Quantlib cython wrapper Documentation

Release 0.1.1

Didrik Pinte & Patrick Hénaff

Aug 30, 2018

Contents

Contents:

CHAPTER 1

Getting started

1.1 PyQL - an overview

Why building a new set of QuantLib wrappers for Python?

The SWIG wrappers provide a very good coverage of the library but have a number of pain points:

- Few Pythonic optimisations in the syntax: the python code for invoking QuantLib functions looks like the C++ version;
- No docstring or function signature are available on the Python side;
- The debugging is complex, and any customization of the wrapper involves complex programming;
- The build process is monolithic: any change to the wrapper requires the recompilation of the entire project;
- Complete loss of the C++ code organisation with a flat namespace in Python;
- SWIG typemaps development is not that fun.

For those reasons, and to have the ability to expose some of the QuantLib internals that could be very useful on the Python side, we chose another road. PyQL is build on top of Cython and creates a thin Pythonic layer on top of QuantLib. It allows a tight control on the wrapping and provides higher level Python integration.

1.1.1 Features:

- Integration with standard datatypes (like datetime objects) and numpy arrays;
- Simplifed API on the Python side (e.g. usage of Handles completely hidden from the user);
- Support full docstring and expose detailed function signatures to Python;
- Code organised in subpackages to provide a clean namespace, very close to the C++ code organisation;
- Easy extendibility thanks to Cython and shorter build time when adding new functionalities;
- Sphinx documentation.

1.2 Building and installing PyQL

Prerequisites:

- Boost (version 1.55 or higher)
- QuantLib (version 1.5 or higher)
- Cython (version 0.19 or higher)

Once the dependencies have been installed, enter the pyql root directory. Open the setup.py file and configure the Boost and QuantLib include and library directories, then run

```
python setup.py build
```

1.3 Installation from source

The following instructions explain how to build the project from source, on a Linux system. The instructions have been tested on Ubuntu 12.04 LTS.

Prerequisites:

- python 2.7
- C++ development environment
- pandas 0.9

1. Install Boost (taken from a nice post by S. Zebardast)

   (a) Download the Boost source package

   ```
   wget -O boost_1_55_0.tar.gz \
   http://sourceforge.net/projects/boost/files/boost/1.55.0/boost_1_55_0.tar.gz/
   ↪download
   tar xzvf boost_1_55_0.tar.gz
   ```

   (b) Make sure you have the required libraries

   ```
   sudo apt-get update
   sudo apt-get install build-essential g++ python-dev autotools-dev libicu-dev␣
   ↪libbz2-dev
   ```

   (c) Build and install

   ```
   cd boost_1_55_0
   sudo ./bootstrap.sh --prefix=/usr/local
   sudo ./b2 install
   ```

   If /usr/local/lib is not in your path:

   ```
   sudo sh -c 'echo "/usr/local/lib" >> /etc/ld.so.conf.d/local.conf'
   ```

   and finally:

   ```
   sudo ldconfig
   ```

2. Install Quantlib

   (a) Download Quantlib 1.5 from Quantlib.org and copy to /opt

   ```
   wget -O QuantLib-1.5.tar.gz  \
   http://sourceforge.net/projects/quantlib/files/QuantLib/1.5/QuantLib-1.5.tar.
   ↪gz/download
   sudo cp QuantLib-1.5.tar.gz /opt
   ```

   (b) Extract the Quantlib folder

   ```
   cd /opt
   sudo tar xzvf QuantLib-1.5.tar.gz
   ```

   (c) Configure QuantLib

   ```
   cd QuantLib-1.5
   ./configure --disable-static CXXFLAGS=-O2 --with-boost-include=/usr/local/
   ↪include --with-boost-lib=/usr/local/lib
   ```

   (d) Make and install

   ```
   make
   sudo make install
   ```

3. Install Cython. While you can install Cython from source, we strongly recommend to install Cython via pip:

   ```
   pip install cython
   ```

   If you do not have the required permissions to install Python packages in the system path, you can install Cython in your local user account via:

   ```
   pip install --user cython
   ```

4. Download pyql (https://github.com/enthought/pyql), then extract, build and test:

   ```
   $ cd ~/dev/pyql
   $ make build
   $ make tests
   ```

If you have installed QuantLib in a directory different from `/opt`, edit the *setup.py* file before running make and update the `INCLUDE_DIRS` and `LIBRARY_DIRS` to point to your installation of QuantLib.

1.4 Installation from source on Windows

The following instructions explain how to build the project from source, on a Windows system. The instructions have been tested on Windows 7 32bit with Visual Studio 2008.

Prerequisites:

- python 2.7 (e.g. Canopy with Cython 0.20 or above)
- pandas 0.9

1. Install Quantlib

   (a) Install the latest version of Boost from sourceforge. You can get the binaries of 1.55 for windows 32 or 64bit depending on your target.

   (b) Download Quantlib 1.5 from Quantlib.org and unzip locally

   (c) Extract the Quantlib folder
   (d) Open the QuantLib_vc9 solution with Visual Studio
   (e) Patch ql/settings.hpp.

   In the ql/settings.hpp file, update the Settings class defintion as following (line 37):

   ```
   class __declspec(dllexport) Settings : public Singleton<Settings> {
   ```

   (f) In the QuantLib project properties

   - Change "General" -> "Configuration type" to "Dynamic Library (DLL)"
   - Apply
   - Add the Boost include directory to "C/C++" -> "Additional Include Directories"
   - Apply

   Do a first build to get all the object files generated

   (g) Generate the def file:

   In your PyQL clone, got the scripts directory, and edit the main function. Set *input_directory* to the Release directory where your object files are and change the *output_file* if appropriate (symbol_win32.def is the default) ! The def file is platform specific (you can't reuse a 32bit def file for a 64bit linker).

   This will generate a def file of about 44 Mb with all the needed symbols for PyQL compilation.

   (h) Build the dll with the new def file

   - Change "Linker" -> "Input" -> "Module definition file" to point to def file you just generated.
   - Apply the changes and build the project

   (a) Copy the QuantLib.dll to a directory which is on the PATH (or just the PyQL directory if you're in development mode)

2. Install Cython. While you can install Cython from source, we strongly recommend to install Cython via the Canopy Package Manager, another Python distribution or via pip:

   ```
   pip install cython
   ```

   If you do not have the required permissions to install Python packages in the system path, you can install Cython in your local user account via:

   ```
   pip install --user cython
   ```

3. Build and test pyql

   Edit the setup.py to make sure the INCLUDE_DIRS and LIBRARY_DIRS point to the correct directories.

   ```
   PS C:\dev\pyql> python setup.py build
   PS C:\dev\pyql> python setup.py install
   ```

   Note: Development mode

   If you want to build the library in place and test things, you can do:

```
PS C:\dev\pyql> python setup.py build_ext --inplace
PS C:\dev\pyql> python -m unittest discover -v
```

CHAPTER 2

Tutorial

CHAPTER 3

User's guide

3.1 Business dates

Business dates handling capabilities is provided by the quantlib.time subpackage. The three core components are Date, Period and Calendar.

3.1.1 Date

A date in QuantLib can be constructed with the following syntax:

```
Date(serial_number)
```

where serial_number 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, month, year)
```

Here, day, month and year are of integer. A set of month constant are available in the date module (January, . . . , December or Jan, . . . , Dec)

After constructing a Date, we can do simple date arithmetics, such as adding/subtracting days and months to the current date. Furthermore, the known convenient operators such as +=,-= can be used.

It is possible to add a Period to a date. Period can be created using time units or frequency:

```
Period(frequency)
Period(lenght, time_units)
```

Frequencies are defined with the following constants: NoFrequency, Once, Annual, Semiannual, EveryFourthMonth, Quartely, Bimonthly, Monthly, EveryFourthWeek, Biweekly, Weekly, Daily and OtherFrequency.

Time units are constants defined in the date module: `Days`, `Weeks`, `Months`, `Years`.

Each `Date` object has the following properties:

- `weekday` returns the weekday using the weekday constants defined in the date module (Sunday to Saturday and Sun to Sat).
- `day` returns the day of the month
- `day_of_year` returns the day of the year
- `month` returns the month
- `year` returns the year
- `serial` returns a the serial number of this date

The `quantlib.time.date` module 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:

- `today()`
- `mindate()`: earliest possible Date in QuantLib
- `maxdate()`: latest possible Date in QuantLib
- `is_leap()`: is year a leap year?
- `end_of_month()`: what is the end of the current month the date is in?
- `is_end_of_month(date)()`: is date the end of the month?
- `next_weekday(date, weekday)()`: on which date is the weekday following the date? (e.g. date of the next Friday)
- `nth_weekday(n, weekday, month, year)()`: what is the n-th weekday in the given year and month? (e.g. date of the 3rd Wednesday in July 2010)

3.1.2 Calendars

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:

```
uk_calendar = UnitedKingdom()
```

for the UK. Calendars implementation are available in the `quantlib.time.calendars` subpackage.

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 New-York Stock Exchange calendar can be initialized via:

```
us_calendar = UnitedStates(NYSE);
```

The following functions are available:

- `is_business_day(date)()`
- `is_holiday(date)()`
- `is_weekend(week_day)()`: is the given weekday part of the weekend?
- `is_end_of_month(date)()`: indicates, whether the given date is the last business day in the month.
- `end_of_month(date)()`: returns the last business day in the month.

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

- `addHoliday(date)()`

- `removeHoliday(date)()`: removes a user specified holiday

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

- `holidayList(calendar, from_date, to_date, include_weekends=False)()`: returns a holiday list, including or excluding weekends. This function returns a DateList object that provides an list/iterator-like interface on top of the C++ QuantLib date vector.

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 the calendar module:

- **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**

The Calendar functions which perform the business day adjustments are :

- **adjust(date, business_day_convention)**
- **advance(date,period, business_day_convention, end_of_month)**: the end_of_month 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:

- **business_days_between(from_date, to_date, include_first, include_last)** calculates the business days between from and to including or excluding the initial/final dates.

We will demonstrate an example by using the Frankfurt Stock Exchange calendar and the dates Date(31,Oct,2009) and Date(1,Jan,2010). From the first date, we advance 2 months in the future, which is December, 31st. Since this is a holiday and the next business day is in the next month, we can check the Modified Following conversion. The Modified Preceding conversion can be checked for January, 1st 2010:

```
frankfcal   = Germany(FrankfurtStockExchange);
first_date  = Date(31,Oct,2009)
second_date = Date(1,Jan ,2010);

print "Date 2       Adv:", frankfcal.adjust(second_date , Preceding)
print "Date 2       Adv:", frankfcal.adjust(second_date , ModifiedPreceding)

mat = Period(2,Months)

print "Date 1 Month Adv:", \
  frankfcal.avance(
        first_date, period=mat, convention=Following,
        end_of_month=False
    )
print "Date 1 Month Adv:", \
  frankfcal.avance(
        first_date, period=mat, convention=ModifiedFollowing,
        end_of_month=False
  )
print "Business Days Between:", \
  frankfcal.business_days_between(
```

(continues on next page)

(continued from previous page)

```
        first_date, second_date, False, False
  )
```

and the output will give

```
Date 2       Adv: 30/12/2009
Date 2       Adv:  4/01/2010
Date 1 Month Adv:  4/01/2010
Date 1 Month Adv: 30/12/2009
Business Days Between: 41
```

3.1.3 Day counters

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:

```
myCounter = ActualActual()
```

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

- dayCount(from_date, to_date)
- yearFraction(from_date, to_date)

TODO : add example

3.1.4 Date generation

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:

- 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:

```
Schedule(effective_date , termination_date, tenor, calendar, convention ,
         termination_date_convention , date_gen_rule,
         end_of_month, first_date, next_to_last_date)
```

The arguments represent the following

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

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

- size(): returns the number of dates
- at(i) : returns the date at index i.
- previous_date(ref_date): returns the previous date in the schedule compared to a reference date.
- next_date(ref_date): returns the next date in the schedule compared to a reference date.
- dates(): returns the whole schedule in a DateList object.

3.1.5 Performance considerations

In [3]: %timeit QuantLib.Date.todaysDate() + QuantLib.Period(10, QuantLib.Days) 100000 loops, best of 3: 9.71 us per loop

In [4]: %timeit datetime.date.today() + datetime.timedelta(days=10) 100000 loops, best of 3: 3.55 us per loop

In [5]: %timeit quantlib.date.today() + quantlib.date.Period(10, quantlib.date.Days) 100000 loops, best of 3: 2.17 us per loop

3.2 Reference

The mlab module provides high-level functions suitable for easily performing common quantitative finance calculations. These functions use as input standardized data structures that are provided to limit the amount of data transformation needed to string functions together.

The mlab functions often use pandas data frames as inputs. In order to encourage inter-operability between functions, we have defined a number of standard data structures. The column names of these data frames are defined in the ''names" module. The standardized data structures should be created with the functions provided in the ''data_structures" module.

3.2.1 Names

The column names of all datasets are defined in names.py. A column name should always be referenced by the corresponding variable name, and not by a character string. For example, refer to the 'Strike' column of an option_quotes data set by:

```
import quantlib.reference.names as nm
strike = option_quotes[nm.STRIKE]
```

rather than:

```
strike = option_quotes['Strike']
```

3.2.2 Data Structures Templates

These data structures are defined to facilitate the inter-operability of the high level functions found in the 'mlab' module.

Option Quotes

This data structure contains the necessary data for calibrating a stochastic model for the underlying asset, also known as volatility model.

An option quotes data structure with 10 rows is created with the statements:

```
import quantlib.reference.data_structures as ds
option_quotes = ds.option_quotes_template().reindex(index=range(10))
```

Risk-free Rate and Dividends

When calibrating a volatility model, the default algorithm is to compute the implied term structure of risk-free rate and dividend yield from the option data, using the call-put parity relationship. The result of this calculation is the 'riskfree_dividend' data structure.

3.3 Mlab

The mlab module provides high-level functions suitable for easily performing common quantitative finance calculations. These functions use as input standardized data structures that are provided to limit the amount of data transformation needed to string functions together.

3.3.1 Standardized data structures

3.3.2 Curve building

3.3.3 Asset pricing

3.4 Notebooks

The notebooks and scripts folder provide sample calculations performed with QuantLib.

3.4.1 Getting started

In order to use the notebokks, you need to install:

- Ipython 0.13
- pylab
- matplotlib

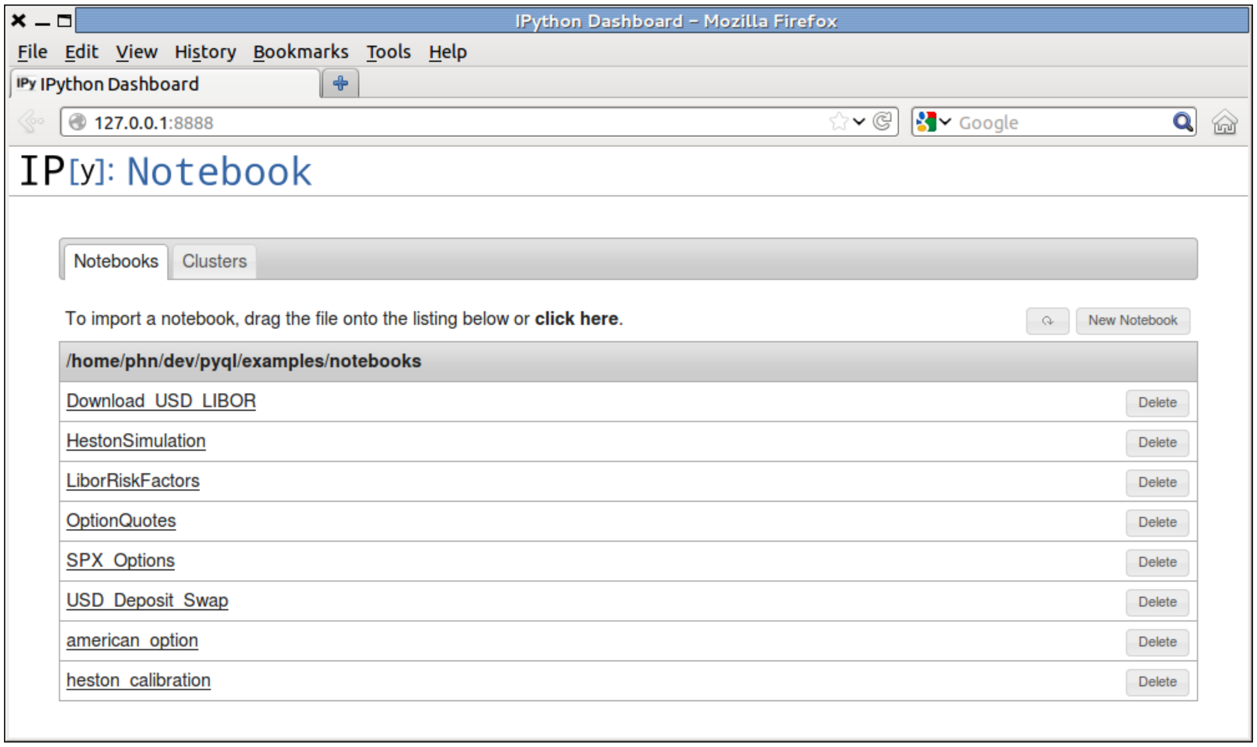
Make sure that pyQL is in the PYTHONPATH. You can access the notebooks with the command:

```
ipython notebook --pylab inline <path to the notebooks folder> --browser=<browser␣
↪name>
```

For example, on a linux system where the pyql project is located in ~/dev, the command to view the notebooks with the Firefox browser would be:

```
ipython notebook --pylab inline ~/dev/pyql/examples/notebooks  --browser=firefox
```

The browser will start and display a menu with several notebooks. As of October 2012, you should see 8 notebooks, as shown below:



To import a notebook, drag the file onto the listing below or **click here**.

/home/phn/dev/pyql/examples/notebooks

Download_USD_LIBOR
HestonSimulation
LiborRiskFactors
OptionQuotes
SPX_Options
USD_Deposit_Swap
american_option
heston_calibration

Fig. 1: Notebook menu in the Firefox browser.

CHAPTER 4

Reference guide

4.1 Reference documentation for the `quantlib` package

The API of the Python wrappers try to be as close as possible to the C++ original source but keeping a Pythonic simple access to classes, methods and functions. Most of the complex structures related to proper memory management are completely hidden being the Python layers (for example boost::shared_ptr and Handle).

4.1.1 `quantlib`

`quantlib.settings`

`quantlib.quotes`

`quantlib.cashflow`

`quantlib.index`

`quantlib.interest_rate`

4.1.2 `quantlib.currency`

`quantlib.currency.currency`

`quantlib.currency.currencies`

4.1.3 `quantlib.indexes`

4.1.4 `quantlib.instruments`

`quantlib.instruments.bonds`

`quantlib.instruments.option`

`quantlib.instruments.credit_default_swap`

4.1.5 quantlib.math

4.1.6 quantlib.model.equity

4.1.7 `quantlib.pricingengines`

`quantlib.pricingengines.swaption`

4.1.8 `quantlib.processes`

4.1.9 `quantlib.termstructures`

`quantlib.termstructures.inflation_term_structure`

`quantlib.termstructures.default_term_structure`

`yield_term_structure`

:`quantlib.termstructures.yields`

mod:~*quantlib.termstructures.yields.rate_helpers*

`bond_helpers`

`flat_forward`

`zero_curve`

The objective is to make available in python a set of modules that exactly mirror the QL class hierarchy. For example, QL provides a class named `SimpleQuote`, that represents a simple price measurement. The C++ class is defined as follows:

```
class SimpleQuote : public Quote {
  public:
    SimpleQuote(Real value = Null<Real>());
    Real value() const;
    bool isValid() const;
    Real setValue(Real value = Null<Real>());
};
```

After wrapping the C++ class, this class is now available in python:

```
from quantlib.quotes import SimpleQuote
spot = SimpleQuote(3.14)
print('Spot %f' % spot.value)
```

A couple of observations are worth mentioning:

- pyql preserves the module hierarchy of QuantLib: the SimpleQuote class is defined in the quote module in C++.
- pyql exposes QuantLib in a pythonic fashion: instead of exposing the accessor value(), pyql implements the property value.

4.2.1 The Interface Code

To expose QL class `foo`, you need to create three files. For the sake of standardization, they should be named as follows:

_foo.pxd A header file to declare the C++ class being exposed,

foo.pxd A header file where the corresponding python class is declared

foo.pyx The implementation of the corresponding python class

The content of each file is now described in details.

4.2.2 Declaration of the QL classes to be exposed

This file contains the declaration of the QL class being exposed. For example, the header file `_quotes.pxd` is as follows:

```
include 'types.pxi'

from libcpp cimport bool

cdef extern from 'ql/quote.hpp' namespace 'QuantLib':
      cdef cppclass Quote:
         Quote() except +
         Real value() except +
         bool isValid() except +

cdef extern from 'ql/quotes/simplequote.hpp' namespace 'QuantLib':

      cdef cppclass SimpleQuote(Quote):
         SimpleQuote(Real value) except +
         Real setValue(Real value) except +
```

In this file, we declare the class `SimpleQuote` and its parent `Quote`. The syntax is almost identical to the corresponding C++ header file. The types used in declaring arguments are defined in `types.pxi`.

The clause 'except +' signals that the method may throw an exception. It is indispensible to append this clause to every declaration. Without it, an exception thrown in QL will terminate the python process.

4.2.3 Declaration of the python class

The second header file declares the python classes that will be wrapping the QL classes. The file `quotes.pxd` is reproduced below:

```
cimport _quote as _qt
from quantlib.handle cimport shared_ptr

cdef class Quote:
   cdef shared_ptr[_qt.Quote]* _thisptr
```

Notice that in our header files we use 'Quote' to refer the the C++ class (in file _quote.pxd) and to the python class (in file quote.pxd). To avoid confusion we use the following convention:

- the C++ class is always refered to as `_qt.Quote`.
- the python class is always refered to as `Quote`

The cython wrapper class holds a reference to the QL C++ class. As we do not want to do any memory handling on the Python side, we always wrap the C++ object into a boost shared pointer that is deallocated properly when deallocation the Cython extension.

4.2.4 Implementation of the python class

The third file contains the implementation of the cython wrapper class. As an illustration, the implementation of the `SingleQuote` python class is reproduced below:

```
cdef class SimpleQuote(Quote):
   def __init__(self, double value=0.0):
      self._thisptr = new shared_ptr[_qt.Quote](new _qt.SimpleQuote(value))

   def __dealloc__(self):
      if self._thisptr is not NULL:
         del self._thisptr # properly deallocates the shared_ptr and
                           # probably the target object if not referenced

   def __str__(self):
      return 'Simple Quote: %f' % self._thisptr.get().value()

   property value:
      def __get__(self):
         if self._thisptr.get().isValid():
            return self._thisptr.get().value()
         else:
            return None

     def __set__(self, double value):
         (<_qt.SimpleQuote*>self._thisptr.get()).setValue(value)
```

The `__init__` method invokes the C++ constructor, which returns a boost shared pointer.

Properties are used to give a more pythonic flavor to the wrapping. In python, we get the value of the `SimpleQuote` with the syntax `spot.value` rather than `spot.value()`, had we exposed directly the C++ accessor.

Remember from the previous section that `_thisptr` is a shared pointer on a `Quote`, which is a virtual class. The `setValue` method is defined in the `SimpleQuote` concrete class, and the shared pointer must therefore be cast into a `SimpleQuote` shared pointer in order to invoke `setValue()`.

4.2.5 Managing C++ references using shared_ptr

All the Cython extension references should be declared using shared_ptr. The `__dealloc__` method should always delete the shared_ptr but never the target pointer!

Every time a shared_ptr reference is received, never assigns the target pointer to a local pointer variables as it might be deallocated. Always use the copy constructor of the shared_ptr to get a local copy of it, stack allocated (there is no need to use new).

CHAPTER 5

Roadmap

- Provide binary version for Mac, Windows and Linux
- Increase the Python coverage for C++ classes
- Make the API more pythonic and user friendly to abstract more of the complex C++ constructions
- Provide a better integration for large datasets
- Investigate potential OpenMP support

CHAPTER 6

Documentation

List of online resources useful for the project:

- add examples from http://quantlib.org/slides/dima-ql-intro-1.pdf
- Fixed income - indexes (see http://quantlib.org/slides/dima-ql-intro-2.pdf p78)

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index