sqrt(variance),
                              riskFreeDiscount);

        results_.value = black.value();
    }
```

Note that this engine can be legitimately used with any process for which the notion of "black volatility" is defined, i.e. any process for which the price at expiry follows a log-normal process.

3.3 Ensuring Consistency among Calculated Data

In the previous section, we observed that the pricing task is only performed when the flag `calculated_= false`, which signals that the cached result is no longer up-to-date. This section describes the scheme that has been implemented in QuantLib to maintain the status of this flag, and thus to ensure consistency between the market data and the calculated results.

The option NPV is a function of:

- four types of market data, through the specification of the process for the underlying asset:
  1. volatility
  2. dividend yield
  3. risk-free rate
  4. spot value
- the calculation method for the discounted expected value

After a first call:

```
price = option.NPV()
```

a second call will return the cached value, as long as all the elements mentioned above remain unchanged. To assess that state, QuantLib uses the Observer design pattern.

An object of the class `BlackScholesMertonProcess` is both an Observer and an Observable.

- It is an Observer, since it must keep track of any changes occurring in the market data that define the process.

- It is Observable, because the characteristics of the process determine the value of the option.

When instantiated, the process object - by virtue of being an Observer - registers its dependency with respect to the 4 types of market data. This logic is found in `processes/blackscholesprocess.cpp`:

Listing 6: Fragment from BlackScholesProcess constructor

```
registerWith(x0_);
registerWith(riskFreeRate_);
registerWith(dividendYield_);
registerWith(blackVolatility_);
```

Note that the method `registerWith` establishes a two-way relationship between the Observer and the observed entity:

- The observer adds the observed entity to the list of items to be monitored
- The observed entity records the dependency between itself and the Observer.

Listing 7: Observer::registerWith(o)

```
inline void Observer::registerWith(const boost::shared_ptr<Observable>& h)
        if (h) {
            observables_.push_front(h);
            h->registerObserver(this);
        }
    }

inline void Observable::registerObserver(Observer* o) {
        observers_.push_front(o);
    }
```


Similarly, as part of the instantiation of a `AnalyticEuropeanEngine` pricing object, the dependency of this object with respect to the underlying process is recorded:

Listing 8: Constructor of AnalyticEuropeanEngine

```
AnalyticEuropeanEngine::AnalyticEuropeanEngine(
    const boost::shared_ptr<GeneralizedBlackScholesProcess>&process)
    : process_(process) {
        registerWith(process_);
    }
```

Finally, the option object records its dependency with respect to the pricing engine:

Listing 9: Instrument::setPricingEngine(p)

```
inline void Instrument::setPricingEngine(
        const boost::shared_ptr<PricingEngine>& e) {
        if (engine_)
```

```
            unregisterWith(engine_);
        engine_ = e;
        if (engine_)
            registerWith(engine_);
        // trigger (lazy) recalculation and notify observers
        update();
    }
```

Let's now consider the chain of events triggered by a change in market data and the subsequent recalculation of NPV.

```
sage: underlying.setValue(float(8.0))
sage: price_new = option.NPV()
```

The method `setValue` calls `notifyObservers` if the new value is not identical to the previous one:

Listing 10: SimpleQuote::setValue(x)

```
inline Real SimpleQuote::setValue(Real value) {
        Real diff = value-value_;
        if (diff != 0.0) {
            value_ = value;
            notifyObservers();
        }
        return diff;
    }
```


In our case, the `BlackScholesMertonProcess` process is an Observer of the market data, thus, `notifyObservers` calls the method `update()` of the `process` object. This has two effects:

- It labels this object as "not up to date" (`updated_ = false`),
- Since the process itself is an Observable, it calls in turn `notifyObservers`

Listing 11: GeneralizedBlackScholesProcess::update()

```
void GeneralizedBlackScholesProcess::update() {
        updated_ = false;
        StochasticProcess1D::update();
    }

void StochasticProcess::update() {
        notifyObservers();
    }
```

The pricing object is an Observer of the `process` object, and is thus also now marked as "not up to date". This chain reaction ultimately reaches the `option` object, and a call to `NPV()` triggers a recalculation.