Introduction to QuantLib C++ source code and basic classes

Introduzione all'Ingegneria Finanziaria

Seminario Professionalizzante

Vincenzo Eugenio Corallo, Ph. D

Sapienza Università di Roma

vincenzo.corallo@uniroma1.it

Why should I use QuantLib?

QuantLib is an open source library for Quantitative Finance. Some reasons to use it are

- It's free!
- If you're starting to code a C++ project from the scratch, you don't have to start from zero. This will allow you to focus on important projects, instead of spending time with coding basic classes such as Date, Interpolation or Yield Curve classes.
- You don't trust open source libraries? You have to admit that open source libraries have some advantages over commercial products. Here, the source code is available and you know exactly what happens in the background. No black-box computations. Clearly, the potential lack of documentation and user support is a disadvantage.
- You already have your own library? Fine. You can still use QuantLib to test and benchmark the results and performance of your own code.
- It is a good place to start learning advanced C++ concepts, such as the usage of design patterns.

Installation instruction

QuantLib

A free/open-source library for quantitative finance

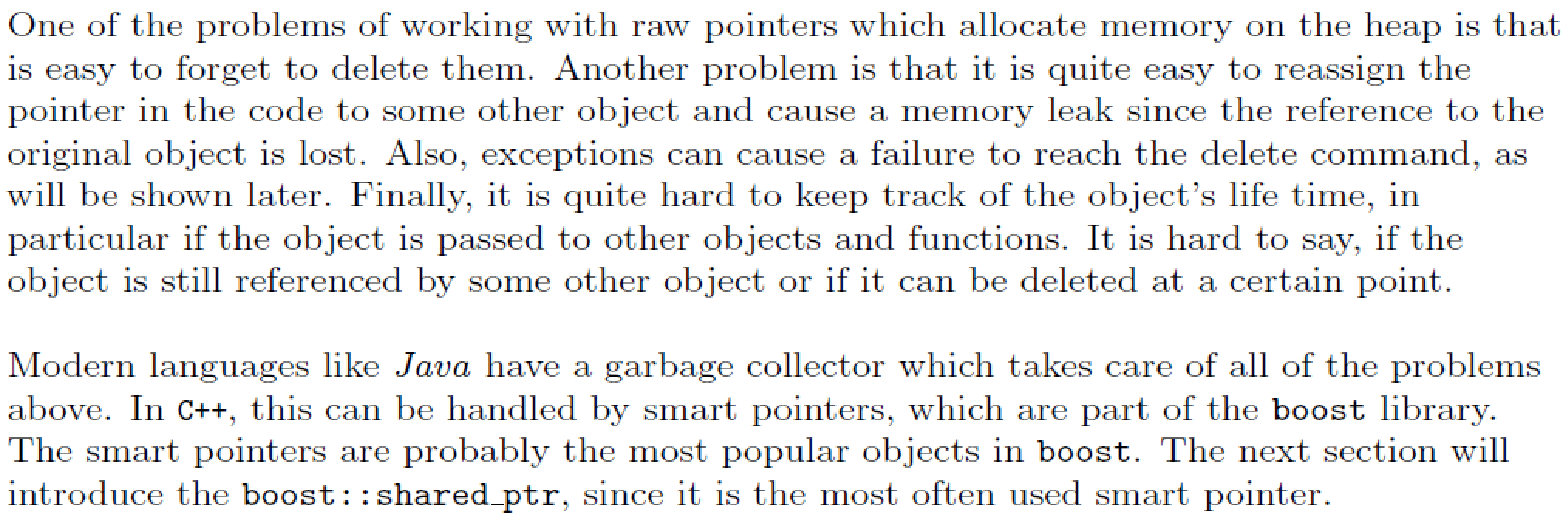
- Detailed instructions for installing QuantLib are available at *https://www.quantlib.org/install/vc10.shtml*

Boost C++ libraries (optional)

For the sake of installing QuantLib and run the source code it would be required to install the Boost libraries first. Boost is a set of libraries for the C++ programming language that provide support for tasks and structures such as linear algebra, pseudorandom number generation, multithreading, image processing, regular expressions, and unit testing. It contains 160 individual libraries (as of version 1.72).

Most of the Boost libraries are licensed under the Boost Software License, designed to allow Boost to be used with both free and proprietary software projects. Many of Boost's founders are on the C++ standards committee, and several Boost libraries have been accepted for incorporation into the C++ Technical Report 1, the C++11 standard (e.g. smart pointers, thread, regex, random, ratio, tuple) and the C++17 standard (e.g. filesystem, any, optional, variant, string_view).

Boost shared pointers (optional)



One of the problems of working with raw pointers which allocate memory on the heap is that is easy to forget to delete them. Another problem is that it is quite easy to reassign the pointer in the code to some other object and cause a memory leak since the reference to the original object is lost. Also, exceptions can cause a failure to reach the delete command, as will be shown later. Finally, it is quite hard to keep track of the object's life time, in particular if the object is passed to other objects and functions. It is hard to say, if the object is still referenced by some other object or if it can be deleted at a certain point.

Modern languages like *Java* have a garbage collector which takes care of all of the problems above. In `C++`, this can be handled by smart pointers, which are part of the `boost` library. The smart pointers are probably the most popular objects in `boost`. The next section will introduce the `boost::shared_ptr`, since it is the most often used smart pointer.

Boost optional (optional)

Boost optional provides a framework to deal with objects whose initialization is optional. An example is a function which returns an object whose construction has either been successful or not. The library provides a framework to check if the object has been initialized or not. Another example is a class member variable, which is initialized by one of the constructors. Before using the variable, a check for initialization has to be performed. The library has to be included with the following header

`<boost/optional.hpp>`

The initialization of an optional object is performed with

- `boost::optional<T> opt(t);`

where `t` is an instance of the template class `T`. The variable `opt` now has a value of 1. If `opt` is not initialized, it has a value of 0. Consequently, we can check if `opt` has one of the values before proceeding. Alternatively, the variable can be checked against the `NULL` keyword.

Boost bind (optional)

The bind class is able to bind any function object argument to a specific value or route input arguments into arbitrary positions. Bind does not need to know anything about the variable types the function accepts/returns, which allows a very convenient syntax. The class works for functions, function objects and function pointers. Example applications are:

- You have a function which accepts many arguments, and you would like to work with one variable only, keeping the others constant.
- You are given a function which you can not modify and you would like to call the function with the parameters being in a different, probably more intuitive order.

Many of the applications above arise in situations where a library is used which accepts a functor where the order or number of variables is different than in the own function interface. The function header is

```
<boost/bind.hpp>
```

Bind can be used very conveniently with `boost function` classes, which will be used throughout this introduction. Boost bind refers to input variables by their number with a _ prefix in the order they are passed to the function. For example, `_1` denotes the first argument of the function.

Dates in QuantLib - 1

A date in QuantLib can be constructed with the following syntax

```
Date(BigInteger serialNumber)
```

where `BigInteger` is the number of days such as 24214, and 0 corresponds to 31.12.1899. This date handling is also known from Excel. The alternative is the construction via

```
Date(Day d, Month m, Year y)
```

Here, `Day`, `Year` are of integer types, while `Month` is a special QuantLib type with

- January: either `QuantLib::January` or `QuantLib::Jan`
- ...
- December: either `QuantLib::December` or `QuantLib::Dec`

After constructing a Date, we can do simple date arithmetics, such as adding/subtracting days and months to the current date.

Dates in QuantLib - 2

Each `Date` object has the following functions

- `weekday()` returns the weekday via the `QuantLib::Weekday` object
- `dayOfMonth()` returns the day of the month as an `QuantLib::Day` object
- `dayOfYear()` returns the day of the year as an `QuantLib::Day` object
- `month()` returns a `QuantLib::Month` object
- `year()` returns a `QuantLib::Year` object
- `serialNumber()` returns a `QuantLib::BigInteger` object

The function names should be self-explaining and all objects can implicitly be converted to integers. For example, you can call `int myDay=myDate.dayOfYear()` instead of `Day myDay=dayOfYear()`

Dates in QuantLib - 3

The QuantLib `Date` class has some useful `static` functions, which give general results, such as whether a given year is a leap year or a given date is the end of the month. The currently available functions are

- `Date::todaysDate()`
- `Date::minDate()`: earliest possible Date in QuantLib
- `Date::maxDate()`: latest possible Date in QuantLib
- `bool::isLeap(Year y)`: is `y` a leap year?
- `Date::endOfMonth(const Date& d)`: what is the end of the month, in which `d` is a day?
- `bool::isEndOfMonth(const Date& d)`: is `d` the end of the month?
- `Date::nextWeekday(const Date& d, Weekday w)`: on which date is the weekday `w` following the date `d`? (e.g. date of the next Friday)
- `Date nthWeekday(Size n, Weekday w,Month m,Year y)`:what is the n-th weekday in the given year and month? (e.g. date of the 3rd Wednesday in July 2010)

Calendars in QuantLib - 1

One of the crucial objects in the daily business is a calendar for different countries which shows the holidays, business days and weekends for the respective country. In QuantLib, a calendar can be set up easily via

```
Calendar myCal=UnitedKingdom()
```

for the UK. Various other calendars are available, for example for Germany, United States, Switzerland, Ukraine, Turkey, Japan, India, Canada and Australia. In addition, special exchange calendars can be initialized for several countries. For example, the Frankfurt Stock Exchange calendar can be initialized via

```
Calendar myCal=Germany(Germany::FrankfurtStockExchange);
```

The following functions are available:

- `bool isBusinessDay(const Date& d)`
- `bool isHoliday(const Date& d)`
- `bool isWeekend(Weekday w)`: is the given weekday part of the weekend?
- `bool isEndOfMonth(const Date& d)`: indicates, whether the given date is the last business day in the month.
- `Date endOfMonth(const Date& d)`: returns the last business day in the month.

Calendars in QuantLib - 2

The calendars are customizable, so you can add and remove holidays in your calendar:

- `void addHoliday(const Date& d)`: adds a user specified holiday
- `void removeHoliday(const Date& d)`: removes a user specified holiday

Furthermore, a function is provided to return a vector of holidays

- `std::vector<Date> holidayList(const Calendar& calendar, const Date& from, const Date& to,bool includeWeekEnds )`: returns a holiday list, including or excluding weekends.

Business Day Conventions in QuantLib - 1

Adjusting a date can be necessary, whenever a transaction date falls on a date that is not a business day. The following Business Day Conventions are available in QuantLib:

- `Following`: the transaction date will be the first following day that is a business day.
- `ModifiedFollowing`: the transaction date will be the first following day that is a business day unless it is in the next month. In this case it will be the first preceding day that is a business day.
- `Preceding`: the transaction date will be the first preceding day that is a business day.
- `ModifiedPreceding`: the transaction date will be the first preceding day that is a business day, unless it is in the previous month. In this case it will be the first following day that is a business day.
- `Unadjusted`

Business Day Conventions in QuantLib - 2

The `QuantLib::Calendar` functions which perform the business day adjustments are

- `Date adjust(const Date&, BusinessDayConvention convention)`
- `Date advance(const Date& date,const Period& period, BusinessDayConvention convention,bool endOfMonth)`: the `endOfMonth` variable enforces the advanced date to be the end of the month if the current date is the end of the month.

Finally, it is possible to count the business days between two dates with the following function

- `BigInteger businessDaysBetween(const Date& from, const Date& to, bool includeFirst, bool includeLast)`: calculates the business days between `from` and `to` including or excluding the initial/final dates.

Daycount Conventions in QuantLib

Daycount conventions are crucial in financial markets. QuantLib offers

- `Actual360`: Actual/360 day count convention
- `Actual365Fixed`: Actual/365 (Fixed)
- `ActualActual`: Actual/Actual day count
- `Business252`: Business/252 day count convention
- `Thirty360`: 30/360 day count convention

The construction is easily performed via

```
DayCounter myCounter=ActualActual();
```

The other conventions can be constructed equivalently. The available functions are

- `BigInteger dayCount(const Date& d1, const Date& d2)`
- `Time yearFraction(const Date&, const Date&)`

Payments Scheduling in QuantLib - 1

An often needed functionality is a schedule of payments, for example for coupon payments of a bond. The task is to produce a series of dates from a start to an end date following a given frequency(e.g. annual, quarterly...). We might want the dates to follow a certain business day convention. And we might want the schedule to go backwards (e.g. start the frequency going backwards from the last date). For example:

- Today is `Date(3,Sep,2009)`. We need a monthly schedule which ends at `Date(15,Dec,2009)`. Going forwards would produce `Date(3,Sep,2009)`,`Date(3,Oct,2009)`,`Date(3,Nov,2009)`,`Date(3,Dec,2009)` and the final date `Date(15,Dec,2009)`.
- Going backwards, on a monthly basis, would produce `Date(3,Sep,2009)`,`Date(15,Sep,2009)`,`Date(15,Oct,2009)`, `Date(15,Nov,2009)`,`Date(15,Dec,2009)`.

The different procedures are given by the `DateGeneration` object and will now be summarized.

Payments Scheduling in QuantLib - 2

- `Backward`: Backward from termination date to effective date.
- `Forward`: Forward from effective date to termination date.
- `Zero`: No intermediate dates between effective date and termination date.
- `ThirdWednesday`: All dates but effective date and termination date are taken to be on the third Wednesday of their month (with forward calculation).
- `Twentieth`: All dates but the effective date are taken to be the twentieth of their month (used for CDS schedules in emerging markets). The termination date is also modified.
- `TwentiethIMM`: All dates but the effective date are taken to be the twentieth of an IMM month (used for CDS schedules). The termination date is also modified.

The schedule is initialized by the `Schedule` class, whose constructor is shown on the next slide.

Schedule Constructor – Example invoke

```
Schedule(const Date& effectiveDate,
 const Date& terminationDate,
 const Period& tenor,
 const Calendar& calendar,
 BusinessDayConvention convention,
 BusinessDayConvention terminationDateConvention,
 DateGeneration::Rule rule,
 bool endOfMonth,
 const Date& firstDate = Date(),
 const Date& nextToLastDate = Date())
```

Payments Scheduling in QuantLib - 3

The variables represent the following

- `effectiveDate,terminationDate`: start/end of the schedule
- `tenor`: the frequency of the schedule (e.g. every 3 months)
- `terminationDateConvention`: allows to specify a special business day convention for the final date.
- `rule`: the generation rule, as previously discussed
- `endOfMonth`: if the effective date is the end of month, enforce the schedule dates to be end of the month too (termination date excluded).
- `firstDate,nextToLastDate`: are optional parameters. If we generate the schedule forwards, the schedule procedure will start from `firstDate` and then increase in the given periods from there. If `nextToLastDate` is set and we go backwards, the dates will be calculated relative to this date.

Payments Scheduling in QuantLib - 4

The `Schedule` object has various useful functions, we will discuss some of them.

- `Size size()`: returns the number of dates
- `const Date& operator[ ](Size i)`: returns the date at index `i`. Alternatively, there is the function `const Date& at(Size i)` to do the same thing.
- `Date previousDate(const Date& refDate)`: returns the previous date in the schedule compared to a reference date.
- `Date nextDate(const Date& refDate)`: returns the next date in the schedule compared to a reference date.
- `const std::vector<Date>& dates()`: returns the whole schedule in a vector.

Numerical Integration in QuantLib

QuantLib provides several procedures to calculate the integral

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

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 number is reached, stop the integration.

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

Numerical Integration via Gaussian Quadrature – 1

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.

Numerical Integration via Gaussian Quadrature – 2

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 several (see the documentation)

Numerical Integration via Gaussian Quadrature – 3

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

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

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 using boost's bind and function libraries.

Gaussian Quadrature Integration Example – 1

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.

Gaussian Quadrature Integration Example – 2

```
#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;
}
```

Gaussian Quadrature Integration Example – 3

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.

Solvers in QuantLib - 1

`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;`

Solvers in QuantLib - 2

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.

Implied Volatiliy via solvers - Example

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.

Interpolation in QuantLib - 1

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)

Interpolation in QuantLib - 2

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 $0.0, 1.0, \ldots, 4.0$ 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.

Linear Interpolation Example - 1

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;
}
```

Linear Interpolation Example - 2

The output of the program is

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

Cubic Splines Interpolation Example - 1

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;
}
```

Cubic Splines Interpolation Example - 2

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 Matrix class in QuantLib - 1

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 Matrix class in QuantLib - 2

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)`

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.

Optimization in QuantLib - 1

One of the most important tools, in particular in calibration procedures, is an optimizer of a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$. 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`

Optimization in QuantLib - 2

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.

Random Numbers in QuantLib

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.

Indexes in QuantLib - 1

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.

Indexes in QuantLib - 2

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;

}
```

Indexes in QuantLib - 3

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.

Interest Rates in QuantLib - 1

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.

Interest Rates in QuantLib - 2

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

Yield Curves in QuantLib - 1

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.

Yield Curves in QuantLib - 2

`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.

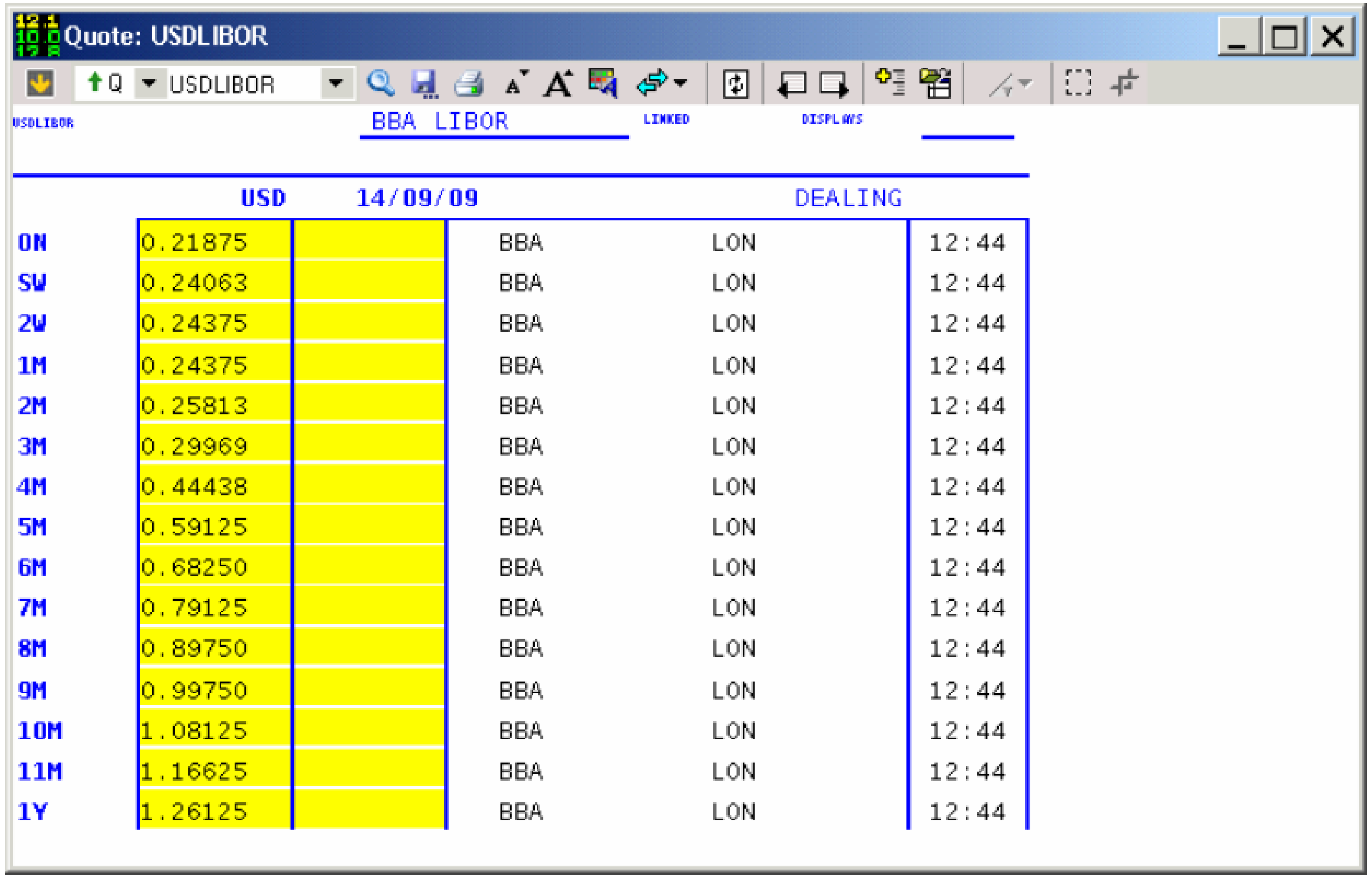
All of the concrete yield classes derive from `YieldTermStructure` and thus inherit automatically the functions above. The only function that they have to implement is the pure virtual `discountImpl(Time)` function. Given the discount factor, all other rates can be derived. This allows for a convenient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

Piecewise Yield Curves Construction - 1

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



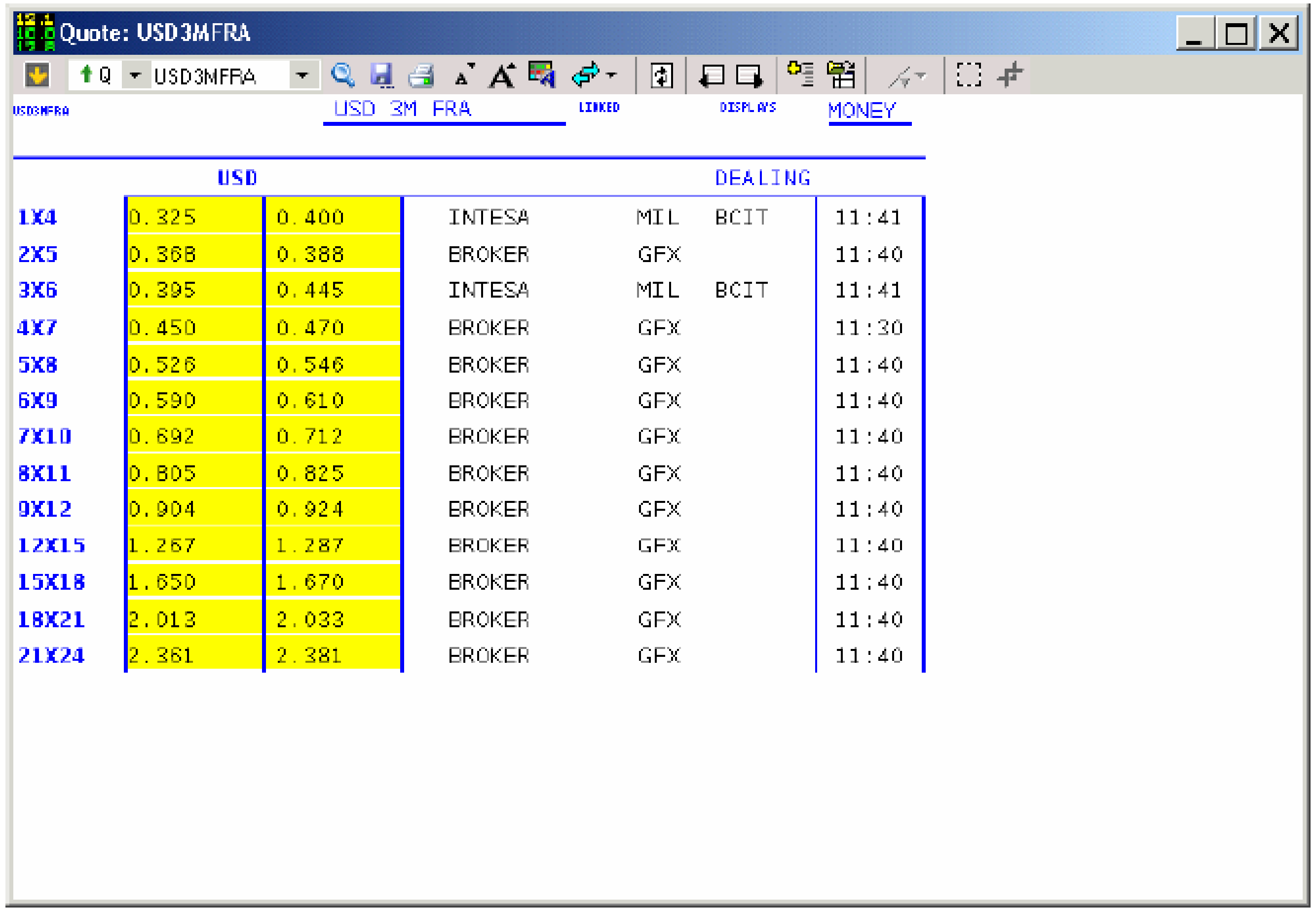
Quote: USDLIBOR

USDLIBOR — BBA LIBOR — LINKED — DISPLAYS

	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



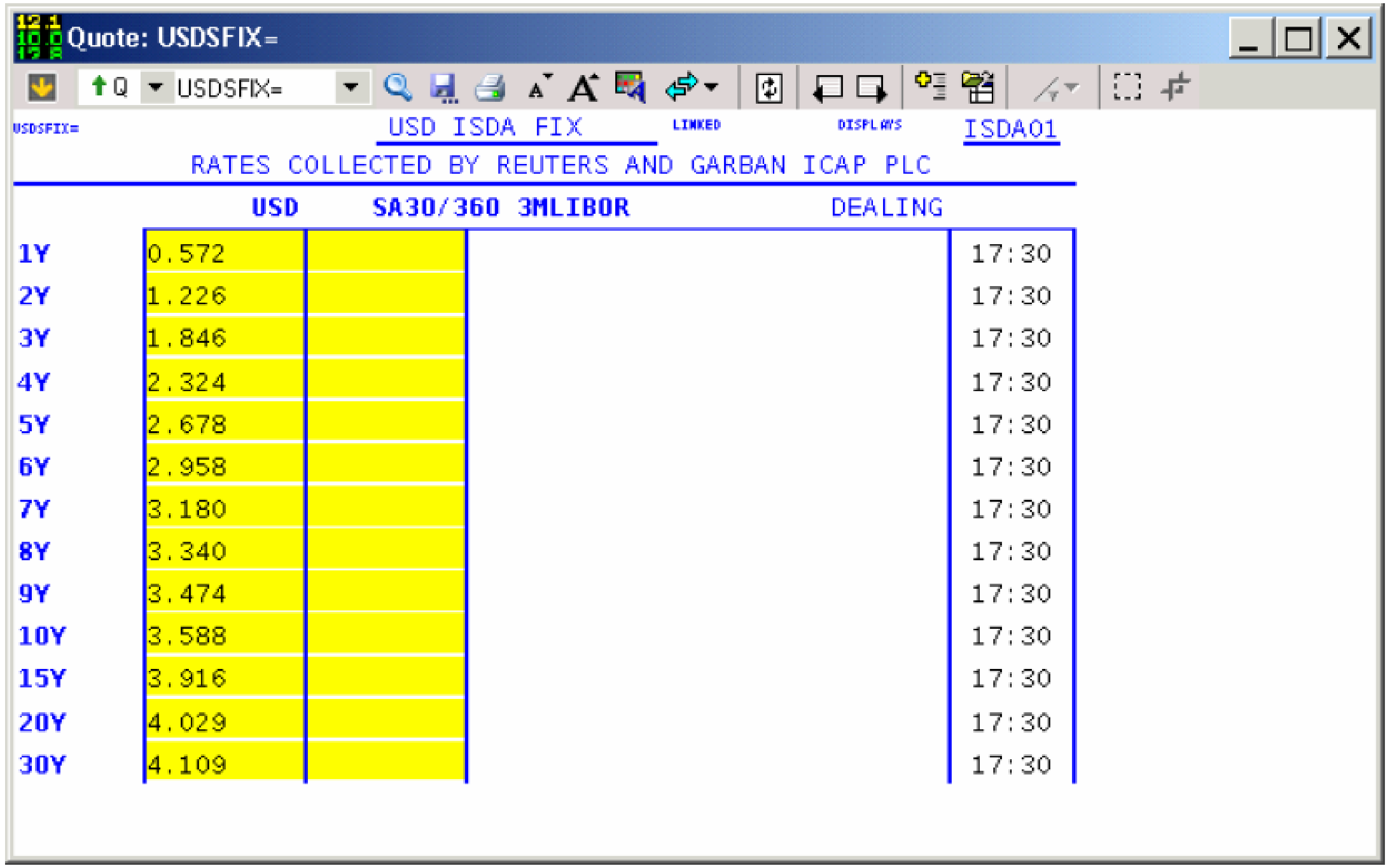
Quote: USD3MFRA

USD3MFRA USD 3M FRA LINKED DISPLAYS MONEY

	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



Quote: USDSFIX=

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

Piecewise Yield Curves Construction - 2

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.

Payoff in QuantLib - 1

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.

Payoff in QuantLib - 2

- `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.

Exercises in QuantLib - 1

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`

Exercises in QuantLib - 2

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