Introduction to Selected Classes of the QuantLib Library II

Dimitri Reiswich

December 2010

In the whole tutorial I assume that you have included the QuantLib header via

```
#include <ql/quantlib.hpp>
```

in the code. I will not state this include command in the example code explicitly, so be sure that you have included the header everywhere. Also, I will use the QuantLib namespace to make the code more readable and compact. To reproduce the code, make sure to state

```
using namespace QuantLib;
```

in the place where you are using the code.

QuantLib provides several procedures to calculate the integ

$$\int_a^b f(x)\mathrm{d}x$$

of a scalar function $f : \mathbb{R} \to \mathbb{R}$. For the majority of the integration procedures we have to provide

- an absolute accuracy: if the increment of the current calculation and the last calculation is below the accuracy, stop the integration.
- the number of maximum evaluations: if this numbe reached, stop the integration.

For special numerical integrations, such as the Gaussian adrature procedure, we have to provide other parameters. The first group of integration procedures will be discussed first.

This group includes the

- `TrapezoidIntegral`
- `SimpsonIntegral`
- `GaussLobattoIntegral`
- `GaussKronrodAdaptive`
- `GaussKronrodNonAdaptive`

The mathemtical details of the procedures are discussed in the numerical standard literature. In `QuantLib`, the setup to construct a general numerical integrator is

```
Integrator myIntegrator(Real absoluteAccuracy, Size maxEvaluations)
```

The integral between a and b is retrieved through

```
Real operator()(const boost::function<Real (Real)>& f,Real a, Real b)
```

by passing a boost function object to the operator.

To test the integration procedures, we will look at the integral representation of a call with strike K given as

$$e^{-r\tau}\mathbb{E}(S-K)^{+} = e^{-r\tau}\int_{K}^{\infty}(x-K)f(x)dx$$

with $f(x)$ being the lognormal density with mean

$$\log(S_0) + (r - \frac{1}{2}\sigma^2)\tau$$

and standard deviation

$$s = \sigma\sqrt{\tau}$$

In the following example, the integral will be calculated numerically. The corresponding code follows.

```
#include <boost/math/distributions.hpp>

Real callFunc(Real spot, Real strike,
                          Rate r, Volatility vol, Time tau, Real x){

        Real mean=log(spot)+(r-0.5*vol*vol)*tau;
        Real stdDev=vol*sqrt(tau);

        boost::math::lognormal_distribution<> d(mean,stdDev);
        return (x-strike)*pdf(d,x)*exp(-r*tau);
}


void testIntegration4(){

        Real spot=100.0;
        Rate r=0.03;
        Time tau=0.5;
        Volatility vol=0.20;
        Real strike=110.0;

        Real a=strike, b=strike*10.0;

        boost::function<Real (Real)> ptrF;
        ptrF=boost::bind(&callFunc,spot,strike,r,vol,tau,_1);

        Real absAcc=0.00001;
        Size maxEval=1000;
        SimpsonIntegral numInt(absAcc,maxEval);

        std::cout << "Call Value: " << numInt(ptrF,a,b) << std::endl;
}
```

The output is the same as the one calculated by a standard Black-Scholes formula

```
Call Value:  2.6119
```

The code shows an elegant application of the `boost::bind` class. First, we define a function called `callFunc` which represents the integrand. This function takes all needed the parameters. Afterwards, we bind this function to a given set of market parameters leaving only the `x` variable as the free variable. The new function maps from $\mathbb{R}$ to $\mathbb{R}$ and can be used for the integration. Of course, any density can be used here, allowing for a general pricer.

An $n-$point Gaussian Quadrature rule is constructed such that it yields an exact integration for polynomials of degree $2n-1$ (or less) by a suitable choice of the points x_i and w_i for $i = 1, ..., n$ with

$$\int_{-1}^{1} f(x)\mathrm{d}x \approx \sum_{i=1}^{n} w_i f(x_i)$$

The integral $[-1, 1]$ can be transformed easily to a general integral $[a, b]$. There are different version of the weighting functions, and integration intervals. Quantlib offers the following (see the documentation)

- `GaussLaguerreIntegration`: Generalized Gauss-Laguerre integration for

$$\int_{0}^{\infty} f(x)\mathrm{d}x$$

  the weighting function here is

$$w(x, s) := x^s e^{-x} \text{ with } s > -1$$

- `GaussHermiteIntegration`: Generalized Gauss-Hermite integration

$$\int_{-\infty}^{\infty} f(x)\mathrm{d}x$$

with weighting function

$$w(x,\mu) = |x|^{2\mu}e^{-x^2} \text{ with } \mu > -0.5$$

- `GaussJacobiIntegration`: Gauss-Jacobi integration

$$\int_{-1}^{1} f(x)\mathrm{d}x$$

with weighting function

$$w(x,\alpha,\beta) = (1-x)^{\alpha}(1+x)^{\beta} \text{ with } \alpha,\beta > 1$$

- `GaussHyperbolicIntegration`:

$$\int_{-\infty}^{\infty} f(x)\mathrm{d}x$$

with weighting

$$w(x) = \frac{1}{\cosh(x)}$$

- `GaussLegendreIntegration`

$$\int_{-1}^{1} f(x)\mathrm{d}x$$

with weighting

$$w(x) = 1$$

- `GaussChebyshevIntegration`

$$\int_{-1}^{1} f(x)\mathrm{d}x$$

with weighting

$$w(x) = \sqrt{(1-x^2)}$$

Also, `GaussChebyshev2thIntegration` (the second kind) is available.

- `GaussGegenbauerIntegration`:

$$\int_{-1}^{1} f(x)\mathrm{d}x$$

with weighting

$$w(x,\lambda) = (1-x^2)^{\lambda-1/2}$$

Some of the Gaussian quadrature integrators will be tested on the next slide. The test will show the integration of the normal density with respect to the fixed limits. The integrators should reproduce the known analytical results (for example, integrating the density over $[0,\infty]$ should give 0.5).

```
#include <boost/function.hpp>
#include <boost/math/distributions.hpp>

void testIntegration2(){

                boost::function<Real (Real)> ptrNormalPdf(normalPdf);
                GaussLaguerreIntegration gLagInt(16); // [0,\infty]
                GaussHermiteIntegration  gHerInt(16); //(-\infty,\infty)
                GaussChebyshevIntegration gChebInt(64); //(-1,1)
                GaussChebyshev2thIntegration gChebInt2(64); //(-1,1)

                Real analytical=normalCdf(1)-normalCdf(-1);

                std::cout << "Laguerre:"        << gLagInt(ptrNormalPdf) << std::endl;
                std::cout << "Hermite:"         << gHerInt(ptrNormalPdf) << std::endl;
                std::cout << "Analytical:"              << analytical << std::endl;
                std::cout << "Cheb:"            << gChebInt(ptrNormalPdf) << std::endl;
                std::cout << "Cheb 2 kind:"     << gChebInt2(ptrNormalPdf) << std::endl;

}
```

The output of this function is

```
Laguerre:0.499992
Hermite:1
Analytical:0.682689
Cheb:0.682738
Cheb 2 kind:0.682595
```

As already mentioned, it is quite easy to transform the interval bounds from $[-1, 1]$ to a general interval $[a, b]$ via the following formula

$$\int_a^b f(x)\mathrm{d}x = \frac{b-a}{2}\int_{-1}^{1} f\Big(\frac{b-a}{2}x + \frac{b+a}{2}\Big)\mathrm{d}x \tag{1}$$

This shows that it is possible to use some Gaussian Quadrature integrators, even though they are specified for the interval $[-1, 1]$. The QuantLib integrators expect a function which takes one variable x. However, the integrated function in the right integral of equation (1) accepts 3 instead of 1 variables: x, a, b. This problem represents one case in a general class of problems, where a function with more than one input parameter is given, but only one parameter is a true variable. To integrate this function in QuantLib, we need to reduce the function to a mapping which accepts one parameter only. This can be achieved easily by using boost's bind and function libraries. We will examine a 3 variable example below. Assume that you need to integrate the function

```
Real myFunc(const Real& x, const Real& a, const Real& b)
```

but the variables a, b are constants. First, declare a function pointer via

```
boost::function <double (double)> myFuncReduced;
```

Given the parameters `a,b` the next step is to bind these variables to the function `func` and assign the result to the function pointer `myFuncReduced`. The corresponding code is

```
myFuncReduced=boost::bind(myFunc,_1,a,b);
```

The pointer `myFuncReduced` now represents a function $f : \mathbb{R} \to \mathbb{R}$ which can be handled over to the Gaussian Quadrature integrator. The code below shows an example, where we integrate the normal density over the interval $[-1.96, 1.96]$.

```
Real normalPdf(const Real& x, const Real& a, const Real& b){
                boost::math::normal_distribution<> d;
                Real t1=0.5*(b-a), t2=0.5*(b+a);
                return t1*pdf(d,t1*x+t2);
}

void testIntegration3(){
        Real a=-1.96, b=1.96;
        boost::function  <double (double)>      myPdf;

        myPdf=boost::bind(normalPdf,_1,a,b);
        GaussChebyshevIntegration gChebInt(64); //(-1,1)

        Real analytical=normalCdf(b)-normalCdf(a);

        std::cout << "Analytical:" << analytical<< std::endl;
        std::cout << "Chebyshev:" << gChebInt(myPdf)<< std::endl;
}
```

The output of this function is

```
Analytical:0.950004
Chebyshev:0.950027
```

`QuantLib` offers a variety of different one-dimensional solvers which search for an x such that

$$f(x) = 0$$

given a function $f : \mathbb{R} \to \mathbb{R}$. The following routines are avalable

- `Brent`
- `Bisection`
- `Secant`
- `Ridder`
- `Newton`: requires a derivative of the objective function
- `FalsePosition`

The constructor is a default constructor, taking no arguments. For example, Brent's solver can be initialized by `Brent mySolv;`

The solver has an overloaded solve function with the following 2 versions

```
Real solve(const F& f,
           Real accuracy,
           Real guess,
           Real step)

Real solve(const F& f,
           Real accuracy,
           Real guess,
           Real xMin,
           Real xMax)
```

The `solve` routine is a template function, which accepts any class `F` that has an operator of the form

```
Real operator()(const Real& x)
```

This can be either a class with such an operator or a function pointer. The `accuracy` parameter has different meanings, depending on the used solver. See the documentation for the definition in the solver you prefer. It enforces the solving routine to stop when

- either $|f(x)| < \epsilon$
- or $|x - x_i| < \epsilon$ with x_i being the true zero.

The other variable meanings are

- `guess`: your initial guess of the root
- `step`: in the first overloaded version, no bounds on the interval for the root are given. An algorithm is implemented to automatically search for the bounds in the neighborhood of your guess. The `step` variable indicates the size of the steps to proceed from the guess.
- `xMin, xMax`: are the left and right interval bounds.

The classical application of a root solver in Quantitative Finance is the implied volatility problem. Given a price p and the parameters S_0, K, r_d, r_f, τ we are seeking a volatility σ such thad

$$f(\sigma) = \text{blackScholesPrice}(S_0, K, r_d, r_f, \sigma, \tau, \phi) - p = 0$$

is fulfilled. The Black-Scholes function accepts either $\phi = 1$ for a call or $\phi = -1$ for a put. In the following example the Black Scholes function will be hard coded, although it is of course available in QuantLib and will be introduced later. Since the root solver accepts an object with an operator `double operator()(double)` we will need an implementation of $f(\sigma)$ given all the other parameters. We will use again boost's bind function for a convenient setup. The next slide shows the hard coded implementation of the Black-Scholes function and implied volatility problem.

```
#include <boost/math/distributions.hpp>

Real    blackScholesPrice(const Real& spot,
                                                const Real& strike,
                                                const Rate& rd,
                                                const Rate& rf,
                                                const Volatility& vol,
                                                const Time& tau,
                                                const Integer& phi){

         boost::math::normal_distribution<> d(0.0,1.0);
         Real dp,dm, fwd, stdDev, res, domDf, forDf;

         domDf=std::exp(-rd*tau); forDf=std::exp(-rf*tau);
         fwd=spot*forDf/domDf;
         stdDev=vol*std::sqrt(tau);

         dp=(std::log(fwd/strike)+0.5*stdDev*stdDev)/stdDev;
         dm=(std::log(fwd/strike)-0.5*stdDev*stdDev)/stdDev;

         res=phi*domDf*(fwd*cdf(d,phi*dp)-strike*cdf(d,phi*dm));
         return res;
}

Real    impliedVolProblem(const Real& spot,
                                                const Rate& strike,
                                                const Rate& rd,
                                                const Rate& rf,
                                                const Volatility& vol,
                                                const Time& tau,
                                                const Integer& phi,
                                                const Real& price){

                return blackScholesPrice(spot,strike, rd,rf,vol,tau, phi) - price;
}
```

The next step is to setup $f(\sigma)$ with

```
boost::function<Real (Volatility)> myVolFunc;
```

bind all the parameters, except the volatility, to this function and call the root solver. The code testing different solvers is given below

```
void testSolver1(){

        // setup of market parameters
        Real spot=100.0,strike=110.0;
        Rate rd=0.002, rf=0.01, tau=0.5;
        Integer phi=1;
        Real vol=0.1423;

        // calculate corresponding Black Scholes price
        Real price=blackScholesPrice(spot,strike,rd,rf,vol,tau,phi);
        // setup a solver
        Bisection mySolv1; Brent mySolv2; Ridder mySolv3;
        Real accuracy=0.00001, guess=0.25;
        Real min=0.0, max=1.0;
        // setup a boost function
        boost::function<Real (Volatility)> myVolFunc;
        // bind the boost function to all market parameters, keep vol as variant
        myVolFunc=boost::bind(&impliedVolProblem,spot,strike,rd,rf,_1,tau,phi,price);
        // solve the problem
        Real res1=mySolv1.solve(myVolFunc,accuracy,guess,min,max);
        Real res2=mySolv2.solve(myVolFunc,accuracy,guess,min,max);
        Real res3=mySolv3.solve(myVolFunc,accuracy,guess,min,max);

        std::cout << "Input Volatility:" << vol << std::endl;
        std::cout << "Implied Volatility Bisection:" << res1 << std::endl;
        std::cout << "Implied Volatility Brent:" << res2 << std::endl;
        std::cout << "Implied Volatility Ridder:" << res3 << std::endl;

}
```

The output of the program is

```
Input Volatility:0.1423
Implied Volatility Bisection:0.142296
Implied Volatility Brent:0.1423
Implied Volatility Ridder:0.1423
```

The Newton algorithm requires the derivative $\frac{\partial f}{\partial \sigma}$ (the vega) of the function $f(\sigma)$ for the root searcher. We will show again for the implied volatility example how the derivative can be incorporated. However, we need a `class` for this case which implements a function called `derivative`. Example code for the class which will be used in the solver is shown next

```
#include <boost/math/distributions.hpp>

class BlackScholesClass{
        private:
        Real spot_,strike_,price_,logFwd_;
        Real dp_,domDf_,forDf_,fwd_,sqrtTau_;
        Rate rd_,rf_;
        Integer phi_;
        Time tau_;
        boost::math::normal_distribution<> d_;
        public:
                BlackScholesClass(const Real& spot,
                                          const Real& strike,
                                          const Rate& rd,
                                          const Rate& rf,
                                          const Time& tau,
                                          const Integer& phi,
                                          const Real& price)
                                          :spot_(spot), strike_(strike), rd_(rd),rf_(rf),phi_(phi),
                                          tau_(tau),price_(price), sqrtTau_(std::sqrt(tau)),
                                          d_(boost::math::normal_distribution<>(0.0,1.0)){

                                                  domDf_=std::exp(-rd_*tau_);
                                                  forDf_=std::exp(-rf_*tau_);
                                                  fwd_=spot_*forDf_/domDf_;
                                                  logFwd_=std::log(fwd_/strike_);

                }

                Real operator()(const Volatility& x) const{
                                return impliedVolProblem(spot_, strike_,rd_,rf_,x,tau_,phi_,price_);
                }
                Real derivative(const Volatility& x)const{
                        // vega
                        Real stdDev=x*sqrtTau_;
                        Real dp=(logFwd_+0.5*stdDev*stdDev)/stdDev;
                        return spot_*forDf_*pdf(d_,dp)*sqrtTau_;
                }
};
```

The class has an operator which returns the output of the function `impliedVolProblem`, as before. Furthermore, a function `derivative` is defined which returns the vega for a given volatility. The code below shows how to setup the corresponding root search problem. In addition, the setup without given interval bounds is shown.

```
void testSolver2(){

// setup of market parameters
        Real spot=100.0,strike=110.0;
        Rate rd=0.002, rf=0.01, tau=0.5;
        Integer phi=1;
        Real vol=0.1423;
        // calculate corresponding Black Scholes price
        Real price=blackScholesPrice(spot,strike,rd,rf,vol,tau,phi);

        BlackScholesClass solvProblem(spot,strike,rd,rf,tau,phi,price);

        Newton mySolv;
        Real accuracy=0.00001, guess=0.10;
        Real step=0.001;
        // solve the problem
        Real res=mySolv.solve(solvProblem,accuracy,guess,step);

        std::cout << "Input Volatility:" << vol << std::endl;
        std::cout << "Implied Volatility:" << res << std::endl;
}
```

The output is:

```
Input Volatility:0.1423
Implied Volatility:0.1423
```

- Calculate the square root of 2 numerically by solving $x^2 - 2 = 0$. Choose any root searcher you prefer.
- Integrate the cosine function from $[0, \frac{\pi}{2}]$ numerically. Choose any integration method you like. Compare this to the analytical result.
- Calculate the number π numerically by solving $\sin(x) = 0$

On of the most frequently used tools in Quantitative Finance is interpolation. The basic idea is that you are given a discrete set of $(x_i, f(x_i))$ $i \in \{0, ..., n\}$ values of an unknown function f and you are interested in a function value at any point $x \in [x_0, x_n]$. The standard application is the interpolation of yield curves or volatility smiles. QuantLib offers the following 1-dimensional and 2− dimensional interpolations

- `LinearInterpolation` (1-D)
- `LogLinearInterpolation` and `LogCubicInterpolation` (1-D)
- `BackwardFlatInterpolation` (1-D)
- `ConvexMonotone` (1-D)
- `CubicInterpolation` (1-D)
- `ForwardFlatInterpolation` (1-D)
- `SABRInterpolation` (1-D)
- `BilinearInterpolation` (2-D)
- `BicubicSpline` (2-D)

We will assume that the x and y values are saved in `std::vector<Real>` vectors called `xVec,yVec`. The basic structure of the constructors is

```
Interpolation myInt(xVec.begin(),xVec.end(),yVec.begin(),optional parameters)
```

As usual, `xVec.begin()` returns a pointer showing to the first element of `xVec`, while `xVec.end()` points to the element succeeding the last element. The interpolated value at a given point `x` can then be obtained via the operator

```
Real operator()(Real x, bool allowExtrapolation)
```

The last boolean indicates, if values outside the initial x-range are allowed to be calculated. This parameter is optional and is by default set to `false`. As a simple example, assume that we are given a grid of x-values $0.0, 1.0, ..., 4.0$ with corresponding y-values produced by the exponential function. We are interested in the linearly interpolated value at $x = 1.5$. Example code for the `LinearInterpolation` class is given below.

In the example below, we set up a grid of x values and generate the corresponding y values with the exponential function. After this setup, we ask for a linearly interpolated value.

```
#include<vector>

void testingInterpolations1(){

        std::vector<Real> xVec(5), yVec(xVec.size());

        xVec[0]=0.0; yVec[0]=std::exp(0.0);
        xVec[1]=1.0; yVec[1]=std::exp(1.0);
        xVec[2]=2.0; yVec[2]=std::exp(2.0);
        xVec[3]=3.0; yVec[3]=std::exp(3.0);
        xVec[4]=4.0; yVec[4]=std::exp(4.0);

        LinearInterpolation linInt(xVec.begin(), xVec.end(), yVec.begin());

        std::cout << "Exp at 0.0  " << linInt(0.0) << std::endl;
        std::cout << "Exp at 0.5  " << linInt(0.5) << std::endl;
        std::cout << "Exp at 1.0  " << linInt(1.0) << std::endl;
}
```

The output of the program is

```
Exp at 0.0 1
Exp at 0.5 1.85914
Exp at 1.0 2.71828
```

A very popular interpolation is the cubic spline interpolation. The natural cubic spline is a cubic spline whose second order derivatives are 0 at the endpoints. The example below shows the setup of this interpolation for a volatility interpolation example

```
#include<map>

void testingInterpolations2(){

        std::vector<Real> strikeVec(5), volVec(strikeVec.size());

        strikeVec[0]=70.0; volVec[0]=0.241;
        strikeVec[1]=80.0; volVec[1]=0.224;
        strikeVec[2]=90.0; volVec[2]=0.201;
        strikeVec[3]=100.0; volVec[3]=0.211;
        strikeVec[4]=110.0; volVec[4]=0.226;

        CubicNaturalSpline  natCubInt(strikeVec.begin(),strikeVec.end(),
                                                        volVec.begin());

        std::cout << "Vol at 70.0  " << natCubInt(70.0) << std::endl;
        std::cout << "Vol at 75.0  " << natCubInt(75.0) << std::endl;
        std::cout << "Vol at 79.0  " << natCubInt(79.0) << std::endl;
}
```

The output is

```
Vol at 70.0 0.241
Vol at 75.0 0.233953
Vol at 79.0 0.226363
```

For a general cubic spline interpolation, `QuantLib` provides the class `CubicInterpolation`. There are a lot of different options to set up such a spline. For example, one can ask

- Do I want monotonicity in my interpolation?
- Which method should be used to calculate derivatives given a discrete set of points?
- Which boundary conditions should be satisfied? For example, we could set the first derivative at the left endpoint to be 1.0.

The current derivative methods are the `Spline` and `Kruger` procedures.

The boundary conditions are defined as

- `NotAKnot`: ensures a continuous third derivative at x_1, x_{n-1}.
- `FirstDerivative`: match the slope at end point
- `SecondDerivative`: match the convexity at end point
- `Periodic`: match slope(first derivative) and convexity(second derivative) at both ends
- `Lagrange`: match end-slope to the slope of the cubic that matches the first four data points at the respective end

The constructor is then given as

```
CubicInterpolation(       const I1& xBegin,
                          const I1& xEnd,
                          const I2& yBegin,
                          CubicInterpolation::DerivativeApprox da,
                          bool monotonic,
                          CubicInterpolation::BoundaryCondition leftCond,
                          Real leftConditionValue,
                          CubicInterpolation::BoundaryCondition rightCond,
                          Real rightConditionValue)
```

To illustrate an explicit construction, we will setup a natural cubic spline manually, without the convenient constructor introduced before. The previously introduced interpolated volatility example is simply rewritten with a cubic spline with a manual setup. We will not require the spline to be monotonic and will enforce the second derivative to be zero by using the `CubicInterpolation::SecondDerivative` option.

Example code is given on the next slide. To test the result, the simple `NaturalCubicInterpolation` class is constructed too. The output of the corresponding program is

```
Nat Cub:  0.233953
Nat Cub Manual:  0.233953
```

Obviously, the interpolations produce the same result. We will not show any examples regarding the 2D interpolations, since these classes are not tested yet.

```
void testingInterpolations3(){


        std::vector<Real> strikeVec(5), volVec(strikeVec.size());

        strikeVec[0]=70.0; volVec[0]=0.241;
        strikeVec[1]=80.0; volVec[1]=0.224;
        strikeVec[2]=90.0; volVec[2]=0.201;
        strikeVec[3]=100.0; volVec[3]=0.211;
        strikeVec[4]=110.0; volVec[4]=0.226;

        CubicNaturalSpline  natCubInt(strikeVec.begin(),strikeVec.end(),
                                                            volVec.begin());

        CubicInterpolation natCubIntManual(strikeVec.begin(),strikeVec.end(),volVec.begin(),
                                CubicInterpolation::Spline, false,
                                CubicInterpolation::SecondDerivative, 0.0,
                                CubicInterpolation::SecondDerivative, 0.0);

        std::cout << "Nat Cub: " << natCubInt(75.0) << std::endl;
        std::cout << "Nat Cub Manual: " << natCubIntManual(75.0) << std::endl;



}
```

A very important issue is the behavior of the interpolations in the case where the original values change. Assume again that you have created `x,y` vectors called `xVec, yVec`. If you change any value of one of the input vectors you need to update your interpolation via the `update()` function. You may ask yourself why this is necessary. You might even remember that we have passed pointers to the constructors and no internal copies of the vectors were made. Consequently, the interpolation should simply look up the new values at the pointed addresses.

The point here is that the interpolation calculates some coefficients after construction and doesn't recalculate them again, if the user doesn't tell it to do it via `update`. Consider the cubic example below

$$f(x) = a + b(x - x_i) + c(x - x_i)^2 + d(x - x_i)^3 \text{ for } x \in [x_i, x_{i+1}]$$

The coefficient parameters $a, ..., d$ are calculated once after the constructor is called, given all input values. It would be inefficient to calculate these values each time the function is called, since the only remaining true variable after construction is the value x. However, if one of the input values changes, you can enforce the recalculation once by calling `update()`. This will update the parameters $a, ..., d$. We will show this in an example where we use the initially introduced linear interpolation of exponential function values.

```
#include<vector>

void testingInterpolations4(){

        std::vector<Real> xVec(5), yVec(xVec.size());

        xVec[0]=0.0; yVec[0]=std::exp(0.0);
        xVec[1]=1.0; yVec[1]=std::exp(1.0);
        xVec[2]=2.0; yVec[2]=std::exp(2.0);
        xVec[3]=3.0; yVec[3]=std::exp(3.0);
        xVec[4]=4.0; yVec[4]=std::exp(4.0);

        LinearInterpolation linInt(xVec.begin(), xVec.end(), yVec.begin());

        std::cout << "Exp at 0.5 original " << linInt(0.5) << std::endl;
        yVec[1]=std::exp(5.0);
        std::cout << "Exp at 0.5 resetted not updated:" << linInt(0.5) << std::endl;
        linInt.update();
        std::cout << "Exp at 0.5 updated:" << linInt(0.5) << std::endl;
}
```

In this example we have set the value at $x = 1.0$ to $y = \exp(5.0)$ instead of $y = 1.0$. This implies a higher y value at the point $x = 0.5$. The output of the function is

```
Exp at 0.5 original 1.85914
Exp at 0.5 resetted not updated:1.85914
Exp at 0.5 updated:74.7066
```

The math section has a matrix library to perform the standard matrix operations. The given `Matrix` class is meant to be a true Linear Algebra object and not a container for other types. Matrix algebra is not the main focus of `QuantLib`, other libraries are more advanced and efficient in this area. It does not mean that `QuantLib` is extremely slow. However, other programming languages such as `Matlab, Octave` are designed with respect to a fast performance in matrix operations and will outperform other libraries. If you need quick matrix results, feel free to use `QuantLib`. The standard constructor is

```
Matrix A(Size rows, Size columns)
```

for a matrix and

```
Array v(Size i)
```

for a vector. The elements are accessed via

`A[i][j]` or `v[i]`

The matrix rows and columns are returned by `rows()` and `columns()` functions. The length of the vector by the `size()` function. The matrix defines a variety of different iterators for iterations through columns or rows. Similarly, vector iterators are available. Also, the standard operators $*, +, -$ are defined for matrix-matrix, matrix-vector and matrix-scalar operations.

The following, self explaining static functions are available

- `Matrix transpose(const Matrix&)`
- `Matrix inverse(const Matrix& m)`
- `Real determinant(const Matrix& m)`
- `outerProduct(const Array& v1, const Array& v2)`

Example code, which uses some of the functions is given below

```
void testingMatrix1(){

        Matrix A(3,3);
        A[0][0]=0.2;A[0][1]=8.4;A[0][2]=1.5;
        A[1][0]=0.6;A[1][1]=1.4;A[1][2]=7.3;
        A[2][0]=0.8;A[2][1]=4.4;A[2][2]=3.2;

        Real det=determinant(A);
        QL_REQUIRE(!close(det,0.0),"Non invertible matrix!");

        Matrix invA=inverse(A);

        std::cout << A << std::endl;
        std::cout << "---------------" << std::endl;
        std::cout << transpose(A) << std::endl;
        std::cout << "---------------" << std::endl;
        std::cout << det << std::endl;
        std::cout << "---------------" << std::endl;
        std::cout << invA << std::endl;
        std::cout << "---------------" << std::endl;
        std::cout << A*invA << std::endl;
        std::cout << "---------------" << std::endl;
}
```

The `close` function checks for equality with respect to some multiple of machine precision. The output of the function is

```
| 0.2 8.4 1.5 |
| 0.6 1.4 7.3 |
| 0.8 4.4 3.2 |
--------------
| 0.2 0.6 0.8 |
| 8.4 1.4 4.4 |
| 1.5 7.3 3.2 |
--------------
29.68
--------------
| -0.931267 -0.683288 1.99528 |
| 0.132075 -0.0188679 -0.0188679 |
| 0.0512129 0.196765 -0.160377 |
--------------
| 1 5.55112e-017 0 |
| -5.55112e-017 1 0 |
| 0 0 1 |
```

Furthermore, various decompositions are available. For example

- `CholeskyDecomposition`: $A = UU^T$ with U being an lower triangular with positive diagonal entries.
- `SymmetricSchurDecomposition`: $A = UDU^T$ with D being the diagonal eigenvalue matrix and U a matrix containing the eigenvectors.
- `SVD` (Singular Value Decomposition): $A = UDV$ with U, V being orthogonal matrices, D being a nonnegative diagonal.
- `pseudoSqrt`: $A = SS^T$, the implementation allows to specify the internally used algorithm to achieve this.

Furthermore, a QR solver is available in the `qrSolve` function. Example code calling the functions above is available on the next slide.

```
void testingMatrix2(){

Matrix A(3,3);
A[0][0] = 1.0;  A[0][1] = 0.9;  A[0][2] = 0.7;
A[1][0] = 0.9;  A[1][1] = 1.0;  A[1][2] = 0.4;
A[2][0] = 0.7;  A[2][1] = 0.4;  A[2][2] = 1.0;

SymmetricSchurDecomposition schurDec(A);
SVD svdDec(A);

        std::cout << "Schur Eigenvalues:" << std::endl;
        std::cout << schurDec.eigenvalues()  << std::endl;
        std::cout << "---------------" << std::endl;
        std::cout << "Schur Eigenvector Mat:" << std::endl;
        std::cout << schurDec.eigenvectors() << std::endl;
        std::cout << "---------------" << std::endl;
        std::cout << "Cholesky:" << std::endl;
        std::cout << CholeskyDecomposition(A) << std::endl;
        std::cout << "---------------" << std::endl;
        std::cout << "SVD U:" << std::endl;
        std::cout << svdDec.U() << std::endl;
        std::cout << "---------------" << std::endl;
        std::cout << "SVD V:" << std::endl;
        std::cout << svdDec.V() << std::endl;
        std::cout << "---------------" << std::endl;
        std::cout << "SVD Diag D:" << std::endl;
        std::cout << svdDec.singularValues() << std::endl;
        std::cout << "---------------" << std::endl;
        std::cout << "Pseudo Sqrt:" << std::endl;
        std::cout << pseudoSqrt(A) << std::endl;
}
```

The output is

```
Schur Eigenvalues:
2.35364; 0.616017; 0.0303474
--------------
Schur Eigenvector Mat:
| 0.643624 0.11108 0.757238 |
| 0.576635 0.58018 -0.575225 |
| 0.50323 -0.806878 -0.309365 |
--------------
Cholesky:
| 1 0 0 |
| 0.9 0.43589 0 |
| 0.7 -0.527656 0.481227 |
--------------
SVD U:
| 0.643624 -0.11108 0.757238 |
| 0.576635 -0.58018 -0.575225 |
| 0.50323 0.806878 -0.309365 |
--------------
SVD V:
| 0.643624 -0.11108 0.757238 |
| 0.576635 -0.58018 -0.575225 |
| 0.50323 0.806878 -0.309365 |
--------------
SVD Diag D:
2.35364; 0.616017; 0.0303474
--------------
Pseudo Sqrt:
| 1 0 0 |
| 0.9 0.43589 0 |
| 0.7 -0.527656 0.481227 |
```

One of the most important tools, in particular in calibration procedures, is an optimizer of a function $f : \mathbb{R}^n \to \mathbb{R}$. The typical problem that requires an optimization is a least squares problem. For example, the typical problem is: find a model parameter set such that some cost function is minimized. The available optimizers in `QuantLib` are

- `LevenbergMarquardt`
- `Simplex`
- `ConjugateGradient` (line search based method)
- `SteepestDescent` (line search based method)
- `BFGS` (line search based method) `QL 1.0.0`

To setup up an optimizer, we need to define the end criteria which lead to a successful optimization. They are summarized in the `EndCriteria` class whose constructor os

```
EndCriteria(Size maxIterations,
            Size minStationaryStateIterations,
            Real rootEpsilon,
            Real functionEpsilon,
            Real gradientNormEpsilon);
```

The input parameters of this class are

- Maximum iterations: restrict the maximum number of solver iterations.
- Minimum stationary state iterations: give a minimum number of iterations at stationary point (for both, function value stationarity and root stationarity).
- Function ϵ: stop if absolute difference of current and last function value is below ϵ.
- Root ϵ: stop if absolute difference of current and last root value is below ϵ.
- Gradient ϵ: stop if absolute difference of the norm of the current and last gradient is below ϵ.

Note that not all of the end criteria are needed in each optimizer. I haven't found any optimizer which checks for the gradient norm. The simplex optimizer is the only one checking for root epsilon. Most of the optimizers check for the maximum number of iterations and the function epsilon criteria.

Next, we need to specify if any constraints on the optimal parameter values are given. This can be specified via the classes derived from the `Constraint` class

- `NoConstraint`
- `PositiveConstraint`: require all parameters to be positive
- `BoundaryConstraint`: require all parameters to be in an interval
- `CompositeConstraint`: require two constraints to be fulfilled at the same time

The last object that needs to be specified for the optimization is the `CostFunction`, which implements the function which needs to be minimized. For a setup of your own function, you need to implement a class which derives from the `CostFunction` class and implements the following `virtual` functions

- `Real value(const Array& x)` the function value at x. This is a pure virtual function, an implementation is required.
- `Array values(const Array& x)`: the value $f(x)$ if $f : \mathbb{R}^n \to \mathbb{R}^m$ maps to higher dimensions. Currently only required for the Levenberg-Marquardt minimization, which will be discussed later. Return a one dimensional array with `value(x)` for other methods.
- `void gradient(Array& grad, const Array& x)` write the vector which will contain the gradient `grad` and the value where this gradient is evaluated `x`. This is optional. A finite difference method will be used if nothing is implemented by derived classes.
- `Real valueAndGradient(Array& grad, const Array& x)` return the function value at `x`, and write the gradient at `x`.

To test the optimizer we will calculate the minimum of the Rosenbrock function, which is a classical test problem for optimization algorithms. The function is defined as

$$f(x, y) := (1 - x)^2 + 100(y - x^2)^2$$

the minimum is located in a valley at $(x, y) = (1, 1)$ with $f(x, y) = 0$.

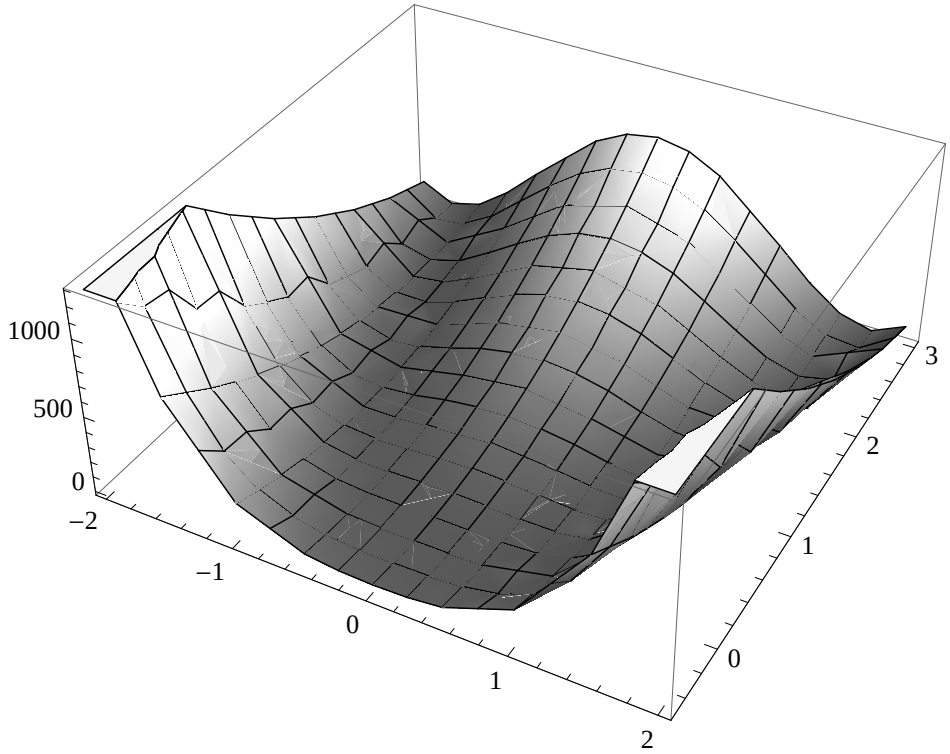


Figure: Rosenbrock Function

To test the problem, we define a class `RosenBrockFunction` which inherits from the `CostFunction`. The value of the function is returned by `value`.

```
class RosenBrockFunction: public CostFunction{
public:
        Real value(const Array& x) const{
                QL_REQUIRE(x.size()==2,"Rosenbrock function is 2-dim.");
                Real res=(1-x[0])*(1-x[0]);
                res+=100.0*(x[1]-x[0]*x[0])*(x[1]-x[0]*x[0]);
                return res;
        }

        Disposable<Array> values(const Array& x) const{
                QL_REQUIRE(x.size()==2,"Rosenbrock function is 2-dim.");
                // irrelevant what you write in res for most of the optimizers
                // most of them are using value anyways. try with res[0]=100.0
                Array res(1); res[0]=value(x);
                return res;
        }
};
```

Next, setup the optimizers. We will test the `Simplex` and `ConjugateGradient` optimizers with starting value $(x, y) = (0.1, 0.1)$ and no constraints. The code is shown next.

```
void testOptimizer1(){

        Size maxIterations=1000;
        Size minStatIterations=100;
        Real rootEpsilon=1e-8;
        Real functionEpsilon=1e-9;
        Real gradientNormEpsilon=1e-5;

        EndCriteria myEndCrit(  maxIterations,
                                                minStatIterations,
                                                rootEpsilon,
                                                functionEpsilon,
                                                gradientNormEpsilon);

        RosenBrockFunction myFunc;
        NoConstraint constraint;

        Problem myProb1(myFunc, constraint, Array(2,0.1));
        Problem myProb2(myFunc, constraint, Array(2,0.1));

        Simplex solver1(0.1);
        ConjugateGradient solver2;

        EndCriteria::Type solvedCrit1=solver1.minimize(myProb1,myEndCrit);
        EndCriteria::Type solvedCrit2=solver2.minimize(myProb2,myEndCrit);

        std::cout << "Criteria Simplex:"<< solvedCrit1 << std::endl;
        std::cout << "Root Simplex:"<< myProb1.currentValue() << std::endl;
        std::cout << "Min F Value Simplex:"<< myProb1.functionValue() << std::endl;
        std::cout << "Criteria CG:"<< solvedCrit2 << std::endl;
        std::cout << "Root CG:"<< myProb2.currentValue() << std::endl;
        std::cout << "Min F Value CG:"<< myProb2.functionValue() << std::endl;

}
```

The output of the function is

```
Criteria Simplex:StationaryPoint
Root Simplex:[ 1; 1 ]
Min F Value Simplex:2.92921e-017
Criteria CG:StationaryFunctionValue
Root CG:[ 0.998904; 0.995025 ]
Min F Value CG:0.000776496
```

The Simplex algorithm finds the correct minimum, while the conjugate gradient version stops very close to it. As shown in the code, the final optimal values will be stored in the `Problem` instance which is passed to the minimization function **by reference**. The minimum can be obtained via `currentValue()` and the function value via `functionValue()`.

Furthermore, the criteria which leads to a stopping of the algorithm is returned. It should be checked that this criteria is not of type `EndCriteria::None`.

Finally, we will show how a setup for the Levenberg Marquardt algorithm works. We will do this in the framework of the following problem:

Assume you are given 4 call prices C_1, C_2, C_3, C_4 in a **flat volatility** world. Each call C_i has a known strike K_i. The only variables which are not known to you are the spot and single volatility which were used for the price calculation. Consequently, the 2 unknowns are (σ, S_0). You could try to solve the problem via a least squares problem by minimizing the following function

$$f(\sigma, S_0) = \sum_{i=1}^{4} (C(K_i, \sigma, S_0) - C_i)^2$$

This is the typical problem where Levenberg-Marquardt is considered as the optimizer. `QuantLib` uses the MINPACK minimization routine, where a general function

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

is considered. Performing the optimization yields the minimum x of the following cost function c

$$c(x) = \sum_{i=}^{m} f_i(x)^2$$

where $f_i : \mathbb{R}^n \to \mathbb{R}$ are the components of the m dimensional function. To solve the problem above, we will reuse the function `blackScholesPrice` which has been introduced before.

The setup for the cost function is shown below.

```
class CallProblemFunction: public CostFunction{
private:

        Real C1_,C2_,C3_,C4_,K1_,K2_,K3_,K4_;
        Rate rd_,rf_;
        Integer phi_;
        Time tau_;
public:
CallProblemFunction(const Rate& rd, const Rate& rf, const Time& tau, const Integer& phi,
                                    const Real& K1,const Real& K2,const Real& K3,const Real& K4,
                                    const Real& C1,const Real& C2,const Real& C3,const Real& C4)
                                    :rd_(rd), rf_(rf), phi_(phi), tau_(tau),
                                    C1_(C1),C2_(C2),C3_(C3),C4_(C4),
                                    K1_(K1),K2_(K2),K3_(K3),K4_(K4){}

        Real value(const Array& x) const{

                Array tmpRes=values(x);
                Real res=tmpRes[0]*tmpRes[0];
                res+=tmpRes[1]*tmpRes[1];
                res+=tmpRes[2]*tmpRes[2];
                res+=tmpRes[3]*tmpRes[3];
                return res;
        }

        Disposable<Array> values(const Array& x) const{

                        Array res(4);
                        res[0]=blackScholesPrice(x[0],K1_,rd_,rf_,x[1],tau_,phi_)-C1_;
                        res[1]=blackScholesPrice(x[0],K2_,rd_,rf_,x[1],tau_,phi_)-C2_;
                        res[2]=blackScholesPrice(x[0],K3_,rd_,rf_,x[1],tau_,phi_)-C3_;
                        res[3]=blackScholesPrice(x[0],K4_,rd_,rf_,x[1],tau_,phi_)-C4_;

                        return res;
        }
};
```

```
void testOptimizer2(){
        // setup of market parameters
        Real spot=98.51;
        Volatility vol=0.134;
        Real K1=87.0, K2=96.0, K3=103.0, K4=110.0;
        Rate rd=0.002, rf=0.01;
        Integer phi=1;
        Time tau=0.6;
        // calculate Black Scholes prices
        Real C1=blackScholesPrice(spot,K1,rd,rf,vol,tau,phi);
        Real C2=blackScholesPrice(spot,K2,rd,rf,vol,tau,phi);
        Real C3=blackScholesPrice(spot,K3,rd,rf,vol,tau,phi);
        Real C4=blackScholesPrice(spot,K4,rd,rf,vol,tau,phi);

        CallProblemFunction optFunc(rd, rf, tau, phi,K1, K2, K3, K4, C1, C2, C3, C4);

        Size maxIterations=1000;
        Size minStatIterations=100;
        Real rootEpsilon=1e-5;
        Real functionEpsilon=1e-5;
        Real gradientNormEpsilon=1e-5;

        EndCriteria myEndCrit(maxIterations,minStatIterations, rootEpsilon,
                functionEpsilon, gradientNormEpsilon);

        Array startVal(2); startVal[0]=80.0; startVal[1]=0.20;
        NoConstraint constraint;
        Problem myProb(optFunc, constraint, startVal);
        LevenbergMarquardt solver;
        EndCriteria::Type solvedCrit=solver.minimize(myProb,myEndCrit);

        std::cout << "Criteria :"<< solvedCrit << std::endl;
        std::cout << "Root :" << myProb.currentValue() << std::endl;
        std::cout << "Min Function Value :"     << myProb.functionValue() << std::endl;
}
```

In the test code on the previous slide we have chosen $S_0 = 98.51$ and $\sigma = 0.134$ to calculate the prices $C_1, ..., C_4$. We have then constructed the optimization algorithm which should roughly recover the input variables. The output of the function is

```
Criteria :StationaryFunctionValue
Root :[ 98.51; 0.134 ]
Min Function Value :3.40282e+038
```

which is a perfect matching of the original variables. The setup in this example has represented the typical calibration problem, where call prices are given by the market and we try to match these prices with a given model (e.g. Heston). The objective is to find model parameters and calculate the model prices at the strikes K_i. In case of a successful model calibration, the market prices should be matched up to a certain error term. For a more complex and flexible setup, we should pass the number of option prices in a container. The function `blackScholesPrice` can be replaced by any vanilla pricer with some degrees of freedom. It has to be pointed out that the returned array in the function `values()` needs a dimension which is at least the dimension of the parameter vector x. In mathematical terms: for

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

we need to have $m \geq n$. Otherwise you will receive a `MINPACK: improper input parameters` exception. This is probably required because of a potentially ill posed problem in case this doesn't hold.

- Generate 6 (x_i, y_i) pairs of a second order parabola

$$f(x) = a + bx + cx^2$$

  by perturbing the $f(x)$ values by some error term $\pm\epsilon$.
- Write an optimizer which accepts the 6 pairs and does a least square parabola fit to these values.
- Check, how well the original variables a, b, c are approximated.

The basis for all random number machines is a basic generator which generates uniform random numbers. Let $X \sim U[0,1]$ be such a uniform random variable. A random number for any distribution can be generated by a transformation of X. This can be done by taking the inverse cumulative distribution F^{-1} and evaluating $F^{-1}(X)$. Other algorithms transform X to some other distribution without the cdf (e.g. Box Muller). There are several uniform distribution generators in `QuantLib`

- `KnuthUniformRng` is the uniform random number generator by Knuth
- `LecuyerUniformRng` it the L'Ecuyer random number generator
- `MersenneTwisterUniformRng` is the famous Mersenne-Twister algorithm

Each of the constructors accepts a seed of type `long` which initializes the corresponding deterministic sequence. The user can also request a random seed by calling the `get()` method of an instance of the `SeedGenerator` class with `SeedGenerator::instance().get()`. The `SeedGenerator` class is a singleton, which prevents any default and copy construction. Calling `SeedGenerator::instance().get()` repeatedly yields a different seed. A sample from the given generator can be obtained by calling

```
Sample<Real> next() const
```

Calling the function repeatedly returns new random numbers.

The `Sample` type is a template class located in

`<ql/methods/montecarlo/sample.hpp>`

Its basic construction is `Sample<T>(T value, Real weight)` with `T` being a template class. The value and weight are public variables which can be accessed by the `mySample.value` or `mySample.weight` operators respectively. In the code below we illustrate the setup for the introduced random number generators.

```
void testingRandomNumbers1(){

        BigInteger seed=SeedGenerator::instance().get();
        std::cout << "Seed 1: " << seed << std::endl;

        MersenneTwisterUniformRng       unifMt(seed);
        LecuyerUniformRng unifLec(seed);
        KnuthUniformRng unifKnuth(seed);


        std::cout << "Mersenne Twister Un:" << unifMt.next().value << std::endl;
        std::cout << "Lecuyer Un:" << unifLec.next().value << std::endl;
        std::cout << "Knut Un:" << unifKnuth.next().value << std::endl;

        seed=SeedGenerator::instance().get();
        std::cout << "---------------------------" << std::endl;
        std::cout << "Seed 2: " << seed << std::endl;
        std::cout << "---------------------------" << std::endl;
        std::cout << "Mersenne Twister Un:" << unifMt.next().value << std::endl;
        std::cout << "Lecuyer Un:" << unifLec.next().value << std::endl;
        std::cout << "Knut Un:" << unifKnuth.next().value << std::endl;

}
```

The output of the program is

```
Seed 1:  873212726
Mersenne Twister Un:0.679093
Lecuyer Un:0.149121
Knut Un:0.874008
-------------------------
Seed 2:  -2046984499
-------------------------
Mersenne Twister Un:0.72727
Lecuyer Un:0.0335365
Knut Un:0.584285
```

Note that your output will differ from this one since the seed will be different. A normal random variable can for example be generated by setting up a

```
<RNG>BoxMullerGaussianRng(const RNG& uniformGenerator)
```

class which is the classical Box Muller algorithm. The template constructor accepts any of the introduced uniform random number generators. The following example code below shows a valid setup

```
void testingRandomNumbers2(){

        BigInteger seed=12324;
        MersenneTwisterUniformRng       unifMt(seed);
        BoxMullerGaussianRng<MersenneTwisterUniformRng> bmGauss(unifMt);

        std::cout << bmGauss.next().value << std::endl;
        std::cout << bmGauss.next().value << std::endl;
        std::cout << bmGauss.next().value << std::endl;
        std::cout << bmGauss.next().value << std::endl;
        std::cout << bmGauss.next().value << std::endl;

}
```

The output of this code is

```
-1.17568
0.1411
1.56958
-0.0267368
-0.822068
```

In this case, you should receive the same output as the one shown above. Another class is the `CLGaussianRng` random number generator, which uses the properties of the central limit theorem.

As already discussed, given a uniform random variable X, it is possible to generate any random number by the evaluation of the inverse cumulative distribution function at X. QuantLib provides the following template class to generate a sequence of random numbers from the inverse cumulative distribution:

```
<class USG, class IC> class InverseCumulativeRsg (
const USG& uniformSequenceGenerator, const IC& inverseCumulative)
```

The uniform sequence generator can be any of the introduced generators, but needs to return a sample sequence instead of a single sample. So, instead of

```
Sample<Real> next() const
```

the passed class needs to implement

```
Sample<std::vector<Real>> nextSequence() const
```

QuantLib offers the template class `RandomSequenceGenerator` with constructor

```
<RNG>RandomSequenceGenerator(Size dimensionality, const RNG& rng)
```

To transform the standard Mersenne-Twister generator to a sequence generator we would construct a `RandomSequenceGenerator` with

```
BigInteger seed=12324;
MersenneTwisterUniformRng unifMt(seed);
RandomSequenceGenerator<MersenneTwisterUniformRng> unifMtSeq(10,unifMt);
```

The passed inverse cumulative class needs to implement a

```
Real operator(const Real& x) const
```

operator. Although QuantLib offers some distribution functions, we will not use them here but use the boost distributions instead. Boost offers more distributions at the current stage. Another reason is that we would like to show the advantage of a template setup of the QuantLib classes. This setup allows to easily incorporate other libraries in the `QuantLib` framework. In the example below we will generate random numbers from the Fisher-Tippett distribution, where the density is given as

$$f(x) = \frac{e^{(\alpha-x)/\beta - e^{(\alpha-x)/\beta}}}{\beta}$$

with location parameter α and scale parameter β.

In the current version, QuantLib does not provide an implementation of this distribution class. However, we will once again use the `bind` class to construct a boost function which can be passed as the `const IC& inverseCumulative` parameter. For example, the constructor of the final random sequence generator can be constructed as

```
InverseCumulativeRsg<RandomSequenceGenerator<MersenneTwisterUniformRng>,
    boost::function<Real (Real)>> myEvInvMt(unifMtSeq,invEv);
```

where `unifMtSeq` is the uniform number sequence generator and `invEv` is the inverse Extreme Value function which is passed as a boost function. The sequence can then be obtained by calling

```
std::vector<Real> sample=myEvInvMt.nextSequence().value;
```

Example code is shown next.

```
#include <boost/math/distributions.hpp>
#include <boost/function.hpp>


Real evInv(boost::math::extreme_value_distribution<> d, const Real& x){
                return quantile(d,x);
}

void testingRandomNumbers3(){

        boost::math::extreme_value_distribution<> d(0.0,0.1);
        boost::function<Real (Real)> invEv=boost::bind(evInv,d,_1);

        // Mersenne Twister setup
        BigInteger seed=12324;
        MersenneTwisterUniformRng       unifMt(seed);
        // sequence setup
        RandomSequenceGenerator<MersenneTwisterUniformRng> unifMtSeq(10,unifMt);
        //

        InverseCumulativeRsg<RandomSequenceGenerator<MersenneTwisterUniformRng>,
                boost::function<Real (Real)>> myEvInvMt(unifMtSeq,invEv);

        std::vector<Real> sample=myEvInvMt.nextSequence().value;
        BOOST_FOREACH(Real x, sample) std::cout << x << std::endl;
}
```

The output of the code is

```
0.344719
0.122182
-0.0639099
0.0490113
0.134179
0.0353269
-0.0838453
0.0714751
0.0538163
0.18975
```

An important class of number series is the class of quasi random numbers, also known as low discrepancy numbers. QuantLib provides the following types

- `LatticeRsg`: lattice rule quasi random numbers
- `FaureRsg`: Faure quasi random numbers
- `HaltonRsg`: Halton
- `SobolRsg`: Sobol

The basic constructor is of the form

```
SomeQRGenerator myGen(Size dimension, optional parameters)
```

The numbers can be obtained in a vector via

```
Sample<std::vector<Real>>nextSequence()
```

Example code with a setup of 5 dimensional Faure, Sobol and Halton uniform random numbers is shown below

```
void testingRandomNumbers4(){
   Size dim=5;

   SobolRsg sobolGen(dim);
   HaltonRsg haltonGen(dim);
   FaureRsg faureGen(dim);

   std::vector<Real> sampleSobol(sobolGen.dimension()),
                                        sampleHalton(haltonGen.dimension()),
                                        sampleFaure(faureGen.dimension());

   sampleSobol=sobolGen.nextSequence().value;
   sampleHalton=haltonGen.nextSequence().value;
   sampleFaure=faureGen.nextSequence().value;

   BOOST_FOREACH(Real x, sampleSobol) std::cout << "S:" << x << std::endl;
   BOOST_FOREACH(Real x, sampleHalton) std::cout << "H:" << x << std::endl;
   BOOST_FOREACH(Real x, sampleFaure) std::cout << "F:" << x << std::endl;

}
```

The output is

```
S:0.5
S:0.5
S:0.5
S:0.5
S:0.5
H:0.621022
H:0.841062
H:0.61028
H:0.75794
H:0.813697
F:0.2
F:0.2
F:0.2
F:0.2
F:0.2
```

Finally, we give an example of sample statistics by generating standard normal random variables with pseudo- and quasi-random Sobol numbers. The mean, variance, skewness and excess kurtosis are printed and compared. For the standard normal we expect the numbers to be $0.0, 1.0, 0.0, 0.0$.

```
void testingRandomNumbers5(){

   SobolRsg sobolGen(1);
   // Mersenne Twister setup
        BigInteger seed=12324;
        MersenneTwisterUniformRng       unifMt(seed);
        BoxMullerGaussianRng<MersenneTwisterUniformRng> bmGauss(unifMt);

    IncrementalStatistics boxMullerStat, sobolStat;
        MoroInverseCumulativeNormal invGauss;

        Size numSim=10000;
        Real currSobolNum;

        for(Size j=1;j<=numSim;++j){
                boxMullerStat.add(bmGauss.next().value);

                currSobolNum=(sobolGen.nextSequence().value)[0];
                sobolStat.add(invGauss(currSobolNum));
        }

        std::cout << "BoxMuller Mean:" << boxMullerStat.mean() << std::endl;
        std::cout << "BoxMuller Var:" << boxMullerStat.variance() << std::endl;
        std::cout << "BoxMuller Skew:" << boxMullerStat.skewness() << std::endl;
        std::cout << "BoxMuller Kurtosis:" << boxMullerStat.kurtosis() << std::endl;
        std::cout << "-----------------------" << std::endl;
        std::cout << "Sobol Mean:" << sobolStat.mean() << std::endl;
        std::cout << "Sobol Var:" << sobolStat.variance() << std::endl;
        std::cout << "Sobol Skew:" << sobolStat.skewness() << std::endl;
        std::cout << "Sobol Kurtosis:" << sobolStat.kurtosis() << std::endl;

}
```

The output of the code is

```
BoxMuller Mean:0.00596648
BoxMuller Var:1.0166
BoxMuller Skew:0.0210064
BoxMuller Kurtosis:-0.0340477
-----------------------
Sobol Mean:-0.000236402
Sobol Var:0.998601
Sobol Skew:-7.74057e-005
Sobol Kurtosis:-0.0207681
```

which shows that the Sobol numbers are closer to the expected moments than pseudo-random numbers. For illustration purposes we have used the Moro inverse cumulative normal distribution implementation, which is available in QuantLib. Also, the moments were calculated with the `IncrementalStatistics` class.

The QuantLib math section provides various 2− dimensional copulas such as the Gaussian-, Gumbel-, Clayton or Independent-Copula. The constructor accepts the copula specific parameters and the copula value is returned by the operator

```
operator()(Real x, Real y)
```

Example code is shown below

```
void testCopulas1(){

        GaussianCopula gaussCop(0.7);
        GumbelCopula gumbCop(1.7);
        Real x=0.7,y=0.2;

        std::cout << "Gauss Copula:"  << gaussCop(x,y) << std::endl;
        std::cout << "Gumbel Copula:" << gumbCop(x,y) << std::endl;
}
```

The output is

```
Gauss Copula:0.194937
Gumbel Copula:0.186116
```

1. Mathematical Tools
   - Integration
   - Solver
   - Exercise
   - Interpolation
   - Matrix
   - Optimizer
   - Exercise
   - Random Numbers
   - Copulas
2. Fixed Income
   - Indexes
   - Interest Rate
   - Yield Curve Construction
3. Volatility Objects
   - Smile Sections
4. Payoffs and Exercises
5. Black Scholes Pricer
   - Black Scholes Calculator
6. Stochastic Processes
   - Generalized Black Scholes Process
   - Ornstein Uhlenbeck Process
   - Heston Process
   - Bates Process

The index classes are used for a representation of known indexes, such as the BBA Libor or Euribor indexes. The properties can depend on several variables such as the underlying currency and maturity. Imagine you are observing a 1 Month EUR Libor rate and you are interested in the interest rate for a given nominal as well as the settlement details of a contract based on this rate. The technical details are given on www.bbalibor.com. Going through the details yields that

- the rate refers to the Actual360() daycounter,
- the value date will be 2 business days after the fixing date, following the TARGET calendar,
- the modified following business day convention is used,
- if the deposit is made on the final business day of a particular calendar month, the maturity of the deposit shall be on the final business day of the month in which it matures (e.g. from 28th of February to 31st of March, not 28th of March)

These properties are different for 1 week rates. In addition, the conventions depend on the underlying currency. Fortunately, you don't have to specify these properties for most of the common indexes as they are implemented in QuantLib.

For example, you can initialize a `EURLibor1M` index with a default constructor. This index inherits from the `IborIndex` class, which inherits from the `InterestRateIndex` class. The classes offer several functions such as

- `std::string name()`
- `Calendar fixingCalendar()`
- `DayCounter& dayCounter()`
- `Currency& currency()`
- `Period tenor()`
- `BusinessDayConvention businessDayConvention()`

Example code is shown below

```
void testingIndexes1(){

EURLibor1M index;

        std::cout << "Name:" << index.familyName() << std::endl;
        std::cout << "BDC:"<< index.businessDayConvention() << std::endl;
        std::cout << "End of Month rule?:"<< index.endOfMonth() << std::endl;
        std::cout << "Tenor:" << index.tenor() << std::endl;
        std::cout << "Calendar:" << index.fixingCalendar() << std::endl;
        std::cout << "Day Counter:" << index.dayCounter() << std::endl;
        std::cout << "Currency:" << index.currency() << std::endl;

}
```

The output is

```
Name:EURLibor
BDC:Modified Following
End of Month rule?:1
Tenor:1M
Calendar:JoinBusinessDays(London stock exchange, TARGET)
Day Counter:Actual/360
Currency:EUR
```

The returned properties are consistent with the ones published on the BBA site. Such an index is needed as an input parameter in several constructors, in particular the yield curve construction helpers, which need the exact properties of the rates. The specification of the rate index class allows for a compact definition of other constructors. Imagine that you have to construct a class with different Libor and Swap rates. Such a constructor would become large if all corresponding rate properties would be provided to the constructor. With the index classes, this is not a problem. Other indexes are available, for example the ISDA swap index `EurLiborSwapIsdaFixA` or the `BMAIndex`. See the `QuantLib` documentation for details.

The definition of the `InterestRate` class is given in `<ql/interestrate.hpp>`. The class represents the properties of various yield types, the constructor is given as

```
InterestRate(Rate r,
             const DayCounter& dc = Actual365Fixed(),
             Compounding comp = Continuous,
             Frequency freq = Annual);
```

The class provides standard functions to return the passed properties via

- `Rate rate()`
- `const DayCounter& dayCounter()`
- `Compounding compounding()`
- `Frequency frequency()`

Also, discount, compounding factors and equivalent rates can be calculated with

- `DiscountFactor discountFactor(const Date& d1, const Date& d2)`
- `Real compoundFactor(const Date& d1, const Date& d2)`
- `Rate equivalentRate(Date d1,Date d2, const DayCounter& resultDC, Compounding comp, Frequency freq=Annual)`

The compound and discount functions allow to adjust for different reference dates (option for Actual/Actual day counters). Also, all functions above are implemented with a `Time` instead of two date variables.

The rate type can be specified with the `Compounding` enumeration. The implemented rate types are

- `Simple`, $1 + r\tau$
- `Compounded`, $(1 + r)^{\tau}$
- `Continuous`, $e^{r\tau}$

The frequency can be specified to

- `NoFrequency` null frequency
- `Once` only once, e.g., a zero-coupon
- `Annual` once a year
- `Semiannual` twice a year
- `EveryFourthMonth` every fourth month
- `Quarterly` every third month
- `Bimonthly` every second month
- `Monthly` once a month
- `EveryFourthWeek` every fourth week
- `Biweekly` every second week
- `Weekly` once a week
- `Daily` once a day

Furthermore, a `static` function is provided which extracts the interest rate of any type from a compound factor.

- `InterestRate impliedRate(Real compound, const Date& d1, const Date& d2, const DayCounter& resultDC, Compounding comp, Frequency freq = Annual)`

This function is used by the yield curve classes which will be introduced later. Example code is shown below. The example shows the setup of an interest rate with a calculation of the compound/discount factors and the calculation of the equivalent semi-annual continuous rate. Given the compound factor, we extract the original rate with the `impliedRate` function.

```
void testingYields1(){

        DayCounter dc=ActualActual();
        InterestRate myRate(0.0341, dc, Simple, Annual);

        std::cout << "Rate: " << myRate << std::endl;
        Date d1(10,Sep,2009), d2=d1+3*Months;

        Real compFact=myRate.compoundFactor(d1,d2);
        std::cout << "Compound Factor:" << compFact << std::endl;
        std::cout << "Discount Factor:" << myRate.discountFactor(d1,d2) << std::endl;
        std::cout << "Equivalent Rate:" << myRate.equivalentRate(d1,d2,
                        dc,Continuous,Semiannual) << std::endl;

        Real implRate=InterestRate::impliedRate(compFact,d1,d2,dc,Simple,Annual);
        std::cout << "Implied Rate from Comp Fact:" << implRate << std::endl;

}
```

The output of the function is

```
Rate:  3.410000 % Actual/Actual (ISDA) simple compounding
Compound Factor:1.0085
Discount Factor:0.99157
Equivalent Rate:3.395586 % Actual/Actual (ISDA) continuous compounding
Implied Rate from Comp Fact:0.0341
```

There are various ways to construct a market consistent yield curve. The methodology depends on the liquidity of available market instruments for the corresponding market. Several choices have to be made; the interpolation procedure and the choice of the market instruments have to be specified. QuantLib allows to construct a yield curve as a

- `InterpolatedDiscountCurve`, construction given discount factors
- `InterpolatedZeroCurve`, construction given zero coupon bond rates
- `InterpolatedForwardCurve`, construction given forward rates
- `FittedBondDiscountCurve`, construction given coupon bond prices
- `PiecewiseYieldCurve`, piecewise construction given a mixture of market instruments (i.e. deposit rates, FRA/Future rates, swap rates).

We will discuss the first and last case here. The other cases are equivalent. Interested readers should take a look at the other useful term structures, such as the `QuantoTermStructure` or `InflationTermStructure`. It should be noted that the classes above derive from general abstract base classes, which can be used by the user in case she is interested in own yield curve implementations. All of the above constructors derive from the base class `YieldTermStructure`. The `YieldTermStructure` class derives from `TermStructure`, which is both, an `Observer` and `Observable`. This base class implements some useful functions. For example, functions are implemented which return the reference date, day counter, calendar, the minimum or maximum date for which the curve returns yields.

`YieldTermStructure` has the following public functions which are inherited by the previously introduced classes

- `DiscountFactor discount(const Date& d, bool extrapolate = false)`, also available with `Time` instead of `Date` input. Extrapolation can be enabled optionally.
- `InterestRate zeroRate(const Date& d, const DayCounter& resultDayCounter, Compounding comp,Frequency freq = Annual,bool extrapolate = false)`, also available with `Time` input.
- `InterestRate forwardRate(const Date& d1,const Date& d2,const DayCounter& dc, Compounding comp, Frequency freq = Annual, bool extrapolate = false)`, also available with `Time` input.
- `Rate parRate(const std::vector<Date>& dates, const DayCounter& dc, Frequency freq = Annual, bool extrapolate = false)` returns the par rate for a bond which pays on the specified dates.

All of the concrete yield classes derive from `YieldTermStructure` and thus inherit automatically the functions above. The only function that they have to implement is the pure virtual `discountImpl(Time)` function. Given the discount factor, all other rates can be derived. This allows for a convenient implementation of an own yield curve, without bothering about the calculations of the different rate types.

We will treat the `InterpolatedDiscountCurve` first. This construction procedure is suited for quotes such as the one given below on Reuters: a given set of discount factors is assigned to corresponding maturities. We have cut the discount factors beyond the maturity of 2 years for illustration purposes.

EUR	Yield	Discount	
TN	0.3148	0.9999656	14:40
1W	0.3083	0.9999072	14:40
1M	0.4225	0.9996074	14:40
2M	0.5443	0.9990040	14:40
3M	0.7242	0.9981237	14:40
6M	0.9614	0.9951358	14:40
9M	0.9372	0.9929456	14:40
1Y	1.0006	0.9899849	14:40
1Y3M	1.1120	0.9861596	14:40
1Y6M	M 1.2457	0.9815178	14:40
1Y9M	M 1.4358	0.9752363	14:40
2Y	1.6263	0.9680804	14:40

Figure: Reuters Discount Factor

The only object that we have to specify for the `InterpolatedDiscountCurve` is the interpolation type between the discount factors. For example, a common choice is the `LogLinear` interpolation, which interpolates linearly in the rates. The constructor accepts the vectors with the discount factors and dates. The first date has to be the reference of the discount curve, e.g. the date with discount factor 1.0! The constructor has the following implementation

```
InterpolatedDiscountCurve(
          const std::vector<Date>& dates,
          const std::vector<DiscountFactor>& dfs,
          const DayCounter& dayCounter,
          const Calendar& cal = Calendar(),
          const std::vector<Handle<Quote> >& jumps = std::vector<Handle<Quote> >(),
          const std::vector<Date>& jumpDates = std::vector<Date>(),
          const Interpolator& interpolator = Interpolator());
```

The constructors shows that jumps and jump times can be handed over, although this is optional. Jumps have to be a number in $[0, 1]$ and shift the whole yield curve downwards by the corresponding factor at the given date. For example, the yield curve can be shifted to be 98% of the original market curve after 3 months of the reference date. And again to 98% of the shifted curve after 4 months.

In the following example, we construct a yield curve based on the Reuters discount factors introduced before.

```
void testingYields2(){
        std::vector<Date> dates;
        std::vector<DiscountFactor> dfs;

        Calendar cal= TARGET();
        Date today(11,Sep,2009);
        EURLibor1M libor;
        DayCounter dc=libor.dayCounter();

        Natural settlementDays = 2;
        Date settlement = cal.advance(today,settlementDays,Days);

        dates.push_back(settlement); dates.push_back(settlement+1*Days);
        dates.push_back(settlement+1*Weeks);dates.push_back(settlement+1*Months);
        dates.push_back(settlement+2*Months);dates.push_back(settlement+3*Months);
        dates.push_back(settlement+6*Months);dates.push_back(settlement+9*Months);
        dates.push_back(settlement+1*Years);dates.push_back(settlement+1*Years+3*Months);
        dates.push_back(settlement+1*Years+6*Months); dates.push_back(settlement+1*Years+9*Months);
        dates.push_back(settlement+2*Years);

        dfs.push_back(1.0);     dfs.push_back(0.9999656);
        dfs.push_back(0.9999072); dfs.push_back(0.9996074);
        dfs.push_back(0.9990040); dfs.push_back(0.9981237);
        dfs.push_back(0.9951358); dfs.push_back(0.9929456);
        dfs.push_back(0.9899849); dfs.push_back(0.9861596); //
        dfs.push_back(0.9815178); dfs.push_back(0.9752363);
        dfs.push_back(0.9680804);

        Date tmpDate1=settlement+1*Years+3*Months;
        InterpolatedDiscountCurve<LogLinear> curve(dates,dfs,dc,cal);
        std::cout << "Zero Rate: " << curve.zeroRate(tmpDate1, dc, Simple, Annual) << std::endl;
        std::cout << "Discount: " << curve.discount(tmpDate1) << std::endl;
        Date tmpDate2=tmpDate1+3*Months;
        std::cout << "1Y3M-1Y6M Fwd Rate: " << curve.forwardRate(tmpDate1,tmpDate2,dc,Continuous) << std::endl;
}
```

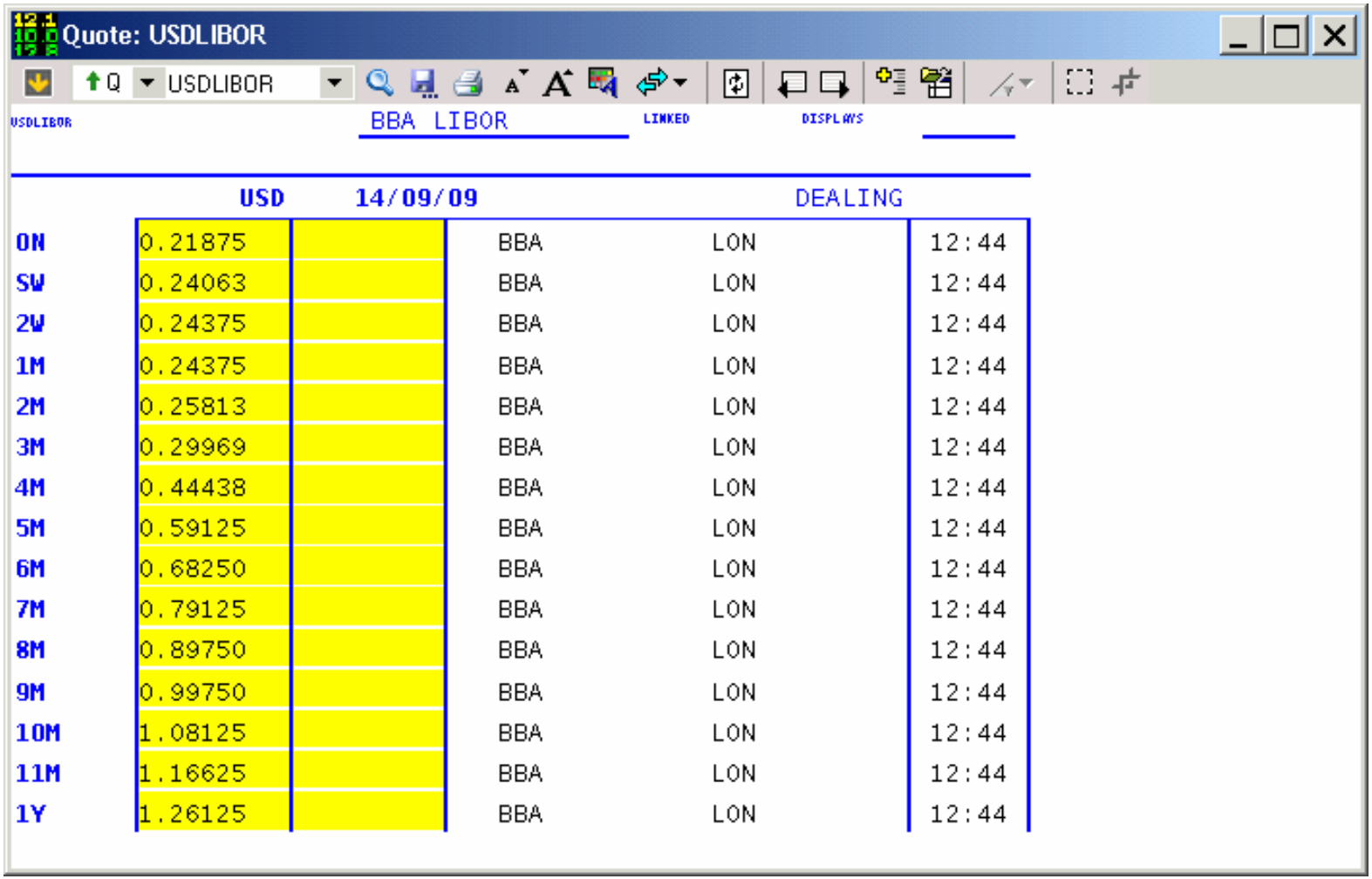
After the construction of the yield curve, we ask for a zero-rate, a discount factor and a forward rate for given dates. The output after calling this function is shown below

```
Zero Rate:  1.107998 % Actual/360 simple compounding
Discount:  0.98616
1Y3M-1Y6M Fwd Rate:  1.887223 % Actual/360 continuous compounding
```

This section will focus on the piecewise construction of the yield curve. This means, that we will use a particular class of market instruments to construct a piece of a yield curve. For example, deposit rates can be used for maturities up to 1 year. Forward Rate Agreements can be used for the construction of the yield curve between 1 and 2 years. Swap rates can be used for the residual maturities. The final result is a single curve which is consistent with all quotes.

The choice of the instruments and maturities depends on the liquidity. In the following example, we will construct a USD yield curve with USD Libor rates for maturities up to -and including- 1 year. For maturities above 1 year and up to -and excluding- 2 years we will use 3 month FRA rates. For larger maturities, we will use ISDA Fix swap rates, the specification of these instruments can be found on http://www.isda.org/fix/isdafix.html. The corresponding Reuters screens which we will use for our construction are shown next.

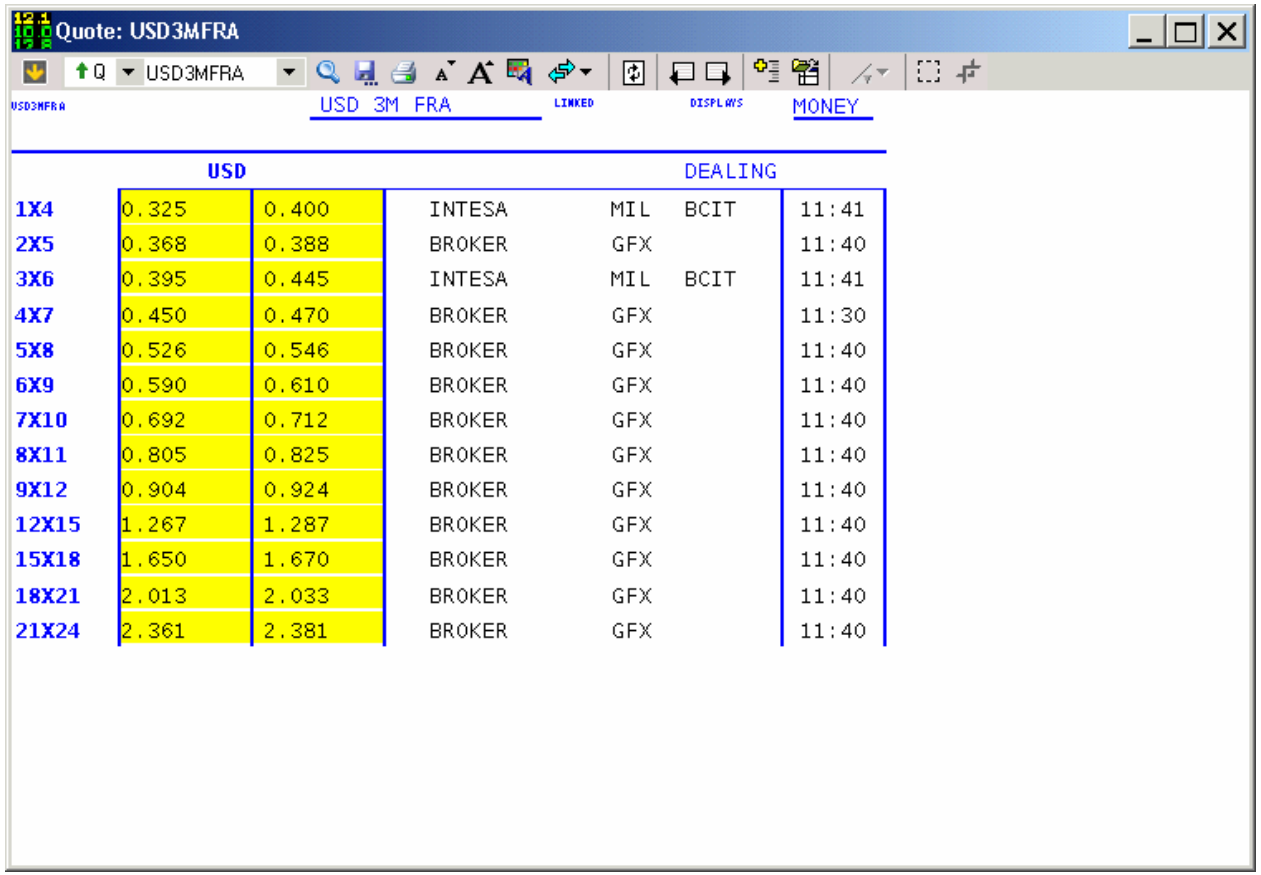
Deposit rates on Reuters



	USD	14/09/09		DEALING	
ON	0.21875		BBA	LON	12:44
SW	0.24063		BBA	LON	12:44
2W	0.24375		BBA	LON	12:44
1M	0.24375		BBA	LON	12:44
2M	0.25813		BBA	LON	12:44
3M	0.29969		BBA	LON	12:44
4M	0.44438		BBA	LON	12:44
5M	0.59125		BBA	LON	12:44
6M	0.68250		BBA	LON	12:44
7M	0.79125		BBA	LON	12:44
8M	0.89750		BBA	LON	12:44
9M	0.99750		BBA	LON	12:44
10M	1.08125		BBA	LON	12:44
11M	1.16625		BBA	LON	12:44
1Y	1.26125		BBA	LON	12:44

Figure: USD Libor Rates

FRA rates on Reuters



	USD					DEALING
1X4	0.325	0.400	INTESA	MIL	BCIT	11:41
2X5	0.368	0.388	BROKER	GFX		11:40
3X6	0.395	0.445	INTESA	MIL	BCIT	11:41
4X7	0.450	0.470	BROKER	GFX		11:30
5X8	0.526	0.546	BROKER	GFX		11:40
6X9	0.590	0.610	BROKER	GFX		11:40
7X10	0.692	0.712	BROKER	GFX		11:40
8X11	0.805	0.825	BROKER	GFX		11:40
9X12	0.904	0.924	BROKER	GFX		11:40
12X15	1.267	1.287	BROKER	GFX		11:40
15X18	1.650	1.670	BROKER	GFX		11:40
18X21	2.013	2.033	BROKER	GFX		11:40
21X24	2.361	2.381	BROKER	GFX		11:40

Figure: USD FRA Rates

Swap rates on Reuters

Quote: USDSFIX=

USDSFIX= | USD ISDA FIX | LINKED | DISPLAYS | ISDA01

RATES COLLECTED BY REUTERS AND GARBAN ICAP PLC

	USD	SA30/360 3MLIBOR	DEALING
1Y	0.572		17:30
2Y	1.226		17:30
3Y	1.846		17:30
4Y	2.324		17:30
5Y	2.678		17:30
6Y	2.958		17:30
7Y	3.180		17:30
8Y	3.340		17:30
9Y	3.474		17:30
10Y	3.588		17:30
15Y	3.916		17:30
20Y	4.029		17:30
30Y	4.109		17:30

Figure: USD Swap Rates

QuantLib allows to incorporate the piecewise construction by using rate helpers which are summarized in

```
<ql/termstructures/yield/ratehelpers.hpp>
```

The piecewise yield curve constructor will accept a

```
std:vector<boost::shared_ptr<RateHelper>>
```

object which stores all market instruments. The following rate helpers are available

- `FuturesRateHelper`
- `DepositRateHelper`
- `FraRateHelper`
- `SwapRateHelper`
- `BMASwapRateHelper` (Bond Market Association Swaps)

Different constructors are available for each rate helper, where all relevant properties can be specified. We will not introduce all of them here, but choose the compact constructors which use the `Index` classes introduced before. Instead of specifying the deposit properties by hand (e.g. business day convention, day counter...) we will specify the index where all properties are implemented automatically.

For example, one constructor for the `DepositRateHelper` is available as

```
DepositRateHelper(const Handle<Quote>& rate,
const boost::shared_ptr<IborIndex>& iborIndex);
```

To set up a rate helper we need to set up a Handle for the rate and a corresponding index. The following example shows such a setup for the USD curve case for all three rate types. The number of quotes will be restricted to 3, the other quote setups are equivalent. There is a lot of code needed for the setup of all rates. However, this will most likely be implemented once in a large system.

The other are implemented equivalently and we assume for the following example that the rate helper vector is returned by a function

```
std::vector<boost::shared_ptr<RateHelper>> getRateHelperVector()
```

The first constructor for the piecewise yield curve is given below

```
PiecewiseYieldCurve(
        const Date& referenceDate,
        const std::vector<boost::shared_ptr<typename Traits::helper> >& instruments,
        const DayCounter& dayCounter,
        const std::vector<Handle<Quote> >& jumps = std::vector<Handle<Quote> >(),
        const std::vector<Date>& jumpDates = std::vector<Date>(),
        Real accuracy = 1.0e-12,
        const Interpolator& i = Interpolator(),
        const Bootstrap<this_curve>& bootstrap = Bootstrap<this_curve>())
```

The constructor has 3 template types called `Traits`, `Interpolator`, `Bootstrap = IterativeBootstrap`. The third template specifies the boot strapping procedure with a default boot strapper. The first parameter specifies the rate type which should be created by the curve. The second parameter specifies the interpolation procedure on this rate type.

A second constructor is available as

```
PiecewiseYieldCurve(
            Natural settlementDays,
            const Calendar& calendar,
            const std::vector<boost::shared_ptr<typename Traits::helper> >& instruments,
            const DayCounter& dayCounter,
            const std::vector<Handle<Quote> >& jumps = std::vector<Handle<Quote> >(),
            const std::vector<Date>& jumpDates = std::vector<Date>(),
            Real accuracy = 1.0e-12,
            const Interpolator& i = Interpolator(),
            const Bootstrap<this_curve>& bootstrap = Bootstrap<this_curve>())
```

This parameter doesn't have a fixed date, since the reference date is internally given by `Settings::instance.valuationDate()`. The number of days to settlement has to be provided with `settlementDays`. This constructor observers the valuation date, such that it updates whenever the date changes. The constructor will be used with the first 4 parameters most of the time, so don't worry about the other parameters for the moment. The next example shows the usage of the first constructor.

```
#include "YieldCurve6.h"

void testingYields4(){

        std::vector<boost::shared_ptr<RateHelper>> instruments=getRateHelperVector();

        Calendar calendar = TARGET();
        Date today(11,Sep,2009);
        Natural settlementDays = 2;
        Date settlement = calendar.advance(today,settlementDays,Days);

        std::cout << "Settlement Date:" << settlement << std::endl;

        DayCounter dc=Actual360();
        boost::shared_ptr<YieldTermStructure> yieldCurve;

        yieldCurve = boost::shared_ptr<YieldTermStructure>(new
                                PiecewiseYieldCurve<ZeroYield,Linear>(settlement,instruments, dc));

        Date d1=settlement+1*Years,d2=d1+3*Months;

        std::cout << "Zero 3M: "        << yieldCurve->zeroRate(settlement+3*Months,dc,Simple) << std::endl;
        std::cout << "Zero 6M: "        << yieldCurve->zeroRate(settlement+6*Months,dc,Simple) << std::endl;
        std::cout << "Zero 9M: "        << yieldCurve->zeroRate(settlement+9*Months,dc,Simple) << std::endl;
        std::cout << "12x15 Fwd: "      << yieldCurve->forwardRate(d1,d2,dc,Simple)<< std::endl;
        std::cout << "15x18 Fwd: "      << yieldCurve->forwardRate(d2,d2+3*Months,dc,Simple)<< std::endl;
        // Check swap rate
        Handle<YieldTermStructure> ycHandle(yieldCurve);
        // ISDA swap is vs 3 month libor, set this up
        boost::shared_ptr<IborIndex> libor3m(new USDLibor(Period(3,Months),ycHandle));
        // set up a 8y ISDA swap, just to have references to all properties
        boost::shared_ptr<SwapIndex> swap8yIndex(new UsdLiborSwapIsdaFixAm(Period(8,Years)));
        // construct a vanilla swap
        VanillaSwap swap =    MakeVanillaSwap(Period(8,Years),libor3m)
                                        .withEffectiveDate(settlement)
                                        .withFixedLegConvention(swap8yIndex->fixedLegConvention())
                                        .withFixedLegTenor(swap8yIndex->fixedLegTenor());

        std::cout << "8Y Swap:" << swap.fairRate() << std::endl;
}
```

The instance was constructed as a `ZeroRate` curve which is linearly interpolated. To test whether the construction was successful, we print some zero and forward rates. Also, we set up a vanilla swap with the built curve to test whether the swap rate can be recovered. Don't worry about the vanilla swap details for the moment. They will be introduced later. The output of the function is

```
Settlement Date:September 15th, 2009
Zero 3M: 0.299690 % Actual/360 simple compounding
Zero 6M: 0.682500 % Actual/360 simple compounding
Zero 9M: 0.997500 % Actual/360 simple compounding
12x15 Fwd:  1.267000 % Actual/360 simple compounding
15x18 Fwd:  1.650000 % Actual/360 simple compounding
8Y Swap:0.0334
```

It can be verified that the Reuters rates are recovered correctly.

The `InterpolatedSmileSection` class represents a single volatility slice which is mapped to a fixed time to maturity. This is a template class with respect to the interpolator. However, the current version is suboptimal since the template class has to be a factory of a `Interpolator` object which implements the `interpolate` functions. The class accepts a grid of total volatilities $\sigma(K)\sqrt{\tau}$ and strikes and interpolates values which are not part of the grid. Below we show two constructors which use quote handles. The class has equivalent versions with simple `Real` objects

```
InterpolatedSmileSection(
    Time expiryTime,
    const std::vector<Rate>& strikes,
    const std::vector<Handle<Quote> >& stdDevHandles,
    const Handle<Quote>& atmLevel,
    const Interpolator& interpolator = Interpolator(),
    const DayCounter& dc = Actual365Fixed());

InterpolatedSmileSection(
    const Date& d,
    const std::vector<Rate>& strikes,
    const std::vector<Handle<Quote> >& stdDevHandles,
    const Handle<Quote>& atmLevel,
    const DayCounter& dc = Actual365Fixed(),
    const Interpolator& interpolator = Interpolator(),
    const Date& referenceDate = Date());
```

The first constructor accepts a time to maturity. This encapsulates the calculation such that the user can provide a variable calculated with his own algorithm. The second constructor accepts a date and a reference date, which is an optional variable. If the reference date is not given, the global evaluation date in `Settings` will be used. The time to maturity is then calculated with the provided day counter. We note again that the constructors are based on $\sigma\sqrt{\tau}$ and not simply σ. The following functions are provided by the `InterpolatedSmileSection`

- `Real variance(Rate strike)` returns $\sigma^2(K)\tau$
- `Volatility volatility(Rate strike)` returns $\sigma(K)$
- `Real minStrike ()` returns the minimum strike in the strike grid
- `Real maxStrike()` returns the maximum strike
- `Real atmLevel()` returns the at-the-money value $\sigma_{ATM}\sqrt{tau}$

In the following example we will construct a smile section using cubic interpolation. We will then check if the volatility is returned correctly at one of the strike grid points. Then, the volatility at some different strike is calculated.

```
void testingVolatilityObjects1(){

        Time tau=0.45,st=std::sqrt(tau);

        std::vector<Rate> strikes(6);
        strikes[0]=78.0;        strikes[1]=88.0;        strikes[2]=98.0;
        strikes[3]=108.0;       strikes[4]=118.0;       strikes[5]=128.0;

        boost::shared_ptr<Quote> q1(new SimpleQuote(0.2406*st)); Handle<Quote> h1(q1);
        boost::shared_ptr<Quote> q2(new SimpleQuote(0.2213*st)); Handle<Quote> h2(q2);
        boost::shared_ptr<Quote> q3(new SimpleQuote(0.2102*st)); Handle<Quote> h3(q3);
        boost::shared_ptr<Quote> q4(new SimpleQuote(0.2156*st)); Handle<Quote> h4(q4);
        boost::shared_ptr<Quote> q5(new SimpleQuote(0.2299*st)); Handle<Quote> h5(q5);
        boost::shared_ptr<Quote> q6(new SimpleQuote(0.2501*st)); Handle<Quote> h6(q6);

        std::vector<Handle<Quote>> volVec(strikes.size());
        volVec[0]=h1; volVec[1]=h2; volVec[2]=h3; volVec[3]=h4; volVec[4]=h5;   volVec[5]=h6;

        InterpolatedSmileSection<Cubic> smileSect(tau,strikes,volVec,h3);

        Real K1=88.0, K2=93.0;

        std::cout << "Min strike:" << smileSect.minStrike() << std::endl;
        std::cout << "Max strike:" << smileSect.maxStrike() << std::endl;
        std::cout << "Atm vol:" << smileSect.atmLevel() << std::endl;
        std::cout << "Volatility at K1:" << smileSect.volatility(K1) << std::endl;
        std::cout << "Variance at K1:" << smileSect.variance(K1) << std::endl;
        std::cout << "Volatility at K2:" << smileSect.volatility(K2) << std::endl;
        std::cout << "Variance at K2:" << smileSect.variance(K2) << std::endl;
}
```

The output of the function is

```
Min strike:78
Max strike:128
Atm vol:0.141006
Volatility at K1:0.2213
Variance at K1:0.0220382
Volatility at K2:0.213988
Variance at K2:0.0206059
```

As expected, the correct volatility is returned for the strike 88.0.

Modelling the whole volatility surface can be done with the `BlackVarianceSurface` class which has the constructor below. This is not a template class, but the type of the 2D interpolation can be set with the `setInterpolation(const Interpolator& i)` template function. The constructor needs a volatility matrix. Also, the type of the strike extrapolation can be chosen depending on the extrapolation to the left of the minimum strike (lower extrapolation) or to the right of the maximum strike (upper extrapolation). The extrapolation types can be `ConstantExtrapolation` or the default extrapolation of the interpolator `InterpolatorDefaultExtrapolation`. The default interpolation is `BiLinear`.

```
BlackVarianceSurface(const Date& referenceDate,
                     const Calendar& cal,
                     const std::vector<Date>& dates,
                     const std::vector<Real>& strikes,
                     const Matrix& blackVolMatrix,
                     const DayCounter& dayCounter,
                     Extrapolation lowerExtrapolation =
                        InterpolatorDefaultExtrapolation,
                     Extrapolation upperExtrapolation =
                        InterpolatorDefaultExtrapolation);
```

Important class functions are

- `Real minStrike()`
- `Real maxStrike()`
- `Date maxDate()`
- `void setInterpolation(const Interpolator& i)`
- `Volatility blackVol(const Date& maturity, Real strike, bool extrapolate)` also available with a `Time` variable
- `Real blackVariance(const Date& maturity, Real strike, bool extrapolate)`, also available with a `Time` variable

The next example shows a volatility surface construction with an example of how to change the interpolation method.

```
#include <boost/assign/std/vector.hpp>

void testingVolatilityObjects2(){

        DayCounter dc=ActualActual();
        Calendar eurexCal=Germany(Germany::Eurex);

        using namespace boost::assign;

        Date settlementDate(27, Sep, 2009);
        settlementDate=eurexCal.adjust(settlementDate);

        std::vector<Date> dateVec;

        std::vector<Size> days;
        days+=13,41,75,165,256,345,524,703;

        for(Size i=0;i<days.size();++i){
                dateVec.push_back(settlementDate+days[i]*Days);
        }

        std::vector<Real> strikes;
        strikes+=100,500,2000,3400,3600,3800,4000,4200,4400,4500,
                        4600,4800,5000,5200,5400,5600,7500,10000,20000,30000;

        std::vector<Volatility> v;
    v+=1.015873, 1.015873, 1.015873, 0.89729, 0.796493, 0.730914, 0.631335, 0.568895,
        0.711309, 0.711309, 0.711309, 0.641309, 0.635593, 0.583653, 0.508045, 0.463182,
        0.516034, 0.500534, 0.500534, 0.500534, 0.448706, 0.416661, 0.375470, 0.353442,
        0.516034, 0.482263, 0.447713, 0.387703, 0.355064, 0.337438, 0.316966, 0.306859,
        0.497587, 0.464373, 0.430764, 0.374052, 0.344336, 0.328607, 0.310619, 0.301865,
        0.479511, 0.446815, 0.414194, 0.361010, 0.334204, 0.320301, 0.304664, 0.297180,
        0.461866, 0.429645, 0.398092, 0.348638, 0.324680, 0.312512, 0.299082, 0.292785,
        0.444801, 0.413014, 0.382634, 0.337026, 0.315788, 0.305239, 0.293855, 0.288660,
        0.428604, 0.397219, 0.368109, 0.326282, 0.307555, 0.298483, 0.288972, 0.284791,
        0.420971, 0.389782, 0.361317, 0.321274, 0.303697, 0.295302, 0.286655, 0.282948,
        0.413749, 0.382754, 0.354917, 0.316532, 0.300016, 0.292251, 0.284420, 0.281164,
        0.400889, 0.370272, 0.343525, 0.307904, 0.293204, 0.286549, 0.280189, 0.277767,
        0.390685, 0.360399, 0.334344, 0.300507, 0.287149, 0.281380, 0.276271, 0.274588,
        0.383477, 0.353434, 0.327580, 0.294408, 0.281867, 0.276746, 0.272655, 0.271617,
        0.379106, 0.349214, 0.323160, 0.289618, 0.277362, 0.272641, 0.269332, 0.268846,
        0.377073, 0.347258, 0.320776, 0.286077, 0.273617, 0.269057, 0.266293, 0.266265,
        0.399925, 0.369232, 0.338895, 0.289042, 0.265509, 0.255589, 0.249308, 0.249665,
```

The output of the function is

```
Bilinear Interpolation:0.376773
Bicubic Interpolation:0.380937
```

1. Mathematical Tools
   - Integration
   - Solver
   - Exercise
   - Interpolation
   - Matrix
   - Optimizer
   - Exercise
   - Random Numbers
   - Copulas
2. Fixed Income
   - Indexes
   - Interest Rate
   - Yield Curve Construction
3. Volatility Objects
   - Smile Sections
4. Payoffs and Exercises
5. Black Scholes Pricer
   - Black Scholes Calculator
6. Stochastic Processes
   - Generalized Black Scholes Process
   - Ornstein Uhlenbeck Process
   - Heston Process
   - Bates Process

Every pricing engines needs a payoff which underlies the derivative instrument. The payoffs are derived from the abstract `Payoff` class which is located in `<ql/payoff.hpp>`. The class has a `name` and `description` function which return the respective properties as a `std::string` object. The payoff is returned by passing the price to the operator

- `Real operator()(Real price)`

The derived payoff classes can be found in `<ql/instruments/payoffs.hpp>`. Most of the payoffs have a given strike. Consequently, a base class `StrikedTypePayoff` is provided. Descendants of this class are

- `PlainVanillaPayoff`: represents the payoff of a plain vanilla call or put

$$\max\{\phi(S_T - K), 0\}$$

  with $\phi = 1$ for a call and $\phi = -1$ for a put.

- `PercentageStrikePayoff`

$$\max\{\phi(S_T - mS_T), 0\}$$

  with m being a percentage or moneyness variable such as 1.10. This can be useful for Cliquet payoffs where the future asset value is taken as the strike reference.

- `AssetOrNothingPayoff`

$$S_T \mathbb{I}_{\phi S_T > \phi K}$$

- `CashOrNothingPayoff`

$$C \mathbb{I}_{\phi S_T > \phi K}$$

  with C being the cash amount.

- `GapPayoff` pays

$$\max\{\phi(S_T - K_2), 0\}\mathbb{I}_{\phi S_T \geq \phi K_1}$$

The constructors of the payoff classes are straightforward. For example, the plain vanilla payoff can be constructed as

- `PlainVanillaPayoff(Option::Type type,Real strike)`

In the example program below, we will construct the payoffs for various option types and print out the payoffs for a concrete parameter set.

```
void testingBsPricingEngines1(){
        Real S0=100.0, ST=123;
        Real K1=105.0, K2=112.0;
        Real moneyness=1.10, cash=10.0;

        PlainVanillaPayoff vanillaPayoffCall(Option::Call,K1);
        PlainVanillaPayoff vanillaPayoffPut(Option::Put,K1);
        //
        PercentageStrikePayoff percentagePayoffCall(Option::Call,moneyness);
        PercentageStrikePayoff percentagePayoffPut(Option::Put,moneyness);
        //
        AssetOrNothingPayoff aonPayoffCall(Option::Call,K1);
        AssetOrNothingPayoff aonPayoffPut(Option::Put,K1);
        //
        CashOrNothingPayoff conPayoffCall(Option::Call,K1,cash);
        CashOrNothingPayoff conPayoffPut(Option::Put,K1,cash);
        //
        GapPayoff gapPayoffCall(Option::Call,K1,K2);
        GapPayoff gapPayoffPut(Option::Put,K1,K2);

        std::cout << "S(0):" << S0<< ", S(T):" << ST << std::endl;
        std::cout << "Strike:" << K1 << std::endl;
        std::cout << "Gap Strike:" << K2 << ", Moneyness Strike:"<<moneyness*S0 << std::endl;
        std::cout << "Vanilla Call Payout:" << vanillaPayoffCall(ST) << std::endl;
        std::cout << "Vanilla Put Payout:" << vanillaPayoffPut(ST) << std::endl;
        std::cout << "Percentage Call Payout:" << percentagePayoffCall(ST) << std::endl;
        std::cout << "Percentage Put Payout:" << percentagePayoffPut(ST) << std::endl;
        std::cout << "AON Call Payout:" << aonPayoffCall(ST) << std::endl;
        std::cout << "AON Put Payout:" << aonPayoffPut(ST) << std::endl;
        std::cout << "CON Call Payout:" << conPayoffCall(ST) << std::endl;
        std::cout << "CON Put Payout:" << conPayoffPut(ST) << std::endl;
        std::cout << "Gap Call Payout:" << gapPayoffCall(ST) << std::endl;
        std::cout << "Gap Put Payout:" << gapPayoffPut(ST) << std::endl;
}
```

The output of the function is

```
S(0):100, S(T):123
Strike:105
Gap Strike:112
Vanilla Call Payout:18
Vanilla Put Payout:0
Percentage Call Payout:0
Percentage Put Payout:12.3
AON Call Payout:123
AON Put Payout:0
CON Call Payout:10
CON Put Payout:0
Gap Call Payout:11
Gap Put Payout:0
```

Another instrument property is its exercise. For example, a vanilla call might be exercised at maturity, on a set of discrete dates (Bermudan exercise) or any time (American exercise). This is modelled by the `Exercise` class which can be found in

`<ql/exercise.hpp>`

Concrete exercise types are derived from the `Exercise` base class. The currently available classes are

- `AmericanExercise`
- `BermudanExercise`
- `EuropeanExercise`

Example constructors for some classes are given below

- `EuropeanExercise(const Date& date)` where the date of the exercise has to be provided
- `AmericanExercise(const Date& earliestDate, const Date& latestDate, bool payoffAtExpiry)`: here, a boolean variable can be set which indicates if the payout is done immediately or at maturity
- `BermudanExercise(const std::vector<Date>& dates, bool payoffAtExpiry = false)`: here a vector of exercise dates has to be provided

The heart of any pricing engine is a class which returns the standard Black-Scholes formula as well as the corresponding Greeks. The corresponding class in QuantLib is `BlackScholesCalculator` which can be found in

`<ql/pricingengines/blackcalculator.hpp>`

The corresponding constructor is

- `BlackScholesCalculator( const boost::shared_ptr<StrikedTypePayoff>& payoff, Real spot, DiscountFactor growth, Real stdDev, DiscountFactor discount);`

with

- `growth` being $e^{-r_f \tau}$ where r_f is the foreign rate or dividend yield
- `discount` being $e^{-r_d \tau}$ being the domestic rate (the usual interest rate)
- `stdDev` being $\sigma\sqrt{\tau}$
- `payoff` being a pointer to a striked payoff, e.g. a vanilla or asset or nothing payoff.

The class avoids any specification of the time to maturity. This is an encapsulation of time to maturity considerations where otherwise day-counters and calendars would have to be taken into account. The discount factors can be taken from the corresponding yield/dividend term structure.

The parameters can be returned by calling

- `value()` v
- `delta()` $\frac{\partial v}{\partial S}$
- `elasticity()` $\frac{\partial v}{\partial S}\frac{S}{v}$
- `gamma()` $\frac{\partial^2 v}{\partial S^2}$
- `vega(Time maturity)` $\frac{\partial v}{\partial \sigma}$
- `rho(Time maturity)` $\frac{\partial v}{\partial r_d}$
- `dividendRho(Time maturity)` $\frac{\partial v}{\partial r_f}$
- `theta(Time maturity)` $\frac{\partial v}{\partial \tau}$
- `deltaForward()` $\frac{\partial v}{\partial f}$
- `gammaForward()` $\frac{\partial^2 v}{\partial f^2}$
- `itmCashProbability()` $P(S_T > K)$ with P being the bond martingale measure
- `itmAssetProbability()` $P(S_T > K)$ with P being the asset martingale measure

The following example shows how the values above can be obtained for a vanilla put and asset or nothing call.

```
void testingBlackScholesCalculator(){

        Real S0=100.0, K=105.0;
        Real rd=0.034, rf=0.021, tau=0.5, vol=0.177;
        Real domDisc=std::exp(-rd*tau), forDisc=std::exp(-rf*tau);
        Real stdDev=vol*std::sqrt(tau);

        boost::shared_ptr<PlainVanillaPayoff> vanillaPayoffPut(
                new PlainVanillaPayoff(Option::Put,K));

        boost::shared_ptr<AssetOrNothingPayoff> aonPayoffCall(
                new AssetOrNothingPayoff(Option::Call,K));

    BlackScholesCalculator vanillaPutPricer(vanillaPayoffPut,S0,forDisc,stdDev,domDisc);
    BlackScholesCalculator aonCallPricer(aonPayoffCall,S0,forDisc,stdDev,domDisc);

         std::cout << "--------------Vanilla Values-------------" << std::endl;
         std::cout << "Value:" << vanillaPutPricer.value() << std::endl;
         std::cout << "Delta:" << vanillaPutPricer.delta() << std::endl;
         std::cout << "Gamma:" << vanillaPutPricer.gamma() << std::endl;
         std::cout << "Vega:" << vanillaPutPricer.vega(tau) << std::endl;
         std::cout << "Theta:" << vanillaPutPricer.theta(tau) << std::endl;
         std::cout << "Delta Fwd:" << vanillaPutPricer.deltaForward() << std::endl;
         std::cout << "Gamma Fwd:" << vanillaPutPricer.gammaForward() << std::endl;
         std::cout << "-------------- AON Values-------------" << std::endl;
         std::cout << "Value:" << aonCallPricer.value() << std::endl;
         std::cout << "Delta:" << aonCallPricer.delta() << std::endl;
         std::cout << "Gamma:" << aonCallPricer.gamma() << std::endl;
         std::cout << "Vega:" << aonCallPricer.vega(tau) << std::endl;
         std::cout << "Theta:" << aonCallPricer.theta(tau) << std::endl;
         std::cout << "Delta Fwd:" << aonCallPricer.deltaForward() << std::endl;
         std::cout << "Gamma Fwd:" << aonCallPricer.gammaForward() << std::endl;

}
```

```
--------------Vanilla Values-------------
Value:7.46648
Delta:-0.602108
Gamma:0.0303691
Vega:26.8767
Theta:-3.72057
Delta Fwd:-0.598207
Gamma Fwd:0.0299769
-------------- AON Values-------------
Value:38.7447
Delta:3.42436
Gamma:0.0971735
Vega:85.9985
Theta:-18.3561
Delta Fwd:3.40217
Gamma Fwd:0.0959184
```

This section introduces the general class of stochastic processes which are modeled in QuantLib. The basic idea of classes which depend on the stochastic process classes is to initialize the process first and pass it to some other class. This class extracts the required variables which are needed from the process. An example is a plain vanilla Black-Scholes option pricer which retrieves the volatility from the process. Another example is a path generator for Monte-Carlo pricing which requires the process parameters.

The base class for stochastic processes `StochasticProcess` can be found in

`<ql/stochasticprocess.hpp>`

The class is an observer and observable and models a general d-dimensional Ito process of the form

$$d\mathrm{S}_t = \mu(t, S_t)\mathrm{d}t + \sigma(t, \mathrm{S}_t)d\mathrm{W}_t.$$

The `StochasticProcess` has a public class called `discretization` which models the discretized version of the process. The discretization is kept general. The class has the following public member functions which are virtual.

- `Disposable<Array> drift(const StochasticProcess&, Time t0, const Array& x0, Time dt)`: returns the drift of the process from time t_0 with $S_{t_0} = x_0$ to time $t_0 + \Delta t$.
- `Disposable<Matrix> diffusion( const StochasticProcess&, Time t0, const Array& x0, Time dt)`: returns the diffusion matrix
- `Disposable<Matrix> covariance( const StochasticProcess&, Time t0, const Array& x0, Time dt)`: returns the covariance matrix

The `StochasticProcess` class provides the following `virtual` functions

- `Size size()`: the dimension of the process
- `Disposable<Array> initialValues()`: returns S_0
- `Disposable<Array> drift(Time t, const Array& x)`: returns $\mu(t, \mathrm{S}_t)$
- `Disposable<Matrix> diffusion(Time t, const Array& x)`: returns $\sigma(t, \mathrm{S}_t)$
- `Disposable<Array> expectation(Time t0, const Array& x0, Time dt)`: returns $E(S_{t_0} + \Delta_t | S_{t_0} = x_0)$ according to a chosen discretization
- `Disposable<Matrix> stdDeviation(Time t0, const Array& x0, Time dt)` the same as the expectation but with standard deviation
- `Disposable<Matrix> covariance(Time t0, const Array& x0, Time dt)` the same as the expecation but with covariance

- `Disposable<Array> evolve(Time t0, const Array& x0, Time dt, const Array& dw)` yields $S_{t_0+\Delta_t}$ given S_{t_0} and the vector of Brownian increments ΔW. This is the crucial function for path generators. Returns by default

$$E(\mathrm{S}_{t_0+\Delta_t}|S_{t_0}) + \sigma(\mathrm{S}_{t_0+\Delta_t}|S_{t_0})\Delta\mathrm{W}$$

  with σ being the standard deviation.
- `Disposable<Array> apply(const Array& x0, const Array& dx)` returns $S_0 + dS$

The same header provides a `StochasticProcess1D` function which derives from the general stochastic process. The class provides all of the functions derived from `StochasticProcess` where `Real` objects instead of `Array` and `Matrix` objects are used.

A concrete discretization of a process is provided in the `EulerDiscretization` class which derives from `StochasticProcess::discretization` and consequently implements the corresponding interface. We will show the implementation of the `drift` function::

- ```
  EulerDiscretization::drift( const StochasticProcess& process, Time t0,
  const Array& x0, Time dt) const {
  return process.drift(t0, x0)*dt;
  }
  ```

which is the well known implementation of the drift in the discretization

$$S(t + \Delta_t) = \mu(t, S_t)\Delta_t + \sigma(t, S_t)\Delta W_t$$
  ```

An alternative discretization is provided in the `EndEulerDiscretization` class which also derives from `StochasticProcess::discretization` and consequently implements the corresponding interface. In opposite to the simple `EulerDiscretization` this class evaluates the drift or diffusion function at the end of the discretized time interval $[t, t + \Delta_t]$. The interface is the same one as in the simple Euler case, but the discretisation returns

$$\mu(t + \Delta_t, S_t) \text{ instead of } \mu(t, S_t)$$

and

$$\sigma(t + \Delta_t, S_t) \text{ instead of } \sigma(t, S_t)$$

The `GeneralizedBlackScholesProcess` interface can be found in

`<ql/processes/blackscholesprocess.hpp>`

The class models a 1 dimensional stochastic process with the following SDE

$$dS_t = (r(t) - q(t) - \frac{\sigma(t, S_t)^2}{2})dt + \sigma dW_t$$

Consequently, a risk-neutral drift is used instead of the general drift μ. The risk neutral rate is adjusted by a dividend yield $q(t)$ and the corresponding diffusion term σ. The constructor is given as

```
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>(new EulerDiscretization));
```

The provided discretization will be be used in the `evolve` function to progress from time t to $t + \Delta_t$. The same header has some derived functions which do not add additional functionalities. The classes represent well known concrete processes such as

- `BlackScholesProcess`: the generalized process without dividend
- `BlackScholesMertonProcess`: the generalized process
- `GarmanKohlagenProcess`: the generalized process with domestic and foreign rate notation

In the following example, we will set up a Black-Scholes-Merton process with a flat risk free rate, dividend yield and volatility term structure. We will then print the drift and diffusion and simulate the process at discrete times. The Euler scheme is used for the simulation. The output of the function is

```
Risk neutral drift:  -0.00368368
Diffusion:  0.2144
----------------------
Time:  0.1, S_t:  47.9983
Time:  0.2, S_t:  48.4418
Time:  0.3, S_t:  53.8613
Time:  0.4, S_t:  53.7439
Time:  0.5, S_t:  50.8117
Time:  0.6, S_t:  51.5973
Time:  0.7, S_t:  52.2014
Time:  0.8, S_t:  56.2463
Time:  0.9, S_t:  65.8869
Time:  1, S_t:  62.3316
```

```
void testingStochasticProcesses1(){

        Date refDate=Date(27,Sep,2009);
        Rate riskFreeRate=0.0321;
        Rate dividendRate=0.0128;
        Real spot=52.0;
        Rate vol=0.2144;
        Calendar cal=TARGET();
        DayCounter dc=ActualActual();

        boost::shared_ptr<YieldTermStructure> rdStruct(new FlatForward(refDate,riskFreeRate,dc));
        boost::shared_ptr<YieldTermStructure> rqStruct(new FlatForward(refDate,dividendRate,dc));
        Handle<YieldTermStructure> rdHandle(rdStruct);
        Handle<YieldTermStructure> rqHandle(rqStruct);

        boost::shared_ptr<SimpleQuote> spotQuote(new SimpleQuote(spot));
        Handle<Quote> spotHandle(spotQuote);

        boost::shared_ptr<BlackVolTermStructure> volQuote(new BlackConstantVol(refDate, cal, vol, dc));
        Handle<BlackVolTermStructure> volHandle(volQuote);

        boost::shared_ptr<BlackScholesMertonProcess> bsmProcess(
                 new BlackScholesMertonProcess(spotHandle,rqHandle, rdHandle,volHandle));

        BigInteger seed=12324;
        MersenneTwisterUniformRng       unifMt(seed);
        BoxMullerGaussianRng<MersenneTwisterUniformRng> bmGauss(unifMt);

        Time dt=0.10,t=0.0;
        Real x=spotQuote->value();
        Real dw;
        Size numVals=10;

        std::cout << "Risk neutral drift: " << bsmProcess->drift(t+dt,x) << std::endl;
        std::cout << "Diffusion: " << bsmProcess->diffusion(t+dt,x) << std::endl;
        std::cout << "-----------------------" << std::endl;

        for(Size j=1;j<=numVals;++j){
                dw=bmGauss.next().value;
                x=bsmProcess->evolve(t,x,dt,dw);
                std::cout << "Time: " << t+dt << ", S_t: " <<  x << std::endl;
                t+=dt;
        }
```

The Ornstein-Uhlenbeck process evolves according to

$$dX_t = \kappa(\theta - X_t)dt + \sigma dW_t$$

The constructor is given as

- `OrnsteinUhlenbeckProcess(Real speed, Volatility vol, Real x0, Real level)`

The variable denoted as speed is κ, the level corresponds to θ, vol is σ and x0 represents X_0. The process can be simulated analytically as the distribution of the SDE integrals is known. Given the current value r_0 the process at time t is normally distributed with mean

$$\mu = e^{-\kappa t} r_0 + \theta(1 - e^{-\kappa t})$$

and variance

$$\sigma^2 = \sigma^2 \frac{1}{2\kappa}(1 - e^{-2\kappa t})$$

The next example will compare the analytical results to the one estimated by using results from the evolve function.

```
void testingStochasticProcesses2(){

        Real x0=0.0311;
        Real x;
        Real theta=0.015;
        Real kappa=0.5;
        Real vol=0.02;

        BigInteger seed=12324;
        MersenneTwisterUniformRng       unifMt(seed);
        BoxMullerGaussianRng<MersenneTwisterUniformRng> bmGauss(unifMt);

        boost::shared_ptr<OrnsteinUhlenbeckProcess> shortRateProces(
                new OrnsteinUhlenbeckProcess(kappa,vol,x0,theta));

        Time dt=0.5,t=0.0;
        Real dw;

        Real mean=0.0,var=0.0;
        Size numVals=10000;

        for(Size j=1;j<=numVals;++j){
                dw=bmGauss.next().value;

                x=shortRateProces->evolve(t,x0,dt,dw);
                mean+=x;
                var+=x*x;
        }

        Real analyticMean=std::exp(-kappa*dt)*x0+theta*(1-std::exp(-kappa*dt));
        Real analyticVar=vol*vol*(0.5/kappa)*(1-std::exp(-2*kappa*dt));
        Real estimatedMean=mean/numVals;
        Real estimatedVar=var/numVals-estimatedMean*estimatedMean;

        std::cout << "Analytical Mean:"          << analyticMean << std::endl;
        std::cout << "Estimated Mean:"           << estimatedMean << std::endl;
        std::cout << "Analytical Variance:" << analyticVar << std::endl;
        std::cout << "Estimated Variance:"       << estimatedVar << std::endl;

}
```

The output of the function is

```
Analytical Mean:0.0275387
Estimated Mean:0.0276135
Analytical Variance:0.000157388
Estimated Variance:0.000159985
```

which shows that the analytical and estimated values are relatively close.

A related process is the square root process described by the following SDE

$$dX_t = \kappa(\theta - X_t)dt + \sigma\sqrt{X_t}dW_t.$$

which is implemented in

`<ql/processes/squarerootprocess.hpp>`

The constructor is given as

- `SquareRootProcess( Real b, Real a, Volatility sigma, Real x0 = 0.0, const boost::shared_ptr<discretization>& d = boost::shared_ptr<discretization>(new EulerDiscretization))`

The famous Heston modell is based on the following stochastic differential equation

$$
\begin{aligned}
dS_t &= \mu S_t dt + \sqrt{V_t} S_t dW_t^S && (2)\\
dV_t &= \kappa(\theta - V_t)dt + \sigma\sqrt{V_t} dW_t^V \, , V_0 = v_0 && (3)\\
dW_t^S dW_t^V &= \rho dt && (4)
\end{aligned}
$$

This parameters are reflected in the constructor which is given as

```
HestonProcess(const Handle<YieldTermStructure>& riskFreeRate,
              const Handle<YieldTermStructure>& dividendYield,
              const Handle<Quote>& s0,
              Real v0, Real kappa,
              Real theta, Real sigma, Real rho,
              Discretization d = FullTruncation);
```

Since the equation system doesn't have a closed form solution, one has to think about the discretization which is used in the `evolve` function. Possible choices are given in the enumeration `Discretization`:

- `Discretization{PartialTruncation, FullTruncation, Reflection, ExactVariance };`

All methods are described in Lord et al. (2006). The `ExactVariance` method is described in detail in the work by Broadie and Kaya (2006). We will discuss the first three methods first. All methods assume that the volatility process is approximated by an Euler discretization

$$V_{t+\Delta_t} = V_t + \kappa(\theta - V_t)\Delta_t + \sigma\sqrt{V_t}\Delta W_t^V$$

with

$$\Delta W_t^V = W_{t+\Delta_t} - W_t$$

This can be reformulated to yield

$$V_{t+\Delta_t} = V_t + \kappa\theta\Delta_t - \kappa V_t \Delta_t + \sigma\sqrt{V_t}\Delta W_t^V$$

The next step is to think about what happens if the value $V_{t+\Delta_t}$ turns out to be a negative number. On the right hand side, the term V_t appears three times and there are different rules to deal with this case at the different occurrences. We will rewrite $V_{t+\Delta_t}$ as

$$V_{t+\Delta_t} = f_3(\tilde{V}_{t+\Delta_t})$$

with

$$\tilde{V}_{t+\Delta_t} = f_1(\tilde{V}_t) + \kappa\Delta_t(\theta - f_2(\tilde{V}_t)) + \sigma\sqrt{f_3(\tilde{V}_t)}\Delta W_t^V$$

The above defined versions correspond to

Table: Heston Discretizations

Scheme	f_1	f_2	f_3
Reflection	$\lvert x\rvert$	$\lvert x\rvert$	$\lvert x\rvert$
Partial Truncation	x	x	x^+
Full Truncation	x	x^+	x^+

The exact variance relies on the fact that the transition law of $V_{t+\Delta_t}$ is given as

$$\frac{\sigma^2(1-e^{-\kappa\Delta_t})}{4\kappa}\chi^2_d\Big(\frac{4\kappa e^{-\kappa\Delta_t}}{\sigma^2(1-e^{-\kappa\Delta_t})}V_t\Big)$$

with χ^2_d being the non central chi square distributions with

$$d=\frac{4\theta\kappa}{\sigma^2}$$

degrees of freedom. The next example shows a simulated asset and volatility proces.

```
void testingStochasticProcesses3(){

        Date refDate=Date(27,Sep,2009);
        Rate riskFreeRate=0.0321;
        Rate dividendRate=0.0128;
        Real spot=52.0;
        Rate vol=0.2144;
        Calendar cal=TARGET();
        DayCounter dc=ActualActual();

        boost::shared_ptr<YieldTermStructure> rdStruct(new FlatForward(refDate,riskFreeRate,dc));
        boost::shared_ptr<YieldTermStructure> rqStruct(new FlatForward(refDate,dividendRate,dc));
        Handle<YieldTermStructure> rdHandle(rdStruct);
        Handle<YieldTermStructure> rqHandle(rqStruct);

        boost::shared_ptr<SimpleQuote> spotQuote(new SimpleQuote(spot));
        Handle<Quote> spotHandle(spotQuote);

        boost::shared_ptr<BlackVolTermStructure> volQuote(new BlackConstantVol(refDate, cal, vol, dc));
        Handle<BlackVolTermStructure> volHandle(volQuote);

        Real v0=0.12, kappa=1.2, theta=0.08, sigma=0.05, rho=-0.6;
        boost::shared_ptr<HestonProcess> hestonProcess(new HestonProcess(rdHandle,rqHandle,spotHandle,v0,
                kappa,theta,sigma,rho,HestonProcess::PartialTruncation));


        BigInteger seed=12324;
        MersenneTwisterUniformRng       unifMt(seed);
        BoxMullerGaussianRng<MersenneTwisterUniformRng> bmGauss(unifMt);

        Time dt=0.10,t=0.0;
        Array dw(2),x(2);
        // x is the 2-dimensional process
        x[0]=spotQuote->value();
        x[1]=v0;
        Size numVals=10;

        for(Size j=1;j<=numVals;++j){
                dw[0]=bmGauss.next().value;
                dw[1]=bmGauss.next().value;

                x=hestonProcess->evolve(t,x,dt,dw);
                std::cout << "Time: " << t+dt << ", S_t: " <<  x[0] << ", V_t: " << x[1] << std::endl;
```

The output of the function is

```
Time:  0.1, S_t:  45.5306, V_t:  0.119682
Time:  0.2, S_t:  53.8413, V_t:  0.109652
Time:  0.3, S_t:  49.2252, V_t:  0.109647
Time:  0.4, S_t:  49.9689, V_t:  0.110166
Time:  0.5, S_t:  63.6438, V_t:  0.0957695
Time:  0.6, S_t:  61.4598, V_t:  0.0929384
Time:  0.7, S_t:  52.4621, V_t:  0.0954027
Time:  0.8, S_t:  45.6489, V_t:  0.0995833
Time:  0.9, S_t:  40.1511, V_t:  0.100911
Time:  1, S_t:  35.9219, V_t:  0.0990155
```

The Bates process is similar to the Heston process but adds jumps to the evolution of the asset price. The corresponding SDE can be stated as

$$\begin{aligned} dS_t &= (r - d - \lambda m)S_t dt + \sqrt{v} S_t dW_1 + (e^J - 1)S_t dN_t \\ dv_t &= \kappa(\theta - v_t)dt + \sigma\sqrt{v_t} dW_2 \\ dW_1 dW_2 &= \rho dt \end{aligned}$$

with

$$J \sim N(\nu, \delta^2)$$

and

$$m = E(e^J) - 1 = e^{\nu + \frac{1}{2}\delta^2} - 1$$

Other formulations are common in the literature. According to the SDE above, the constructor is given as

```
BatesProcess(const Handle<YieldTermStructure>& riskFreeRate,
             const Handle<YieldTermStructure>& dividendYield,
             const Handle<Quote>& s0,
             Real v0, Real kappa,
             Real theta, Real sigma, Real rho,
             Real lambda, Real nu, Real delta,
             HestonProcess::Discretization d
             = HestonProcess::FullTruncation)
```

The class derives from the `HestonProcess` class, the discretization types are the same as in the Heston case. The `evolve` function accepts a 4 dimensional normal vector `Array dw`. The first 2 random variables are used to construct the Heston process, the last 2 are used to generate the Poisson and normal variables for the jumps. This is clearly suboptimal, as the choice of the type of the random number generator can not be changed. The next slide shows an example, where we generate a path for both the asset and volatility. The output of the function is shown below. It is obvious that the asset path has jumps.

```
Time:  0.1, S(t):  45.4782, V(t):  0.119682
Time:  0.2, S(t):  41.351, V(t):  0.118632
Time:  0.3, S(t):  53.0736, V(t):  0.102813
Time:  0.4, S(t):  44.8663, V(t):  0.104301
Time:  0.5, S(t):  39.2927, V(t):  0.105148
Time:  0.6, S(t):  37.8036, V(t):  0.102556
Time:  0.7, S(t):  38.4143, V(t):  0.0965597
Time:  0.8, S(t):  39.3437, V(t):  0.102807
Time:  0.9, S(t):  35.9401, V(t):  0.101385
Time:  1, S(t):  31.8307, V(t):  0.104224
```

```
void testingStochasticProcesses4(){

        Date refDate=Date(27,Sep,2009);
        Rate riskFreeRate=0.0321;
        Rate dividendRate=0.0128;
        Real spot=52.0;
        Rate vol=0.2144;
        Calendar cal=TARGET();
        DayCounter dc=ActualActual();

        boost::shared_ptr<YieldTermStructure> rdStruct(new FlatForward(refDate,riskFreeRate,dc));
        boost::shared_ptr<YieldTermStructure> rqStruct(new FlatForward(refDate,dividendRate,dc));
        Handle<YieldTermStructure> rdHandle(rdStruct);
        Handle<YieldTermStructure> rqHandle(rqStruct);

        boost::shared_ptr<SimpleQuote> spotQuote(new SimpleQuote(spot));
        Handle<Quote> spotHandle(spotQuote);

        boost::shared_ptr<BlackVolTermStructure> volQuote(new BlackConstantVol(refDate, cal, vol, dc));
        Handle<BlackVolTermStructure> volHandle(volQuote);

        Real v0=0.12, kappa=1.2, theta=0.08, sigma=0.05, rho=-0.6;
        Real lambda=0.25, nu=0.0, delta=0.30;


        boost::shared_ptr<BatesProcess> batesProcess(new BatesProcess(rdHandle,rqHandle,spotHandle,v0,
                kappa,theta,sigma,rho,lambda,nu,delta, HestonProcess::PartialTruncation));


        BigInteger seed=12324;
        MersenneTwisterUniformRng       unifMt(seed);
        BoxMullerGaussianRng<MersenneTwisterUniformRng> bmGauss(unifMt);

        Time dt=0.10,t=0.0;
        Array dw(4),x(2);
        // x is the 2-dimensional process
        x[0]=spotQuote->value();
        x[1]=v0;
        Size numVals=10;

        for(Size j=1;j<=numVals;++j){
                dw[0]=bmGauss.next().value;
                dw[1]=bmGauss.next().value;
```

Thank you!

Broadie, M. and Kaya, "Exact simulation of stochastic volatility and other affine jump diffusion processes," *Operations Research*, 2006, *54* (2), 217–231.

Lord, R., R. Koekkoek, D.J.C. Van Dijk, and R.A. Investments, "A Comparison of biased simulation schemes for stochastic volatility models," 2006.