QUANTLIB PYTHON COOKBOOK

Hands-on Quantitative Finance in Python

Goutham Balaraman and Luigi Ballabio

QuantLib Python Cookbook

Luigi Ballabio and Goutham Balaraman

This book is available at https://leanpub.com/quantlibpythoncookbook

This version was published on 2025-07-16

Tweet This Book!

Please help Luigi Ballabio and Goutham Balaraman by spreading the word about this book on Twitter!

The suggested hashtag for this book is #quantlib.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#quantlib

Also By Luigi Ballabio

A QuantLib Guide

Implementing QuantLib

Contents

The cover image is in the public domain and available from the New York Public Library[1]. The cover font is Open Sans Condensed, released by Steve Matteson[2] under the Apache License version 2.0[3].

[1] http://digitalcollections.nypl.org/items/510d47df-335e-a3d9-e040-e00a18064a99
[2] https://twitter.com/SteveMatteson1
[3] http://www.apache.org/licenses/LICENSE-2.0

A note on Python and C++

The choice of using the QuantLib Python bindings and Jupyter was due to their interactivity, which make it easier to demonstrate features, and to the fact that the platform provides out of the box excellent modules like `matplotlib` for graphing and `pandas` for data analysis.

This choice might seem to leave C++ users out in the cold. However, it's easy enough to translate the Python code shown here into the corresponding C++ code. An example of such translation is shown in the appendix.

Code conventions used in this book

The recipes in this cookbook are written as Jupyter notebooks[4], and follow their structure: blocks of explanatory text, like the one you're reading now, are mixed with cells containing Python code (*inputs*) and the results of executing it (*outputs*). The code and its output—if any—are marked by `In [N]` and `Out [N]`, respectively, with `N` being the index of the cell. You can see an example in the computations below:

```
In [1]: def f(x, y):
            return x + 2*y

In [2]: a = 4
        b = 2
        f(a, b)

Out[2]: 8
```

By default, Jupyter displays the result of the last instruction as the output of a cell, like it did above; however, `print` statements can display further results.

```
In [3]: print(a)
        print(b)
        print(f(b, a))

Out[3]: 4
        2
        10
```

Jupyter also knows a few specific data types, such as Pandas data frames, and displays them in a more readable way:

```
In [4]: import pandas as pd
        pd.DataFrame({ 'foo': [1,2,3], 'bar': ['a','b','c'] })

Out[4]:
```

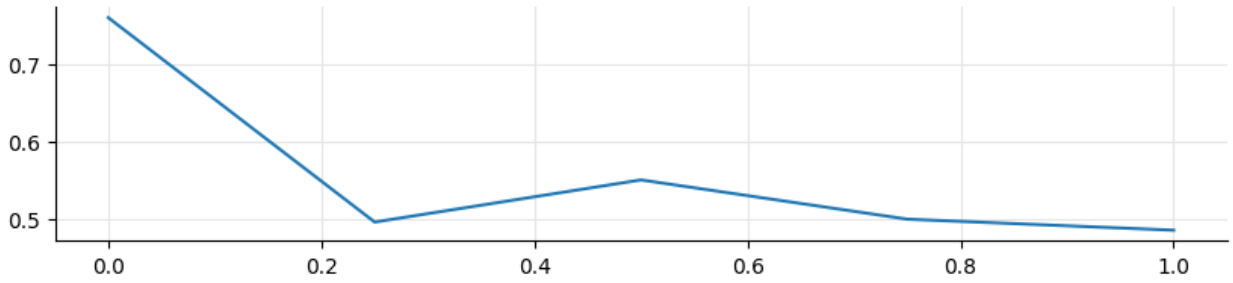
[4] http://jupyter.org/

	foo	bar
0	1	a
1	2	b
2	3	c

The index of the cells shows the order of their execution. Jupyter doesn't constrain it; however, in all of the recipes of this book the cells were executed in sequential order as displayed. All cells are executed in the global Python scope; this means that, as we execute the code in the recipes, all variables, functions and classes defined in a cell are available to the ones that follow.

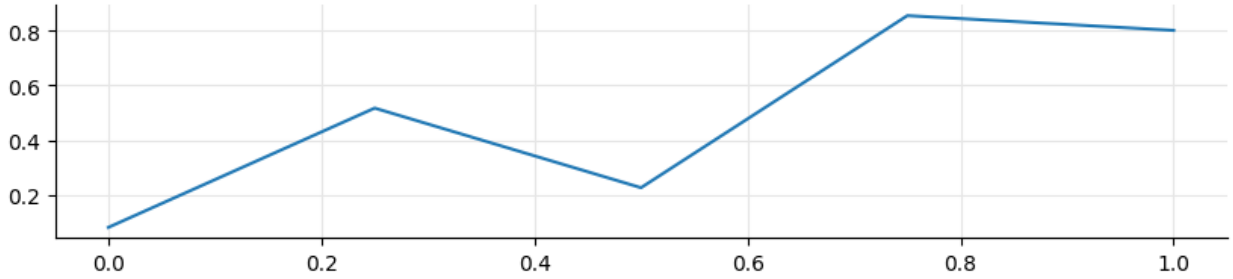
Notebooks can also include plots, as in the following cell:

```
In [5]: %matplotlib inline
        import numpy as np
        import utils
        f, ax = utils.plot(figsize=(10,2))
        ax.plot([0, 0.25, 0.5, 0.75, 1.0], np.random.random(5))

Out[5]: [<matplotlib.lines.Line2D at 0x7f615b41b3a0>]
```



As you might have noted, the cell above also printed a textual representation of the object returned from the plot, since it's the result of the last instruction in the cell. To prevent this, cells in the recipes might have a semicolon at the end, as in the next cell. This is just a quirk of the Jupyter display system, and it doesn't have any particular significance; I'm mentioning it here just so that you dont't get confused by it.

```
In [6]: f, ax = utils.plot(figsize=(10,2))
        ax.plot([0, 0.25, 0.5, 0.75, 1.0], np.random.random(5));
```



Finally, the `utils` module that I imported above is a short module containing convenience functions, mostly related to plots, for the notebooks in this collection. It's not necessary to understand its implementation to follow the recipes, and therefore we won't cover it here; but if you're interested and want to look at it, it's included in the zip archive that you can download from Leanpub if you purchased the book.

1. QuantLib basics

In this chapter we will introduce some of the basic concepts such as `Date`, `Period`, `Calendar` and `Schedule`. These are QuantLib constructs that are used throughout the library in creation of instruments, models, term structures etc.

```
In [1]: import QuantLib as ql
        import pandas as pd
```

Date Class

The `Date` object can be created using the constructor as `Date(day, month, year)`. It would be worthwhile to pay attention to the fact that `day` is the first argument, followed by `month` and then the `year`. This is different from the Python `datetime` object instantiation.

```
In [2]: date = ql.Date(31, 3, 2015)
        print(date)
```

```
Out[2]: March 31st, 2015
```

The fields of the `Date` object can be accessed using the `month()`, `dayOfMonth()` and `year()` methods. The `weekday()` method can be used to fetch the day of the week.

```
In [3]: print("%d-%d-%d" %(date.month(),
                           date.dayOfMonth(),
                           date.year()))
```

```
Out[3]: 3-31-2015
```

```
In [4]: date.weekday() == ql.Tuesday
```

```
Out[4]: True
```

The `Date` objects can also be used to perform arithmetic operations such as advancing by days, weeks, months etc. Periods such as weeks or months can be denoted using the `Period` class. `Period` object constructor signature is `Period(num_periods, period_type)`. The `num_periods` is an integer and represents the number of periods. The `period_type` can be `Weeks`, `Months` and `Years`.

```
In [5]: type(date+1)

Out[5]: QuantLib.QuantLib.Date


In [6]: print("Add a day      : {0}".format(date + 1))
        print("Subtract a day : {0}".format(date - 1))
        print("Add a week     : {0}".format(date + ql.Period(1, ql.Weeks)))
        print("Add a month    : {0}".format(date + ql.Period(1, ql.Months)))
        print("Add a year     : {0}".format(date + ql.Period(1, ql.Years)))

Out[6]: Add a day      : April 1st, 2015
        Subtract a day : March 30th, 2015
        Add a week     : April 7th, 2015
        Add a month    : April 30th, 2015
        Add a year     : March 31st, 2016
```

One can also do logical operations using the `Date` object.

```
In [7]: print(date == ql.Date(31, 3, 2015))
        print(date > ql.Date(30, 3, 2015))
        print(date < ql.Date(1, 4, 2015))
        print(date != ql.Date(1, 4, 2015))

Out[7]: True
        True
        True
        True
```

The `Date` object is used in setting valuation dates, issuance and expiry dates of instruments. The `Period` object is used in setting tenors, such as that of coupon payments, or in constructing payment schedules.

Calendar Class

The `Date` arithmetic above did not take holidays into account. But valuation of different securities would require taking into account the holidays observed in a specific exchange or country. The `Calendar` class implements this functionality for all the major exchanges. Let us take a look at a few examples here.

```
In [8]: date = ql.Date(31, 3, 2015)
        us_calendar = ql.UnitedStates(ql.UnitedStates.GovernmentBond)
        italy_calendar = ql.Italy()

        period = ql.Period(60, ql.Days)
        raw_date = date + period
        us_date = us_calendar.advance(date, period)
        italy_date = italy_calendar.advance(date, period)

        print("Add 60 days: {0}".format(raw_date))
        print("Add 60 business days in US: {0}".format(us_date))
        print("Add 60 business days in Italy: {0}".format(italy_date))

Out[8]: Add 60 days: May 30th, 2015
        Add 60 business days in US: June 24th, 2015
        Add 60 business days in Italy: June 26th, 2015
```

The `addHoliday` and `removeHoliday` methods in the calendar can be used to add and remove holidays to the calendar respectively. If a calendar has any missing holidays or has a wrong holiday, then these methods come handy in fixing the errors. The `businessDaysBetween` method helps find out the number of business days between two dates per a given calendar. Let us use this method on the `us_calendar` and `italy_calendar` as a sanity check.

```
In [9]: us_busdays = us_calendar.businessDaysBetween(date, us_date)
        italy_busdays = italy_calendar.businessDaysBetween(date, italy_date)

        print("Business days US: {0}".format(us_busdays))
        print("Business days Italy: {0}".format(italy_busdays))

Out[9]: Business days US: 60
        Business days Italy: 60
```

In valuation of certain deals, more than one calendar's holidays are observed. QuantLib has `JointCalendar` class that allows you to combine the holidays of two or more calendars. Let us take a look at a working example.

```
In [10]: joint_calendar = ql.JointCalendar(us_calendar, italy_calendar)

         joint_date = joint_calendar.advance(date, period)
         joint_busdays = joint_calendar.businessDaysBetween(date, joint_date)

         print("Add 60 business days in US-Italy: {0}".format(joint_date))
         print("Business days US-Italy: {0}".format(joint_busdays))

Out[10]: Add 60 business days in US-Italy: June 29th, 2015
         Business days US-Italy: 60
```

Schedule Class

The `Schedule` object is necessary in creating coupon schedules or call schedules. `Schedule` object constructors have the following signature:

```
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())
```

and

```
Schedule(const std::vector<Date>&,
         const Calendar& calendar,
         BusinessDayConvention rollingConvention)
```

```
In [11]: effective_date = ql.Date(1, 1, 2015)
         termination_date = ql.Date(1, 1, 2016)
         tenor = ql.Period(ql.Monthly)
         calendar = ql.UnitedStates(ql.UnitedStates.GovernmentBond)
         business_convention = ql.Following
         termination_business_convention = ql.Following
         date_generation = ql.DateGeneration.Forward
         end_of_month = False

         schedule = ql.Schedule(effective_date,
                                termination_date,
                                tenor,
```

```
                                calendar,
                                business_convention,
                                termination_business_convention,
                                date_generation,
                                end_of_month)

        pd.DataFrame({'date': list(schedule)})
```

Out[11]:

	date
0	January 2nd, 2015
1	February 2nd, 2015
2	March 2nd, 2015
3	April 1st, 2015
4	May 1st, 2015
5	June 1st, 2015
6	July 1st, 2015
7	August 3rd, 2015
8	September 1st, 2015
9	October 1st, 2015
10	November 2nd, 2015
11	December 1st, 2015
12	January 4th, 2016

Here we have generated a `Schedule` object that will contain dates between `effective_date` and `termination_date` with the `tenor` specifying the `Period` to be `Monthly`. The `calendar` object is used for determining holidays. Here we have chosen the convention to be the day following holidays. That is why we see that holidays are excluded in the list of dates.

The `Schedule` class can handle generation of dates with irregularity in schedule. The two extra parameters `firstDate` and `nextToLastDate` parameters along with a combination of forward or backward date generation rule can be used to generate short or long stub payments at the front or back end of the schedule. For example, the following combination of `firstDate` and backward generation rule creates a short stub in the front on the January 15, 2015.

```
In [12]: # short stub in the front
         effective_date = ql.Date(1, 1, 2015)
         termination_date = ql.Date(1, 1, 2016)
         first_date = ql.Date(15, 1, 2015)
         schedule = ql.Schedule(effective_date,
                                termination_date,
                                tenor,
                                calendar,
                                business_convention,
                                termination_business_convention,
```

```
                                ql.DateGeneration.Backward,
                                end_of_month,
                                first_date)

        pd.DataFrame({'date': list(schedule)})
```

Out[12]:

	date
0	January 2nd, 2015
1	January 15th, 2015
2	February 2nd, 2015
3	March 2nd, 2015
4	April 1st, 2015
5	May 1st, 2015
6	June 1st, 2015
7	July 1st, 2015
8	August 3rd, 2015
9	September 1st, 2015
10	October 1st, 2015
11	November 2nd, 2015
12	December 1st, 2015
13	January 4th, 2016

Using the `nextToLastDate` parameter along with the forward date generation rule creates a short stub at the back end of the schedule.

```
In [13]: # short stub at the back
         effective_date = ql.Date(1, 1, 2015)
         termination_date = ql.Date(1, 1, 2016)
         penultimate_date = ql.Date(15, 12, 2015)
         schedule = ql.Schedule(effective_date,
                                termination_date,
                                tenor,
                                calendar,
                                business_convention,
                                termination_business_convention,
                                ql.DateGeneration.Forward,
                                end_of_month,
                                ql.Date(),
                                penultimate_date)

         pd.DataFrame({'date': list(schedule)})
```

Out[13]:

	date
0	January 2nd, 2015
1	February 2nd, 2015
2	March 2nd, 2015
3	April 1st, 2015
4	May 1st, 2015
5	June 1st, 2015
6	July 1st, 2015
7	August 3rd, 2015
8	September 1st, 2015
9	October 1st, 2015
10	November 2nd, 2015
11	December 1st, 2015
12	December 15th, 2015
13	January 4th, 2016

Using the backward generation rule along with the `firstDate` allows us to create a long stub in the front. Below the first two dates are longer in duration than the rest of the dates.

```
In [14]: # long stub in the front
         first_date = ql.Date(1, 2, 2015)
         effective_date = ql.Date(15, 12, 2014)
         termination_date = ql.Date(1, 1, 2016)
         schedule = ql.Schedule(effective_date,
                                termination_date,
                                tenor,
                                calendar,
                                business_convention,
                                termination_business_convention,
                                ql.DateGeneration.Backward,
                                end_of_month,
                                first_date)

         pd.DataFrame({'date': list(schedule)})

Out[14]:
```

	date
0	December 15th, 2014
1	February 2nd, 2015
2	March 2nd, 2015
3	April 1st, 2015
4	May 1st, 2015
5	June 1st, 2015
6	July 1st, 2015
7	August 3rd, 2015
8	September 1st, 2015
9	October 1st, 2015
10	November 2nd, 2015
11	December 1st, 2015
12	January 4th, 2016

Similarly the usage of `nextToLastDate` parameter along with forward date generation rule can be used to generate long stub at the back of the schedule.

```
In [15]: # long stub at the back
         effective_date = ql.Date(1, 1, 2015)
         penultimate_date = ql.Date(1, 12, 2015)
         termination_date = ql.Date(15, 1, 2016)
         schedule = ql.Schedule(effective_date,
                                termination_date,
                                tenor,
                                calendar,
                                business_convention,
                                termination_business_convention,
                                ql.DateGeneration.Forward,
                                end_of_month,
                                ql.Date(),
                                penultimate_date)

         pd.DataFrame({'date': list(schedule)})

Out[15]:
```

	date
0	January 2nd, 2015
1	February 2nd, 2015
2	March 2nd, 2015
3	April 1st, 2015
4	May 1st, 2015
5	June 1st, 2015
6	July 1st, 2015
7	August 3rd, 2015
8	September 1st, 2015
9	October 1st, 2015
10	November 2nd, 2015
11	December 1st, 2015
12	January 15th, 2016

Below the `Schedule` is generated from a list of dates.

```
In [16]: dates = [ql.Date(2,1,2015), ql.Date(2, 2,2015),
                  ql.Date(2,3,2015), ql.Date(1,4,2015),
                  ql.Date(1,5,2015), ql.Date(1,6,2015),
                  ql.Date(1,7,2015), ql.Date(3,8,2015),
                  ql.Date(1,9,2015), ql.Date(1,10,2015),
                  ql.Date(2,11,2015), ql.Date(1,12,2015),
                  ql.Date(4,1,2016)]
         rolling_convention = ql.Following

         schedule = ql.Schedule(dates, calendar,
                                rolling_convention)

         pd.DataFrame({'date': list(schedule)})

Out[16]:
```

	date
0	January 2nd, 2015
1	February 2nd, 2015
2	March 2nd, 2015
3	April 1st, 2015
4	May 1st, 2015
5	June 1st, 2015
6	July 1st, 2015
7	August 3rd, 2015
8	September 1st, 2015
9	October 1st, 2015
10	November 2nd, 2015
11	December 1st, 2015
12	January 4th, 2016

Interest Rate

The `InterestRate` class can be used to store the interest rate with the compounding type, day count and the frequency of compounding. Below we show how to create an interest rate of 5.0% compounded annually, using Actual/Actual day count convention.

```
In [17]: annual_rate = 0.05
         day_count = ql.ActualActual(ql.ActualActual.ISDA)
         compound_type = ql.Compounded
         frequency = ql.Annual

         interest_rate = ql.InterestRate(annual_rate,
                                         day_count,
                                         compound_type,
                                         frequency)
         print(interest_rate)

Out[17]: 5.000000 % Actual/Actual (ISDA) Annual compounding
```

Lets say if you invest a dollar at the interest rate described by `interest_rate`, the `compoundFactor` method in the `InterestRate` object gives you how much your investment will be worth after any period. Below we show that the value returned by `compound_factor` for 2 years agrees with the expected compounding formula.

```
In [18]: t = 2.0
         print(interest_rate.compoundFactor(t))
         print((1+annual_rate)*(1.0+annual_rate))

Out[18]: 1.1025
         1.1025
```

The `discountFactor` method returns the reciprocal of the `compoundFactor` method. The discount factor is useful while calculating the present value of future cashflows.

```
In [19]: print(interest_rate.discountFactor(t))
         print(1.0/interest_rate.compoundFactor(t))

Out[19]: 0.9070294784580498
         0.9070294784580498
```

A given interest rate can be converted into other compounding types and compounding frequency using the `equivalentRate` method.

```
In [20]: new_frequency = ql.Semiannual
         new_interest_rate = interest_rate.equivalentRate(compound_type,
                                                          new_frequency, t)
         print(new_interest_rate)

Out[20]: 4.939015 % Actual/Actual (ISDA) Semiannual compounding
```

The discount factor for the two `InterestRate` objects, `interest_rate` and `new_interest_rate` are the same, as shown below.

```
In [21]: print(interest_rate.discountFactor(t))
         print(new_interest_rate.discountFactor(t))

Out[21]: 0.9070294784580498
         0.9070294784580495
```

The `impliedRate` method in the `InterestRate` class takes compound factor to return the implied rate. The `impliedRate` method is a static method in the `InterestRate` class and can be used without an instance of `InterestRate`. Internally the `equivalentRate` method invokes the `impliedRate` method in its calculations.

Conclusion

This chapter gave an introduction to the basics of QuantLib. Here we explained the `Date`, `Schedule`, `Calendar` and `InterestRate` classes.

2. Instruments and pricing engines

In this notebook, I'll show how instruments and their available engines can monitor changes in their input data.

Setup

To begin, we import the QuantLib module and set up the global evaluation date.

```
In [1]: import QuantLib as ql
```

```
In [2]: today = ql.Date(7, ql.March, 2014)
        ql.Settings.instance().evaluationDate = today
```

The instrument

As a sample instrument, we'll take a textbook example: a European option.

Building the option requires only the specification of its contract, so its payoff (it's a call option with strike at 100) and its exercise, three months from today's date. Market data will be selected and passed later, depending on the calculation methods.

```
In [3]: option = ql.EuropeanOption(ql.PlainVanillaPayoff(ql.Option.Call, 100.0),
                                   ql.EuropeanExercise(ql.Date(7, ql.June, 2014)))
```

First pricing method: analytic Black-Scholes formula

The different pricing methods are implemented as pricing engines holding the required market data. The first we'll use is the one encapsulating the analytic Black-Scholes formula.

First, we collect the quoted market data. We'll assume flat risk-free rate and volatility, so they can be expressed by `SimpleQuote` instances: those model numbers whose value can change and that can notify observers when this happens. The underlying value is at 100, the risk-free value at 1%, and the volatility at 20%.

```
In [4]: u = ql.SimpleQuote(100.0)
        r = ql.SimpleQuote(0.01)
        sigma = ql.SimpleQuote(0.20)
```

In order to build the engine, the market data are encapsulated in a Black-Scholes process object. First we build flat curves for the risk-free rate and the volatility…

```
In [5]: riskFreeCurve = ql.FlatForward(0, ql.TARGET(),
                                       ql.QuoteHandle(r), ql.Actual360())
        volatility = ql.BlackConstantVol(0, ql.TARGET(),
                                         ql.QuoteHandle(sigma), ql.Actual360())
```

…then we instantiate the process with the underlying value and the curves we just built. The inputs are all stored into handles, so that we could change the quotes and curves used if we wanted. I'll skip over this for the time being.

```
In [6]: process = ql.BlackScholesProcess(ql.QuoteHandle(u),
                                         ql.YieldTermStructureHandle(riskFreeCurve),
                                         ql.BlackVolTermStructureHandle(volatility))
```

Once we have the process, we can finally use it to build the engine…

```
In [7]: engine = ql.AnalyticEuropeanEngine(process)
```

…and once we have the engine, we can set it to the option and evaluate the latter.

```
In [8]: option.setPricingEngine(engine)

In [9]: print(option.NPV())

Out[9]: 4.155543462156206
```

Depending on the instrument and the engine, we can also ask for other results; in this case, we can ask for Greeks.

```
In [10]: print(option.delta())
         print(option.gamma())
         print(option.vega())

Out[10]: 0.5302223303784392
         0.03934493301271913
         20.109632428723106
```

Market changes

As I mentioned, market data are stored in `Quote` instances and thus can notify the option when any of them changes. We don't have to do anything explicitly to tell the option to recalculate: once we set a new value to the underlying, we can simply ask the option for its NPV again and we'll get the updated value.

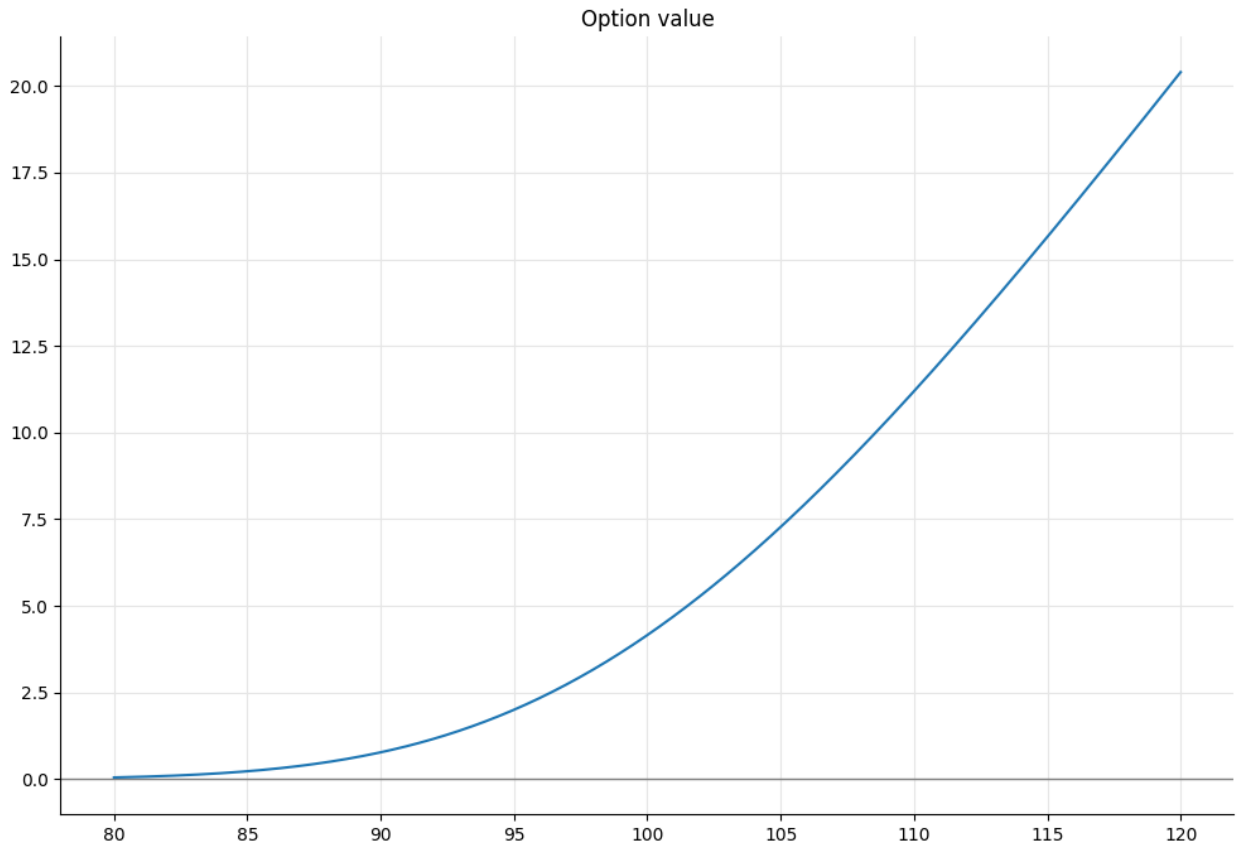
```
In [11]: u.setValue(105.0)
         print(option.NPV())

Out[11]: 7.27556357927846
```

Just for showing off, we can use this to graph the option value depending on the underlying asset value. After a bit of graphic setup (don't pay attention to the man behind the curtains)...

```
In [12]: %matplotlib inline
         import numpy as np
         from IPython.display import display
         import utils
```

...we can take an array of values from 80 to 120, set the underlying value to each of them, collect the corresponding option values, and plot the results.

```
In [13]: f, ax = utils.plot()
         xs = np.linspace(80.0, 120.0, 400)
         ys = []
         for x in xs:
             u.setValue(x)
             ys.append(option.NPV())
         ax.set_title('Option value')
         utils.highlight_x_axis(ax)
         ax.plot(xs, ys);
```



Other market data also affect the value, of course.

```
In [14]: u.setValue(105.0)
         r.setValue(0.01)
         sigma.setValue(0.20)

In [15]: print(option.NPV())

Out[15]: 7.27556357927846
```

We can see it when we change the risk-free rate…

```
In [16]: r.setValue(0.03)

In [17]: print(option.NPV())

Out[17]: 7.624029148527754
```

…or the volatility.

```
In [18]: sigma.setValue(0.25)

In [19]: print(option.NPV())

Out[19]: 8.531296969971573
```

Date changes

Just as it does when inputs are modified, the value also changes if we advance the evaluation date. Let's look first at the value of the option when its underlying is worth 105 and there's still three months to exercise…

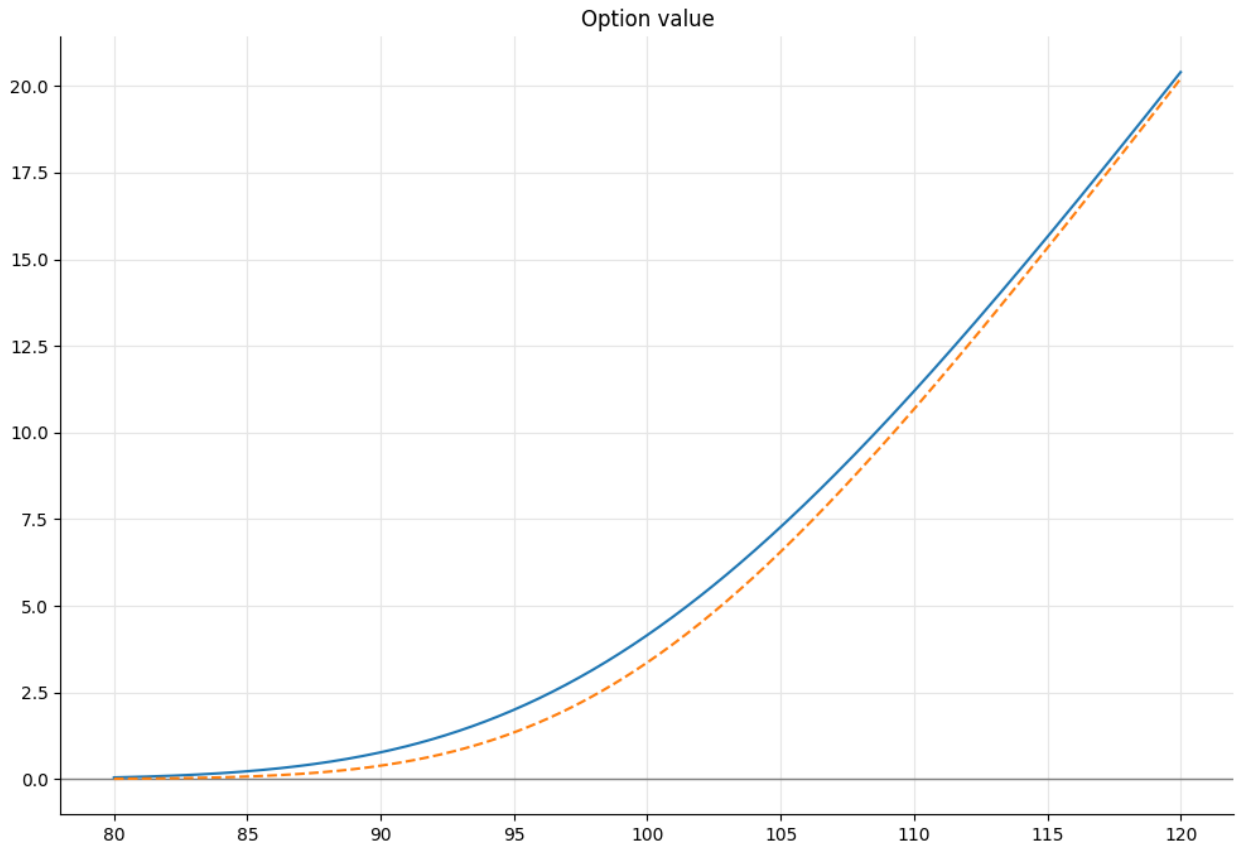
```
In [20]: u.setValue(105.0)
         r.setValue(0.01)
         sigma.setValue(0.20)
         print(option.NPV())

Out[20]: 7.27556357927846
```

…and then move to a date two months before exercise.

```
In [21]: ql.Settings.instance().evaluationDate = ql.Date(7, ql.April, 2014)
```

Again, we don't have to do anything explicitly: we just ask the option its value, and as expected it has decreased, as can also be seen by updating the plot.

```
In [22]: print(option.NPV())

Out[22]: 6.560073820974377

In [23]: ys = []
         for x in xs:
             u.setValue(x)
             ys.append(option.NPV())
         ax.plot(xs, ys, '--')
         display(f)
```



In the default library configuration, the returned value goes down to 0 when we reach the exercise date.

```
In [24]: ql.Settings.instance().evaluationDate = ql.Date(7, ql.June, 2014)

In [25]: print(option.NPV())

Out[25]: 0.0
```

Other pricing methods

The pricing-engine mechanism allows us to use different pricing methods. For comparison, I'll first set the input data back to what they were previously and output the Black-Scholes price.

```
In [26]: ql.Settings.instance().evaluationDate = today
         u.setValue(105.0)
         r.setValue(0.01)
         sigma.setValue(0.20)

In [27]: print(option.NPV())

Out[27]: 7.27556357927846
```

Let's say that we want to use a Heston model to price the option. What we have to do is to instantiate the corresponding class with the desired inputs...

```
In [28]: model = ql.HestonModel(
             ql.HestonProcess(
                 ql.YieldTermStructureHandle(riskFreeCurve),
                 ql.YieldTermStructureHandle(ql.FlatForward(0, ql.TARGET(),
                                                            0.0, ql.Actual360())),
                 ql.QuoteHandle(u),
                 0.04, 0.1, 0.01, 0.05, -0.75))
```

...pass it to the corresponding engine, and set the new engine to the option.

```
In [29]: engine = ql.AnalyticHestonEngine(model)
         option.setPricingEngine(engine)
```

Asking the option for its NPV will now return the value according to the new model.

```
In [30]: print(option.NPV())

Out[30]: 7.295356086978629
```

Lazy recalculation

One last thing. Up to now, we haven't really seen evidence of notifications going around. After all, the instrument might just have recalculated its value every time, regardless of notifications. What I'm going to show, instead, is that the option doesn't just recalculate every time anything changes; it also avoids recalculations when nothing has changed.

We'll switch to a Monte Carlo engine, which takes a few seconds to run the required simulation.

```
In [31]: engine = ql.MCEuropeanEngine(process, "PseudoRandom",
                                       timeSteps=20,
                                       requiredSamples=250000)
         option.setPricingEngine(engine)
```

When we ask for the option value, we have to wait for the calculation to finish…

```
In [32]: %time print(option.NPV())

Out[32]: 7.248816340995633
         CPU times: user 1.59 s, sys: 0 ns, total: 1.59 s
         Wall time: 1.58 s
```

…but a second call to the `NPV` method will be instantaneous when made before anything changes. In this case, the option didn't calculate its value; it just returned the result that it cached from the previous call.

```
In [33]: %time print(option.NPV())

Out[33]: 7.248816340995633
         CPU times: user 1.53 ms, sys: 18 µs, total: 1.55 ms
         Wall time: 1.65 ms
```

If we change anything (e.g., the underlying value)…

```
In [34]: u.setValue(104.0)
```

…the option is notified of the change, and the next call to `NPV` will again take a while.

```
In [35]: %time print(option.NPV())

Out[35]: 6.598508180782789
         CPU times: user 1.56 s, sys: 4.84 ms, total: 1.57 s
         Wall time: 1.56 s
```

3. EONIA curve bootstrapping

In the next notebooks, I'll reproduce the results of the paper by F. M. Ametrano and M. Bianchetti, *Everything You Always Wanted to Know About Multiple Interest Rate Curve Bootstrapping but Were Afraid to Ask* (April 2, 2013). The paper is available at SSRN: http://ssrn.com/abstract=2219548.

```
In [1]: %matplotlib inline
        import math
        import utils

In [2]: import QuantLib as ql

In [3]: today = ql.Date(11, ql.December, 2012)
        ql.Settings.instance().evaluationDate = today
```

First try

We start by instantiating helpers for all the rates used in the bootstrapping process, as reported in figure 25 of the paper.

The first three instruments are three 1-day deposit that give us discounting between today and the day after spot. They are modeled by three instances of the `DepositRateHelper` class with a tenor of 1 day and a number of fixing days going from 0 (for the deposit starting today) to 2 (for the deposit starting on the spot date).

```
In [4]: helpers = [
            ql.DepositRateHelper(ql.QuoteHandle(ql.SimpleQuote(rate/100)),
                                 ql.Period(1,ql.Days), fixingDays,
                                 ql.TARGET(), ql.Following,
                                 False, ql.Actual360())
            for rate, fixingDays in [(0.04, 0), (0.04, 1), (0.04, 2)]
        ]
```

Then, we have a series of OIS quotes for the first month. They are modeled by instances of the `OISRateHelper` class with varying tenors. They also require an instance of the `Eonia` class, which doesn't need a forecast curve and can be shared between the helpers.

```
In [5]: eonia = ql.Eonia()

In [6]: helpers += [
            ql.OISRateHelper(2, ql.Period(*tenor),
                             ql.QuoteHandle(ql.SimpleQuote(rate/100)), eonia)
            for rate, tenor in [(0.070, (1,ql.Weeks)), (0.069, (2,ql.Weeks)),
                                (0.078, (3,ql.Weeks)), (0.074, (1,ql.Months))]
        ]
```

Next, five OIS forwards on ECB dates. For these, we need to instantiate the `DatedOISRateHelper` class and specify start and end dates explicitly.

```
In [7]: helpers += [
            ql.DatedOISRateHelper(start_date, end_date,
                                  ql.QuoteHandle(ql.SimpleQuote(rate/100)), eonia)
            for rate, start_date, end_date in [
                ( 0.046, ql.Date(16,ql.January,2013), ql.Date(13,ql.February,2013)),
                ( 0.016, ql.Date(13,ql.February,2013), ql.Date(13,ql.March,2013)),
                (-0.007, ql.Date(13,ql.March,2013), ql.Date(10,ql.April,2013)),
                (-0.013, ql.Date(10,ql.April,2013), ql.Date(8,ql.May,2013)),
                (-0.014, ql.Date(8,ql.May,2013), ql.Date(12,ql.June,2013))]
        ]
```

Finally, we add OIS quotes up to 30 years.

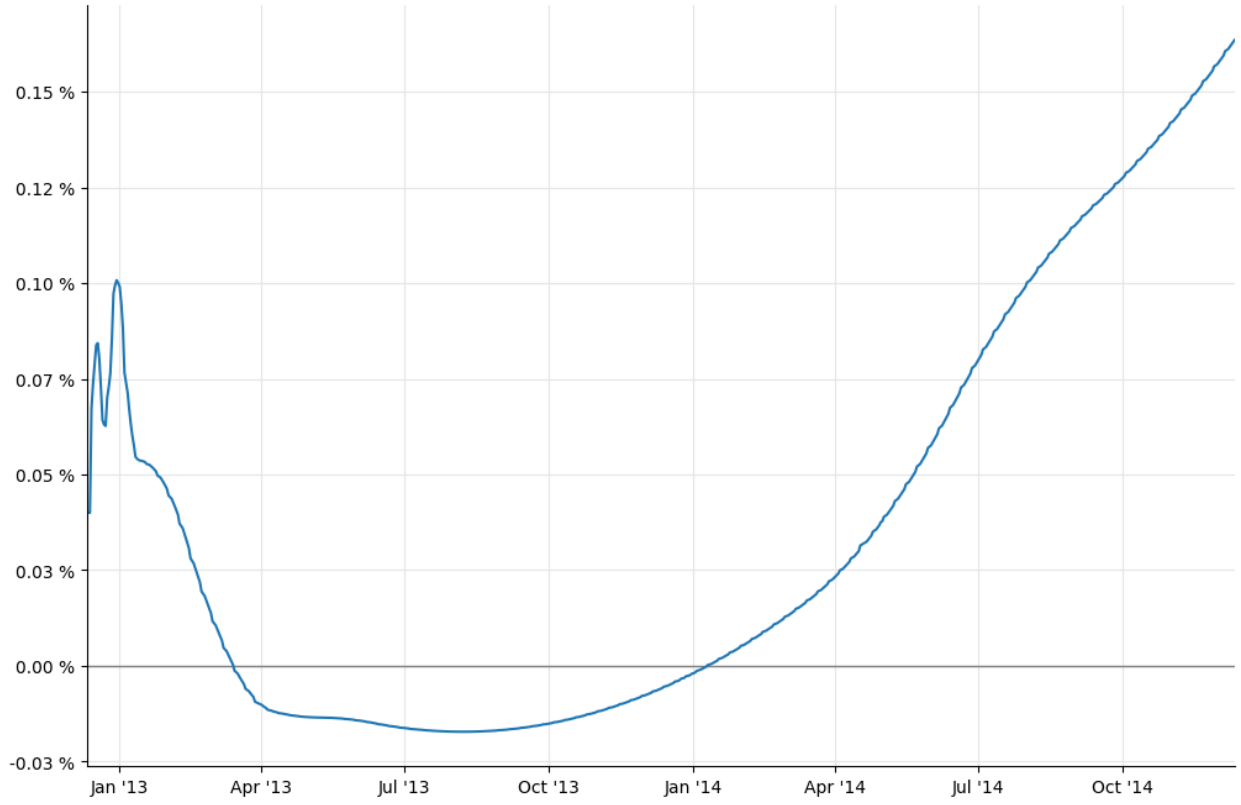
```
In [8]: helpers += [
            ql.OISRateHelper(2, ql.Period(*tenor),
                             ql.QuoteHandle(ql.SimpleQuote(rate/100)), eonia)
            for rate, tenor in [(0.002, (15,ql.Months)), (0.008, (18,ql.Months)),
                                (0.021, (21,ql.Months)), (0.036, (2,ql.Years)),
                                (0.127, (3,ql.Years)), (0.274, (4,ql.Years)),
                                (0.456, (5,ql.Years)), (0.647, (6,ql.Years)),
                                (0.827, (7,ql.Years)), (0.996, (8,ql.Years)),
                                (1.147, (9,ql.Years)), (1.280, (10,ql.Years)),
                                (1.404, (11,ql.Years)), (1.516, (12,ql.Years)),
                                (1.764, (15,ql.Years)), (1.939, (20,ql.Years)),
                                (2.003, (25,ql.Years)), (2.038, (30,ql.Years))]
        ]
```

The curve is an instance of `PiecewiseLogCubicDiscount` (corresponding to the `PiecewiseYieldCurve<Discount,LogCubic>` class in C++; I won't repeat the argument for this choice made in section 4.5 of the paper). We let the reference date of the curve move with the global evaluation date, by specifying it as 0 days after the latter on the TARGET calendar. The day counter chosen is not of much consequence, as it is only used internally to convert dates into times. Also, we enable extrapolation beyond the maturity of the last helper; that is mostly for convenience as we retrieve rates to plot the curve near its far end.

```
In [9]: eonia_curve_c = ql.PiecewiseLogCubicDiscount(0, ql.TARGET(),
                                                     helpers, ql.Actual365Fixed())
        eonia_curve_c.enableExtrapolation()
```

To compare the curve with the one shown in figure 26 of the paper, we can retrieve daily overnight rates over its first two years and plot them:

```
In [10]: today = eonia_curve_c.referenceDate()
         end = today + ql.Period(2,ql.Years)
         dates = [ ql.Date(serial) for serial in range(today.serialNumber(),
                                                       end.serialNumber()+1) ]
         rates_c = [ eonia_curve_c.forwardRate(d, ql.TARGET().advance(d,1,ql.Days),
                                               ql.Actual360(), ql.Simple).rate()
                     for d in dates ]

In [11]: _, ax = utils.plot()
         utils.highlight_x_axis(ax)
         utils.plot_curve(ax, dates, [(rates_c,'-')], format_rates=True)
```



However, we still have work to do. Out plot above shows a rather large bump at the end of 2012 which is not present in the paper. To remove it, we need to model properly the turn-of-year effect.

Turn-of-year jumps

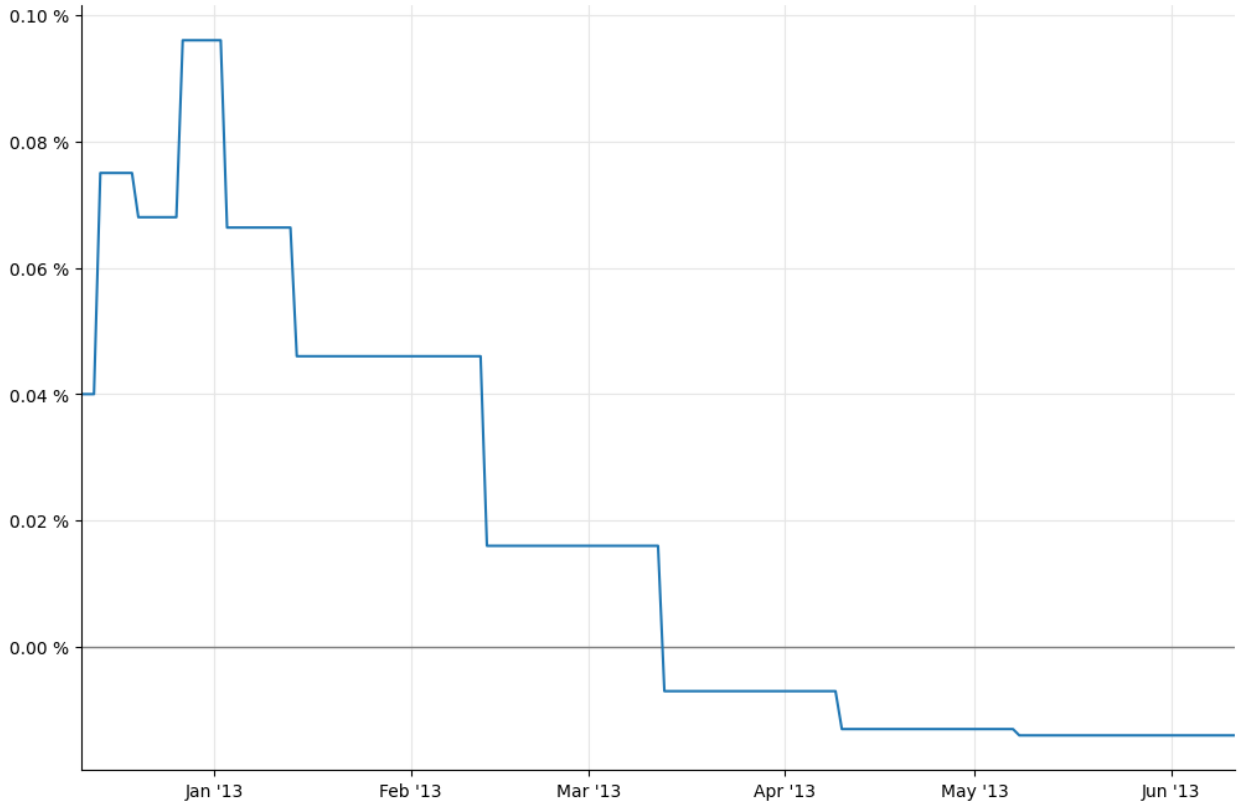
As explained in section 4.8 of the paper, the turn-of-year effect is a jump in interest rates due to an increased demand for liquidity at the end of the year. The jump is embedded in any quoted rates that straddles the end of the year and must be treated separately; the `YieldTermStructure` class allows this by taking any number of jumps, modeled as additional discount factors, and applying them at the specified dates.

Our problem is to estimate the size of the jump. To simplify analysis, we turn to flat forward rates instead of log-cubic discounts; thus, we instantiate a `PiecewiseFlatForward` curve (corresponding to `PiecewiseYieldCurve<ForwardRate,BackwardFlat>` in C++).

```
In [12]: eonia_curve_ff = ql.PiecewiseFlatForward(0, ql.TARGET(),
                                                  helpers, ql.Actual365Fixed())
         eonia_curve_ff.enableExtrapolation()
```

To show the jump more clearly, I'll restrict the plot to the first 6 months:

```
In [13]: end = today + ql.Period(6,ql.Months)
         dates = [ ql.Date(serial) for serial in range(today.serialNumber(),
                                                       end.serialNumber()+1) ]
         rates_ff = [ eonia_curve_ff.forwardRate(d, ql.TARGET().advance(d,1,ql.Days),
                                                 ql.Actual360(), ql.Simple).rate()
                      for d in dates ]

In [14]: _, ax = utils.plot()
         utils.highlight_x_axis(ax)
         utils.plot_curve(ax, dates, [(rates_ff,'-')], format_rates=True)
```



As we see, the forward ending at the beginning of January 2013 is out of line. In order to estimate the jump, we need to estimate a "clean" forward that doesn't include it.

A possible estimate (although not the only one) can be obtained by interpolating the forwards around the one we want to replace. To do so, we extract the values of the forwards rates and their corresponding dates.

```
In [15]: nodes = list(eonia_curve_ff.nodes())
```

If we look at the first few nodes, we can clearly see that the seventh is out of line.

```
In [16]: nodes[:9]

Out[16]: [(Date(11,12,2012), 0.00040555533025081675),
          (Date(12,12,2012), 0.00040555533025081675),
          (Date(13,12,2012), 0.00040555533047721286),
          (Date(14,12,2012), 0.00040555533047721286),
          (Date(20,12,2012), 0.0007604110692568178),
          (Date(27,12,2012), 0.0006894305026004767),
          (Date(3,1,2013), 0.0009732981324671213),
          (Date(14,1,2013), 0.0006728161005748453),
          (Date(13,2,2013), 0.0004663805459104 82)]
```

To create a curve that doesn't include the jump, we replace the relevant forward rate with a simple average of the ones that precede and follow…

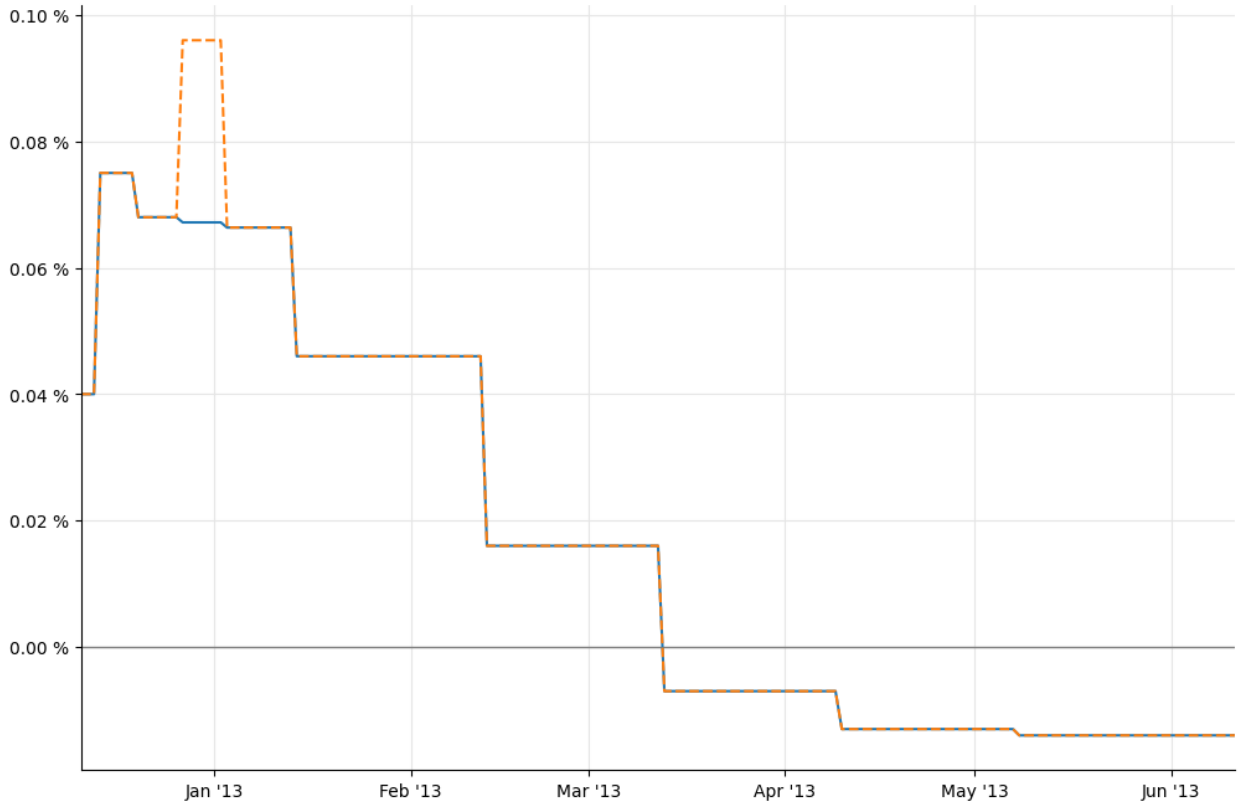
```
In [17]: nodes[6] = (nodes[6][0], (nodes[5][1]+nodes[7][1])/2.0)
         nodes[:9]

Out[17]: [(Date(11,12,2012), 0.00040555533025081675),
          (Date(12,12,2012), 0.00040555533025081675),
          (Date(13,12,2012), 0.00040555533047721286),
          (Date(14,12,2012), 0.00040555533047721286),
          (Date(20,12,2012), 0.0007604110692568178),
          (Date(27,12,2012), 0.0006894305026004767),
          (Date(3,1,2013), 0.000681123301587661),
          (Date(14,1,2013), 0.0006728161005748453),
          (Date(13,2,2013), 0.000466380545910482)]
```

…and instantiate a `ForwardCurve` with the modified nodes.

```
In [18]: temp_dates, temp_rates = zip(*nodes)
         temp_curve = ql.ForwardCurve(temp_dates, temp_rates,
                                      eonia_curve_ff.dayCounter())
```

For illustration, we can extract daily overnight nodes from the doctored curve and plot them alongside the old ones:

```
In [19]: temp_rates = [ temp_curve.forwardRate(d, ql.TARGET().advance(d,1,ql.Days),
                                                ql.Actual360(), ql.Simple).rate()
                        for d in dates ]

In [20]: _, ax = utils.plot()
         utils.highlight_x_axis(ax)
         utils.plot_curve(ax, dates, [(temp_rates,'-'), (rates_ff,'--')],
                          format_rates=True)
```



Now we can estimate the size of the jump. As the paper hints, it's more an art than a science. I've been able to reproduce closely the results of the paper by extracting from the two curves the forward rate over the two weeks around the end of the year:

```
In [21]: d1 = ql.Date(31,ql.December,2012) - ql.Period(1,ql.Weeks)
         d2 = ql.Date(31,ql.December,2012) + ql.Period(1,ql.Weeks)

In [22]: F = eonia_curve_ff.forwardRate(d1, d2, ql.Actual360(), ql.Simple).rate()
         F_1 = temp_curve.forwardRate(d1, d2, ql.Actual360(), ql.Simple).rate()
         print(utils.format_rate(F,digits=3))
         print(utils.format_rate(F_1,digits=3))

Out[22]: 0.082 %
         0.067 %
```

We want to attribute the whole jump to the last day of the year, so we rescale it according to

$$(F - F_1) \cdot t_{12} = J \cdot t_J$$

where t_{12} is the time between the two dates and t_J is the time between the start and end date of the end-of-year overnight deposit. This gives us a jump quite close to the value of 10.2 basis points reported in the paper.

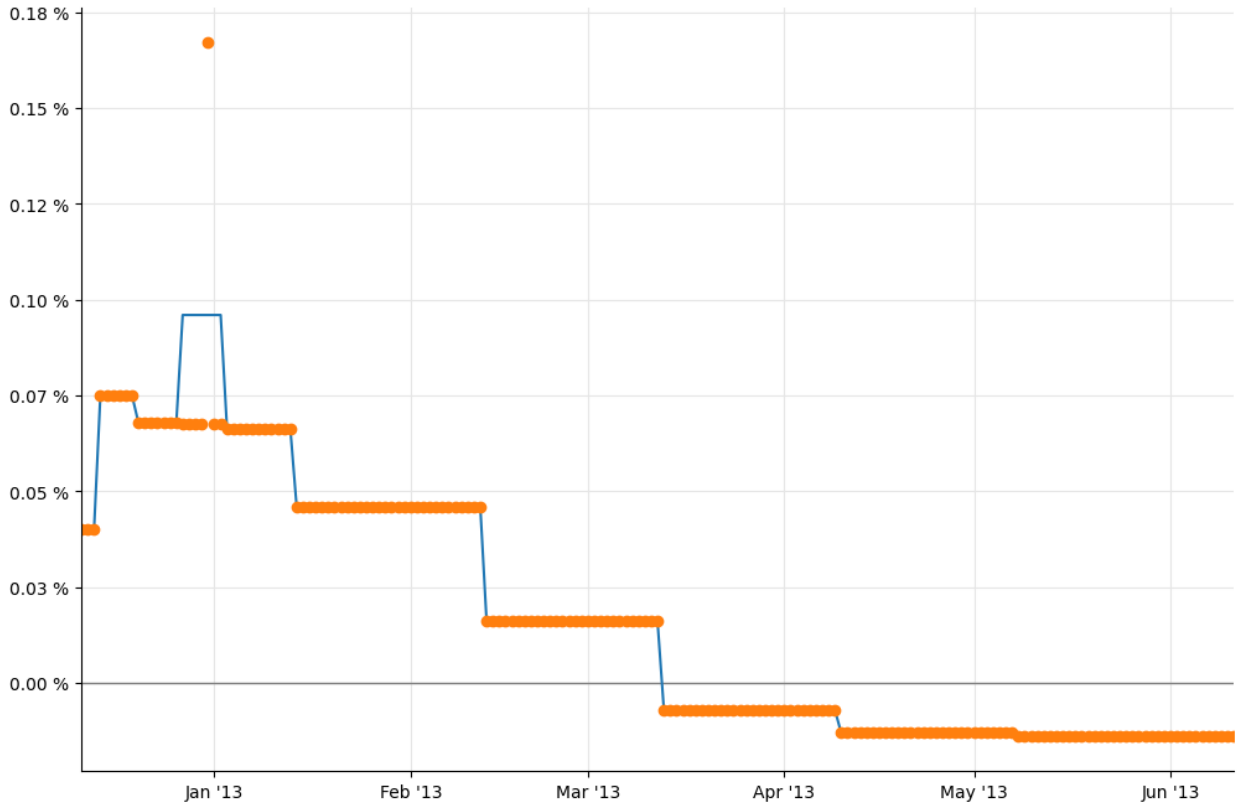
```
In [23]: t12 = eonia_curve_ff.dayCounter().yearFraction(d1,d2)
         t_j = eonia_curve_ff.dayCounter().yearFraction(ql.Date(31,ql.December,2012),
                                                        ql.Date(2,ql.January,2013))
         J = (F-F_1)*t12/t_j
         print(utils.format_rate(J,digits=3))

Out[23]: 0.101 %
```

As I mentioned previously, the jump can be added to the curve as a corresponding discount factor $1/(1 + J \cdot t_J)$ on the last day of the year. The information can be passed to the curve constructor, giving us a new instance:

```
In [24]: B = 1.0/(1.0+J*t_j)
         jumps = [ql.QuoteHandle(ql.SimpleQuote(B))]
         jump_dates = [ql.Date(31,ql.December,2012)]
         eonia_curve_j = ql.PiecewiseFlatForward(0, ql.TARGET(),
                                                 helpers, ql.Actual365Fixed(),
                                                 jumps, jump_dates)
```

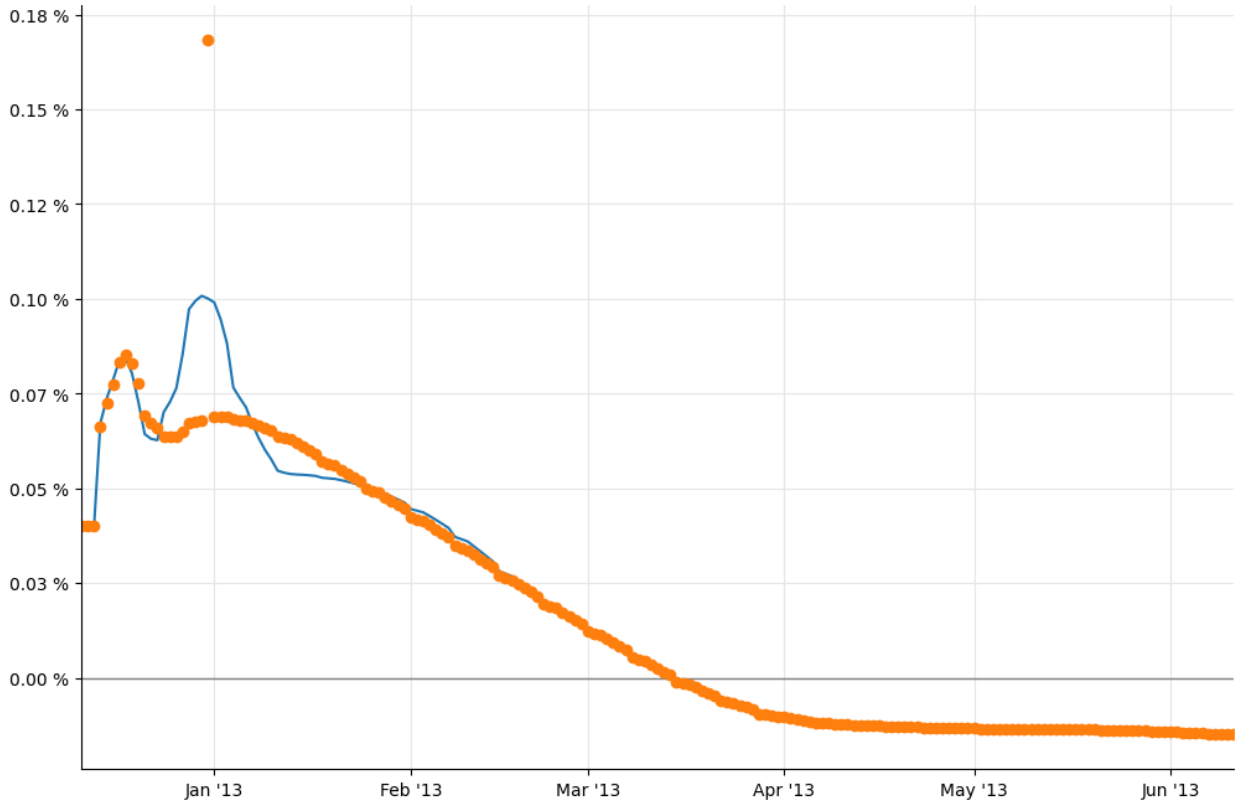
Retrieving daily overnight rates from the new curve and plotting them, we can see the jump quite clearly:

```
In [25]: rates_j = [ eonia_curve_j.forwardRate(d, ql.TARGET().advance(d,1,ql.Days),
                                               ql.Actual360(), ql.Simple).rate()
                     for d in dates ]

In [26]: _, ax = utils.plot()
         utils.highlight_x_axis(ax)
         utils.plot_curve(ax, dates, [(rates_ff,'-'), (rates_j,'o')],
                          format_rates=True)
```



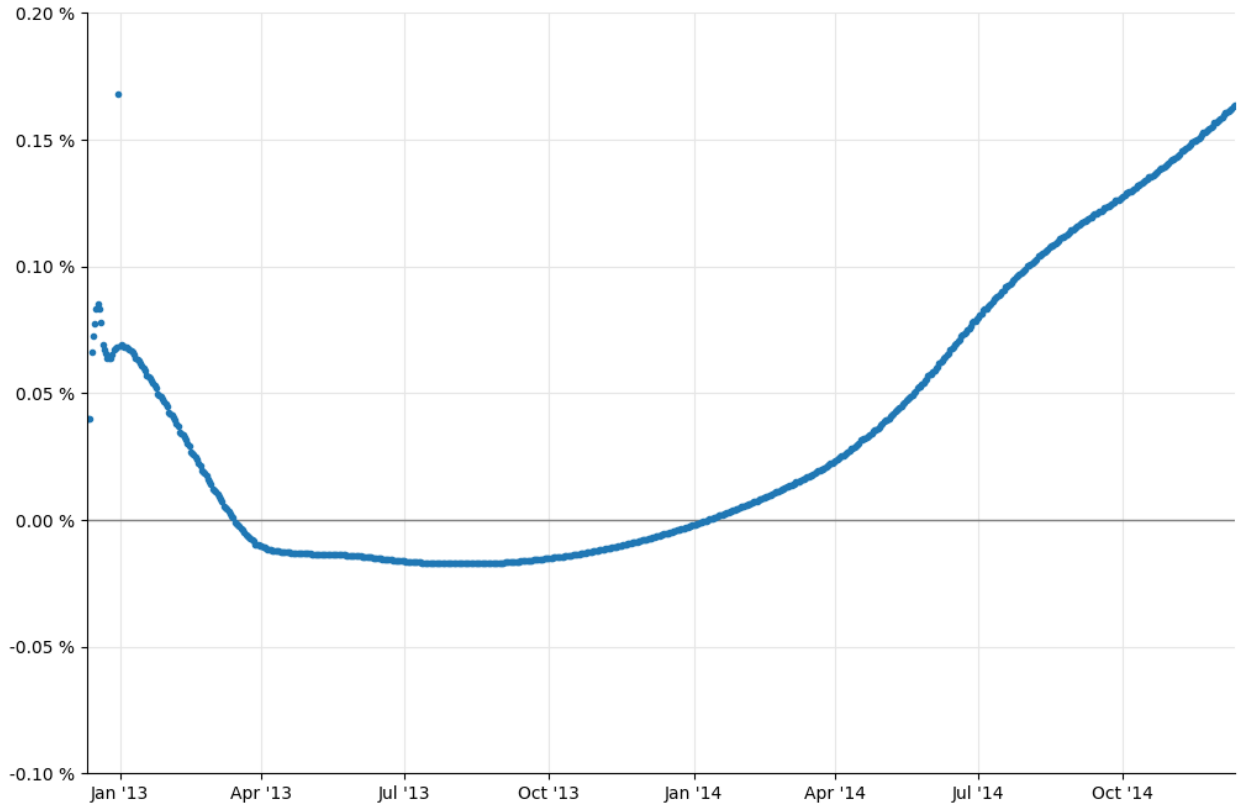
We can now go back to log-cubic discounts and add the jump.

```
In [27]: eonia_curve = ql.PiecewiseLogCubicDiscount(0, ql.TARGET(),
                                                    helpers, ql.Actual365Fixed(),
                                                    jumps, jump_dates)
         eonia_curve.enableExtrapolation()

In [28]: rates_c = [ eonia_curve_c.forwardRate(d, ql.TARGET().advance(d,1,ql.Days),
                                                ql.Actual360(), ql.Simple).rate()
                     for d in dates ]
         rates = [ eonia_curve.forwardRate(d, ql.TARGET().advance(d,1,ql.Days),
                                           ql.Actual360(), ql.Simple).rate()
                   for d in dates ]

In [29]: _, ax = utils.plot()
         utils.highlight_x_axis(ax)
         utils.plot_curve(ax, dates, [(rates_c,'-'), (rates,'o')],
                          format_rates=True)
```

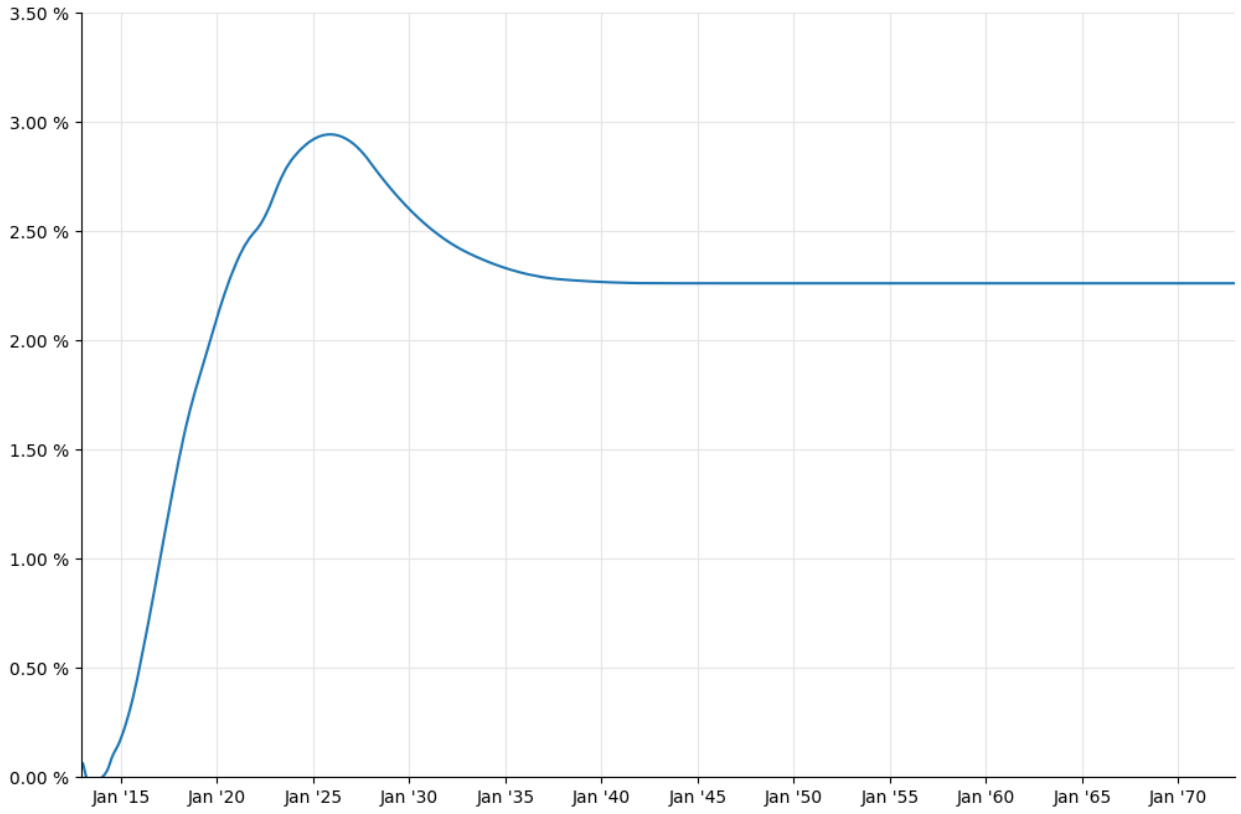


As you can see, the large bump is gone now. The two plots in figure 26 can be reproduced as follows (omitting the jump at the end of 2013 for brevity, and the flat forwards for clarity):

```
In [30]: dates = [ today+ql.Period(i,ql.Days) for i in range(0, 365*2+1) ]
         rates = [ eonia_curve.forwardRate(d, ql.TARGET().advance(d,1,ql.Days),
                                           ql.Actual360(), ql.Simple).rate()
                   for d in dates ]
         _, ax = utils.plot()
         utils.highlight_x_axis(ax)
         utils.plot_curve(ax, dates, [(rates,'.')], ymin=-0.001, ymax=0.002,
                          format_rates=True)
```




```
In [31]: dates = [ today+ql.Period(i,ql.Months) for i in range(0, 12*60+1) ]
         rates = [ eonia_curve.forwardRate(d, ql.TARGET().advance(d,1,ql.Days),
                                           ql.Actual360(), ql.Simple).rate()
                   for d in dates ]
         _, ax = utils.plot()
         utils.plot_curve(ax, dates, [(rates,'-')], ymin=0.0, ymax=0.035,
                          format_rates=True)
```



A final word of warning: as you saw, the estimate of the jumps is not an exact science, so it's best to check it manually and not to leave it to an automated procedure.

Moreover, jumps nowadays might be present at the end of each month, as reported for instance in Paolo Mazzocchi's presentation at the QuantLib User Meeting 2014[1]. This, too, suggests particular care in building the Eonia curve.

[1] https://speakerdeck.com/nando1970/eonia-jumps-and-proper-euribor-forwarding

4. Constructing a yield curve

In this chapter we will go over the construction of treasury yield curve. Let's start by importing QuantLib and other necessary libraries.

```
In [1]: import QuantLib as ql
        from pandas import DataFrame
        import numpy as np
        import utils
        %matplotlib inline
```

This is an example based on Exhibit 5-5 given in Frank Fabozzi's Bond Markets, Analysis and Strategies, Sixth Edition.

```
In [2]: depo_maturities = [ql.Period(6,ql.Months), ql.Period(12, ql.Months)]
        depo_rates = [5.25, 5.5]

        # Bond rates
        bond_maturities = [ql.Period(6*i, ql.Months) for i in range(3,21)]
        bond_rates = [5.75, 6.0, 6.25, 6.5, 6.75, 6.80, 7.00, 7.1, 7.15,
                      7.2, 7.3, 7.35, 7.4, 7.5, 7.6, 7.6, 7.7, 7.8]

        maturities = depo_maturities+bond_maturities
        rates = depo_rates+bond_rates
        DataFrame(list(zip(maturities, rates)),
                  columns=["Maturities","Curve"],
                  index=['']*len(rates))
```

Out[2]:

Maturities	Curve
6M	5.25
12M	5.50
18M	5.75
24M	6.00
30M	6.25
36M	6.50
42M	6.75
48M	6.80
54M	7.00
60M	7.10

Maturities	Curve
66M	7.15
72M	7.20
78M	7.30
84M	7.35
90M	7.40
96M	7.50
102M	7.60
108M	7.60
114M	7.70
120M	7.80

Below we declare some constants and conventions used here. For the sake of simplicity, we assume that some of the constants are the same for deposit rates and bond rates.

```
In [3]: calc_date = ql.Date(15, 1, 2015)
        ql.Settings.instance().evaluationDate = calc_date

        calendar = ql.UnitedStates(ql.UnitedStates.GovernmentBond)
        business_convention = ql.Unadjusted
        day_count = ql.Thirty360(ql.Thirty360.BondBasis)
        end_of_month = True
        settlement_days = 0
        face_amount = 100
        coupon_frequency = ql.Period(ql.Semiannual)
        settlement_days = 0
```

The basic idea of bootstrapping is to use the deposit rates and bond rates to create individual rate helpers. Then use the combination of the two helpers to construct the yield curve. As a first step, we create the deposit rate helpers as shown below.

```
In [4]: depo_helpers = [
            ql.DepositRateHelper(ql.QuoteHandle(ql.SimpleQuote(r/100.0)),
                                 m,
                                 settlement_days,
                                 calendar,
                                 business_convention,
                                 end_of_month,
                                 day_count)
            for r, m in zip(depo_rates, depo_maturities)
        ]
```

The rest of the points are coupon bonds. We assume that the YTM given for the bonds are all par rates. So we have bonds with coupon rate same as the YTM. Using this information, we construct the fixed rate bond helpers below.

```
In [5]: bond_helpers = []
        for r, m in zip(bond_rates, bond_maturities):
            termination_date = calc_date + m
            schedule = ql.Schedule(calc_date,
                                   termination_date,
                                   coupon_frequency,
                                   calendar,
                                   business_convention,
                                   business_convention,
                                   ql.DateGeneration.Backward,
                                   end_of_month)

            bond_helper = ql.FixedRateBondHelper(
                ql.QuoteHandle(ql.SimpleQuote(face_amount)),
                settlement_days,
                face_amount,
                schedule,
                [r/100.0],
                day_count,
                business_convention)
            bond_helpers.append(bond_helper)
```

The union of the two helpers is what we use in bootstrapping shown below.

```
In [6]: rate_helpers = depo_helpers + bond_helpers
```

The `get_spot_rates` is a convenient wrapper function that we will use to get the spot rates on a monthly interval.

```
In [7]: def get_spot_rates(
                yieldcurve, day_count,
                calendar=ql.UnitedStates(ql.UnitedStates.GovernmentBond),
                months=121
        ):
            spots = []
            tenors = []
            ref_date = yieldcurve.referenceDate()
            calc_date = ref_date
            for month in range(0, months):
                yrs = month/12.0
                d = calendar.advance(ref_date, ql.Period(month, ql.Months))
                compounding = ql.Compounded
                freq = ql.Semiannual
                zero_rate = yieldcurve.zeroRate(yrs, compounding, freq)
                tenors.append(yrs)
                eq_rate = zero_rate.equivalentRate(
```

```
                day_count,compounding,freq,calc_date,d).rate()
            spots.append(100*eq_rate)
        return DataFrame(list(zip(tenors, spots)),
                         columns=["Maturities","Curve"],
                         index=['']*len(tenors))
```

The bootstrapping process is fairly generic in QuantLib. You can chose what variable you are bootstrapping, and what is the interpolation method used in the bootstrapping. There are multiple piecewise interpolation methods that can be used for this process. The `PiecewiseLogCubicDiscount` will construct a piece wise yield curve using `LogCubic` interpolation of the `Discount` factor. Similarly `PiecewiseLinearZero` will use `Linear` interpolation of `Zero` rates. `PiecewiseCubicZero` will interpolate the `Zero` rates using a `Cubic` interpolation method.

```
In [8]: yc_logcubicdiscount = ql.PiecewiseLogCubicDiscount(calc_date,
                                                           rate_helpers,
                                                           day_count)
```

The zero rates from the tail end of the `PiecewiseLogCubicDiscount` bootstrapping is shown below.

```
In [9]: splcd = get_spot_rates(yc_logcubicdiscount, day_count)
        splcd.tail()
```

```
Out[9]:
```

	Maturities	Curve
	9.666667	7.981384
	9.750000	8.005292
	9.833333	8.028145
	9.916667	8.050187
	10.000000	8.071649

The yield curves using the `PiecewiseLinearZero` and `PiecewiseCubicZero` is shown below. The tail end of the zero rates obtained from `PiecewiseLinearZero` bootstrapping is also shown below. The numbers can be compared with that of the `PiecewiseLogCubicDiscount` shown above.

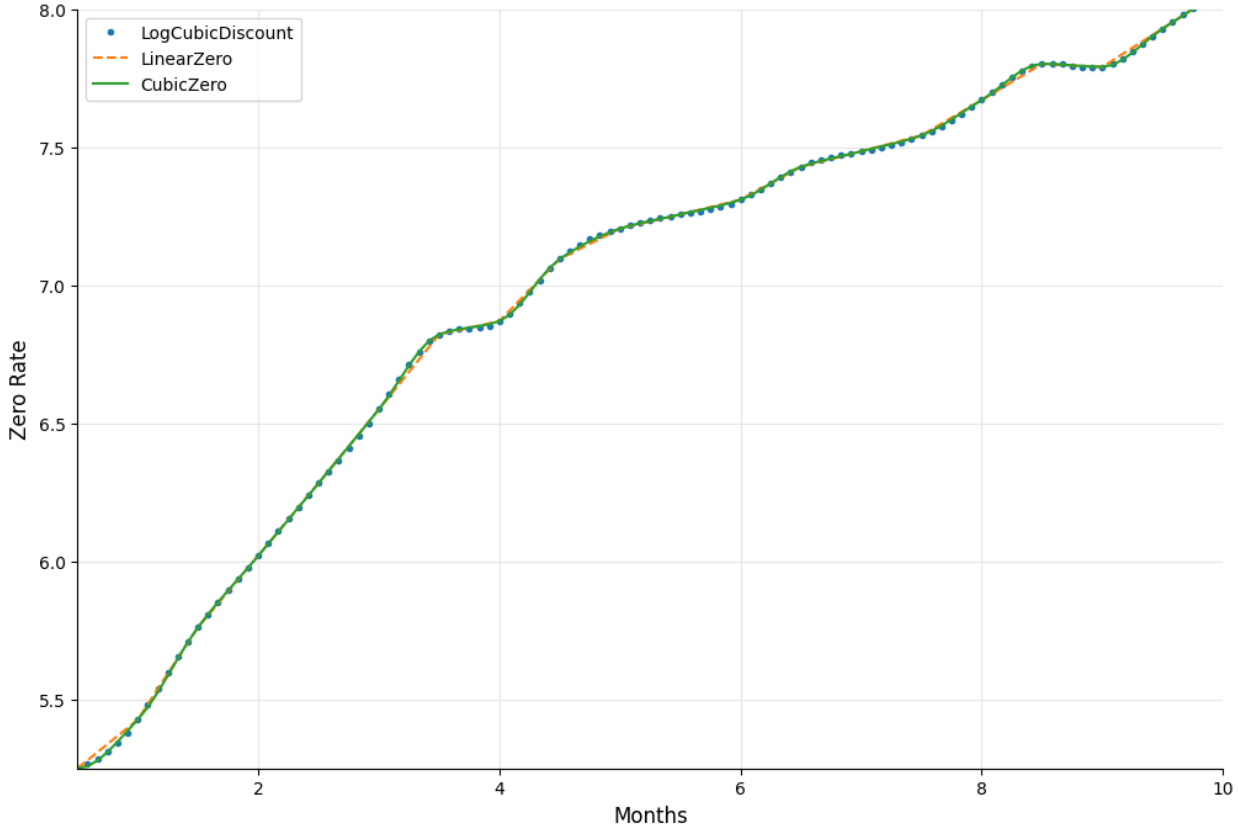
```
In [10]: yc_linearzero = ql.PiecewiseLinearZero(
             calc_date,rate_helpers,day_count
         )
         yc_cubiczero = ql.PiecewiseCubicZero(
             calc_date,rate_helpers,day_count
         )

         splz = get_spot_rates(yc_linearzero, day_count)
         spcz = get_spot_rates(yc_cubiczero, day_count)
         splz.tail()
```

Out[10]:

	Maturities	Curve
	9.666667	7.976804
	9.750000	8.000511
	9.833333	8.024221
	9.916667	8.047934
	10.000000	8.071649

All three are plotted below to give you an overall perspective of the three methods.

```
In [11]: fig, ax = utils.plot()
         ax.plot(splcd["Maturities"],splcd["Curve"], '.',
                 label="LogCubicDiscount")
         ax.plot(splz["Maturities"],splz["Curve"],'--',
                 label="LinearZero")
         ax.plot(spcz["Maturities"],spcz["Curve"],
                 label="CubicZero")
         ax.set_xlabel("Months", size=12)
         ax.set_ylabel("Zero Rate", size=12)
         ax.set_xlim(0.5,10)
         ax.set_ylim([5.25,8])
         ax.legend(loc=0);
```



Conclusion

In this chapter we saw how to construct yield curves by bootstrapping bond quotes.

5. Dangerous day-count conventions

(Based on a question by Min Gao on the QuantLib mailing list. Thanks!)

```
In [1]: import QuantLib as ql
```

```
In [2]: today = ql.Date(22,1,2018)
        ql.Settings.instance().evaluationDate = today
```

```
In [3]: %matplotlib inline
        import utils
```

The problem

Talking about term structures in *Implementing QuantLib*[1], I suggest to use simple day-count conventions such as Actual/360 or Actual/365 to initialize curves. That's because the convention is used internally to convert dates into times, and we want the conversion to be as regular as possible. For instance, we'd like distances between dates to be additive: given three dates d_1, d_2 and d_3, we would expect that $T(d_1, d_2) + T(d_2, d_3) = T(d_1, d_3)$, where T denotes the time between dates.

Unfortunately, that's not always the case for some day counters. The property holds for most dates…

```
In [4]: dc = ql.Thirty360(ql.Thirty360.USA)
```

```
In [5]: d1 = ql.Date(1, ql.January, 2018)
        d2 = ql.Date(15, ql.January, 2018)
        d3 = ql.Date(31, ql.January, 2018)
```

```
In [6]: print(dc.yearFraction(d1,d2) + dc.yearFraction(d2,d3))
        print(dc.yearFraction(d1,d3))
```

```
Out[6]: 0.08333333333333334
        0.08333333333333333
```

…but doesn't for some.

[1] https://leanpub.com/implementingquantlib

```
In [7]: d1 = ql.Date(1, ql.January, 2018)
        d2 = ql.Date(30, ql.January, 2018)
        d3 = ql.Date(31, ql.January, 2018)

In [8]: print(dc.yearFraction(d1,d2) + dc.yearFraction(d2,d3))
        print(dc.yearFraction(d1,d3))

Out[8]: 0.08055555555555556
        0.08333333333333333
```

That's because some day-count conventions were designed to calculate the duration of a coupon, not the distance between any two given dates. They have particular formulas and exceptions that make coupons more regular; but those exceptions also cause some pairs of dates to have strange properties. For instance, there might be no distance at all between some particular distinct dates:

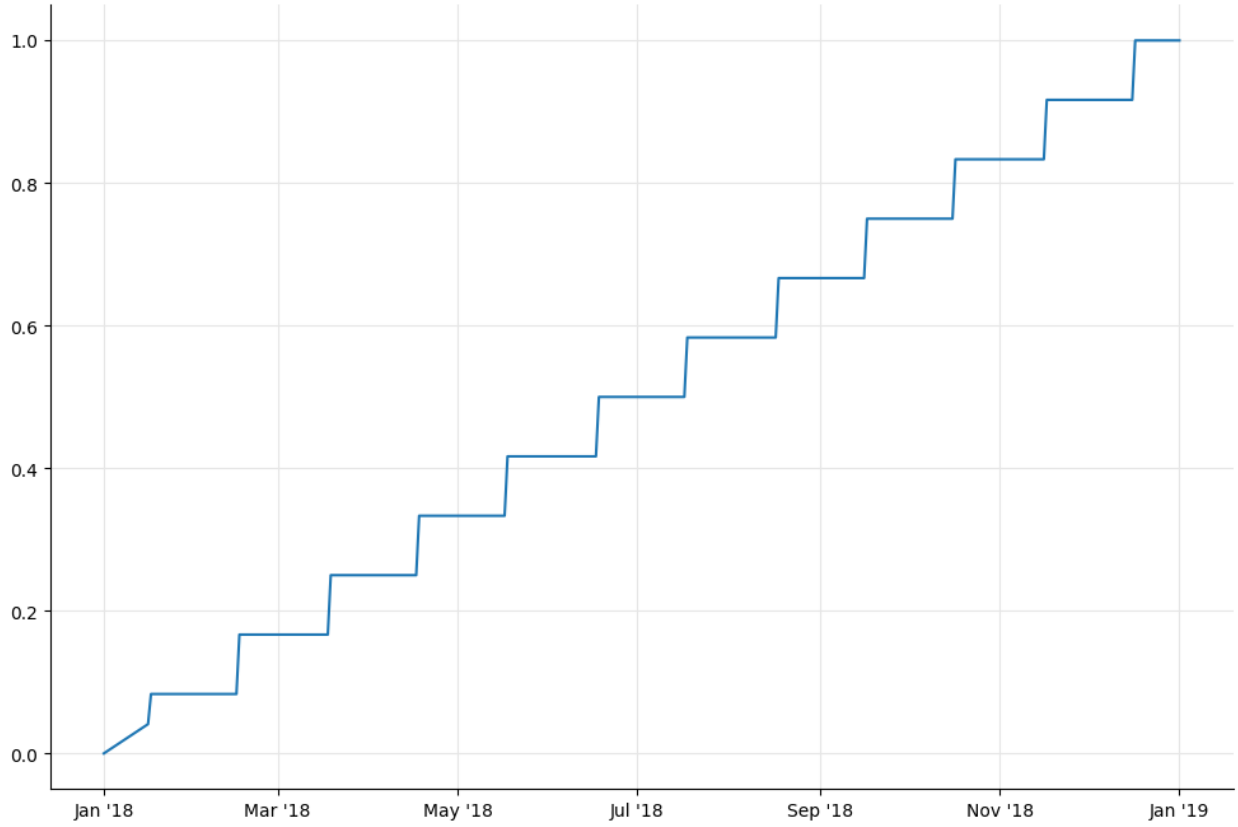
```
In [9]: d1 = ql.Date(30, ql.January, 2018)
        d2 = ql.Date(31, ql.January, 2018)

        print(dc.yearFraction(d1,d2))

Out[9]: 0.0
```

The 30/360 convention is not the worst offender, either. Min Gao's question came from using for the term structure the same convention used for the bond being priced, that is, ISMA actual/actual. This day counter is supposed to be given a reference period, as well as the two dates whose distance one needs to measure; failing to do so will result in the wrong results…

```
In [10]: d1 = ql.Date(1, ql.January, 2018)
         d2 = ql.Date(15, ql.January, 2018)

         reference_period = (ql.Date(1, ql.January, 2018), ql.Date(1, ql.July, 2018))

In [11]: dc = ql.ActualActual(ql.ActualActual.ISMA)

         print(dc.yearFraction(d1, d2, *reference_period))
         print(dc.yearFraction(d1, d2))

Out[11]: 0.03867403314917127
         0.038356164383561646
```

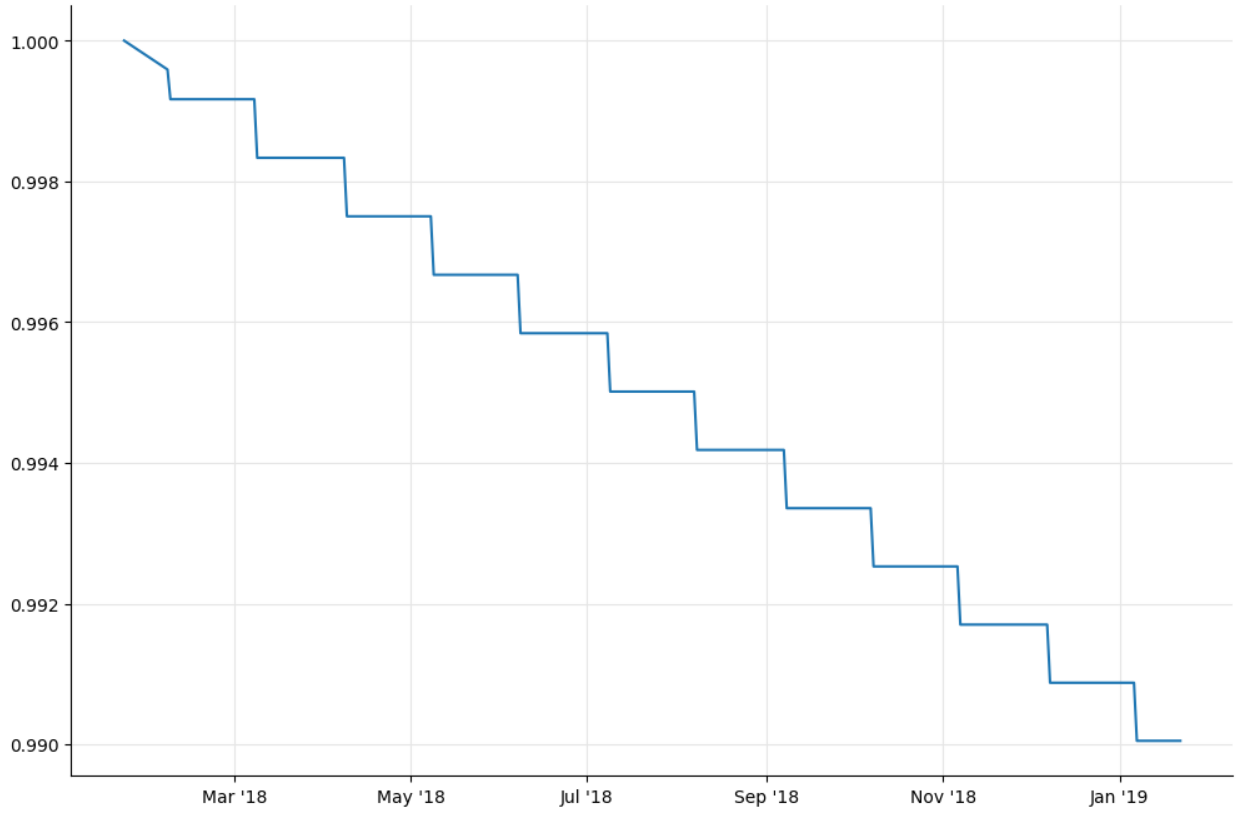
…and sometimes, in spectacularly wrong results. Here is what happens if we plot the year fraction since January 1st, 2018 as a function of the date over that same year.

```
In [12]: d1 = ql.Date(1, ql.January, 2018)
         dates = [ (d1 + i) for i in range(366) ]
         times = [ dc.yearFraction(d1, d) for d in dates ]

In [13]: fig, ax = utils.plot()
         ax.xaxis.set_major_formatter(utils.date_formatter())
         ax.plot_date([ utils.to_datetime(d) for d in dates ], times,'-');
```



Of course, that's no way to convert dates into times. Using this day-count convention inside a coupon is ok, of course. Using it inside a term structure, which doesn't have any concept of a reference period, leads to very strange behaviors.

```
In [14]: curve = ql.FlatForward(today, 0.01, ql.ActualActual(ql.ActualActual.ISMA))

In [15]: dates = [ (today + i) for i in range(366) ]
         discounts = [ curve.discount(d) for d in dates ]
         fig, ax = utils.plot()
         ax.xaxis.set_major_formatter(utils.date_formatter())
         ax.plot_date([ utils.to_datetime(d) for d in dates ], discounts,'-');
```



Any solutions?

Not really, at this time. Work is underway to store a schedule inside an ISMA actual/actual day counter and use it to retrieve the correct reference period, but that's not fully working yet. In the meantime, what I can suggest is to use the specified day-count conventions for coupons; but, unless something prevents it, use a simple day-count convention such as actual/360 or actual/365 for term structures.

6. Valuing European and American options

I have written about option pricing earlier. The introduction to option pricing[1] gave an overview of the theory behind option pricing. The post on introduction to binomial trees[2] outlined the binomial tree method to price options.

In this post, we will use QuantLib and the Python extension to illustrate a simple example. Here we are going to price a European option using the Black-Scholes-Merton formula. We will price them again using the Binomial tree and understand the agreement between the two.

```
In [1]: import QuantLib as ql
        import utils
        %matplotlib inline
```

European Option

Let us consider a European call option for AAPL with a strike price of 130 maturing on 15th Jan, 2016. Let the spot price be 127.62. The volatility of the underlying stock is know to be 20%, and has a dividend yield of 1.63%. Let's value this option as of 8th May, 2015.

```
In [2]: maturity_date = ql.Date(15, 1, 2016)
        spot_price = 127.62
        strike_price = 130
        volatility = 0.20 # the historical vols for a year
        dividend_rate =  0.0163
        option_type = ql.Option.Call

        risk_free_rate = 0.001
        day_count = ql.Actual365Fixed()
        calendar = ql.UnitedStates(ql.UnitedStates.GovernmentBond)

        calculation_date = ql.Date(8, 5, 2015)
        ql.Settings.instance().evaluationDate = calculation_date
```

We construct the European option here.

[1] http://gouthamanbalaraman.com/blog/option-model-handbook-part-I-introduction-to-option-models.html
[2] http://gouthamanbalaraman.com/blog/option-model-handbook-part-II-introduction-to-binomial-trees.html

```
In [3]: payoff = ql.PlainVanillaPayoff(option_type, strike_price)
        exercise = ql.EuropeanExercise(maturity_date)
        european_option = ql.VanillaOption(payoff, exercise)
```

The Black-Scholes-Merton process is constructed here.

```
In [4]: spot_handle = ql.QuoteHandle(
            ql.SimpleQuote(spot_price)
        )
        flat_ts = ql.YieldTermStructureHandle(
            ql.FlatForward(calculation_date,
                           risk_free_rate,
                           day_count)
        )
        dividend_yield = ql.YieldTermStructureHandle(
            ql.FlatForward(calculation_date,
                           dividend_rate,
                           day_count)
        )
        flat_vol_ts = ql.BlackVolTermStructureHandle(
            ql.BlackConstantVol(calculation_date,
                                calendar,
                                volatility,
                                day_count)
        )
        bsm_process = ql.BlackScholesMertonProcess(spot_handle,
                                                   dividend_yield,
                                                   flat_ts,
                                                   flat_vol_ts)
```

Lets compute the theoretical price using the `AnalyticEuropeanEngine`.

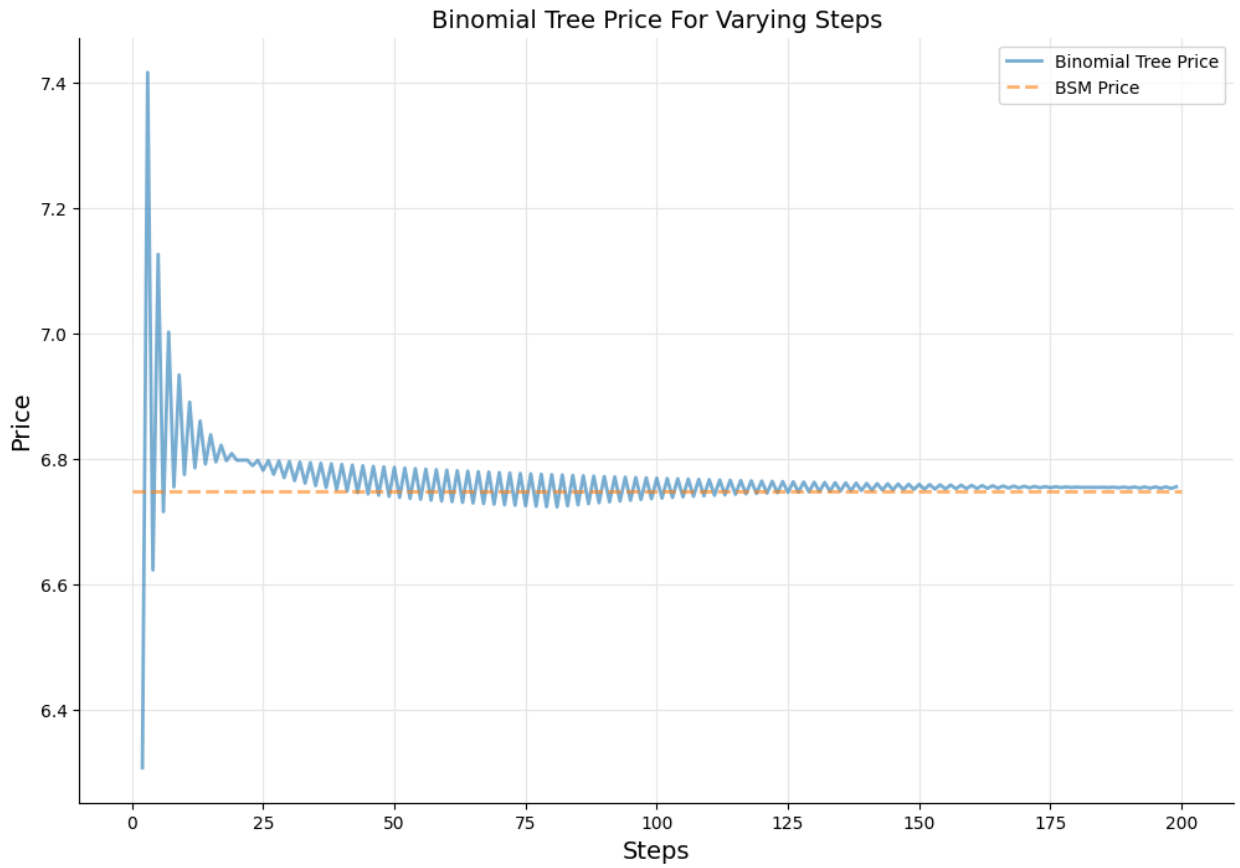
```
In [5]: european_option.setPricingEngine(ql.AnalyticEuropeanEngine(bsm_process))
        bs_price = european_option.NPV()
        print("The theoretical price is %lf" % bs_price)

Out[5]: The theoretical price is 6.749272
```

Lets compute the price using the binomial-tree approach.

```
In [6]: def binomial_price(option, bsm_process, steps):
            binomial_engine = ql.BinomialVanillaEngine(bsm_process, "crr", steps)
            option.setPricingEngine(binomial_engine)
            return option.NPV()

        steps = range(2, 200, 1)
        prices = [binomial_price(european_option, bsm_process, step) for step in step\
s]
```

In the plot below, we show the convergence of binomial-tree approach by comparing its price with the BSM price.

```
In [7]: fig, ax = utils.plot()
        ax.plot(steps, prices, label="Binomial Tree Price", lw=2, alpha=0.6)
        ax.plot([0,200],[bs_price, bs_price], "--", label="BSM Price", lw=2, alpha=0.\
6)
        ax.set_xlabel("Steps", size=14)
        ax.set_ylabel("Price", size=14)
        ax.set_title("Binomial Tree Price For Varying Steps", size=14)
        ax.legend();
```



American Option

The above exercise was pedagogical, and introduces one to pricing using the binomial tree approach and compared with Black-Scholes. As a next step, we will use the Binomial pricing to value American options.

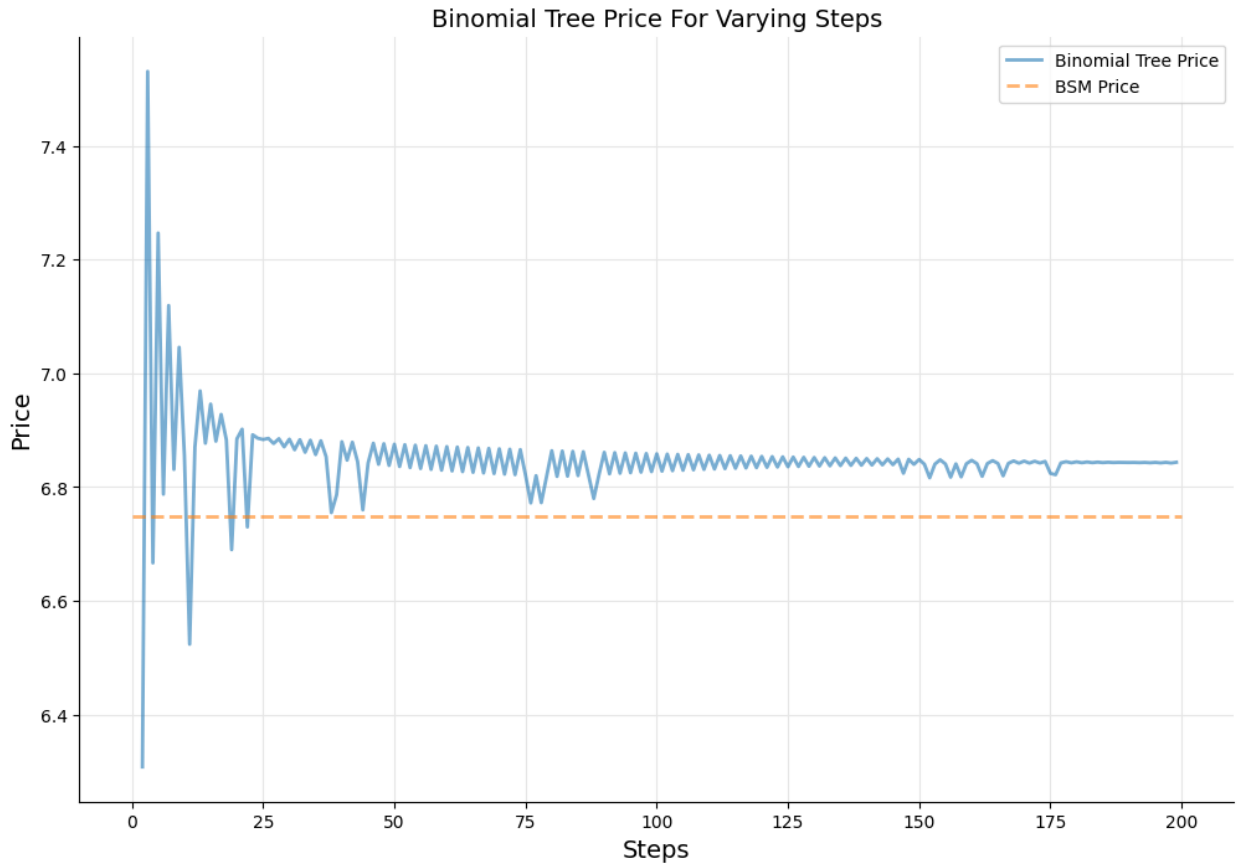
The construction of an American option is similar to the construction of `European` option discussed above. The one main difference is the use of `AmericanExercise` instead of `EuropeanExercise` use above.

```
In [8]: payoff = ql.PlainVanillaPayoff(option_type, strike_price)
        settlement = calculation_date
        am_exercise = ql.AmericanExercise(settlement, maturity_date)
        american_option = ql.VanillaOption(payoff, am_exercise)
```

Once we have constructed the `american_option` object, we can price them using the Binomial trees as done above. We use the same function we constructed above.

```
In [9]: steps = range(2, 200, 1)
        prices = [binomial_price(american_option, bsm_process, step) for step in step\
s]
```

```
In [10]: fig, ax = utils.plot()
         ax.plot(steps, prices, label="Binomial Tree Price", lw=2, alpha=0.6)
         ax.plot([0,200],[bs_price, bs_price], "--", label="BSM Price", lw=2, alpha=0\
.6)
         ax.set_xlabel("Steps", size=14)
         ax.set_ylabel("Price", size=14)
         ax.set_title("Binomial Tree Price For Varying Steps", size=14)
         ax.legend();
```



Above, we plot the price of the American option as a function of steps used in the binomial tree, and compare with that of the Black-Scholes price for the European option with all other variables remaining the same. The binomial tree converges as the number of steps used in pricing increases. American option is valued more than the European BSM price because of the fact that it can be exercised anytime during the course of the option.

Conclusion

In this chapter we learnt about valuing European and American options using the binomial tree method.

7. Duration of floating-rate bonds

(Based on a question by Antonio Savoldi on the QuantLib mailing list. Thanks!)

```
In [1]: import QuantLib as ql
        from pandas import DataFrame

In [2]: today = ql.Date(8,ql.October,2014)
        ql.Settings.instance().evaluationDate = today
```

The problem

We want to calculate the modified duration of a floating-rate bond. First, we need an interest-rate curve to forecast its coupon rates: for illustration's sake, let's take a flat curve with a 0.2% rate.

```
In [3]: forecast_curve = ql.RelinkableYieldTermStructureHandle()
        forecast_curve.linkTo(ql.FlatForward(today, 0.002, ql.Actual360(),
                                             ql.Compounded, ql.Semiannual))
```

Then, we instantiate the index to be used. The bond has semiannual coupons, so we create a `Euribor6M` instance and we pass it the forecast curve. Also, we set a past fixing for the current coupon (which, having fixed in the past, can't be forecast).

```
In [4]: index = ql.Euribor6M(forecast_curve)
        index.addFixing(ql.Date(6,ql.August,2014), 0.002)
```

The bond was issued a couple of months before the evaluation date and will run for 5 years with semiannual coupons.

```
In [5]: issueDate = ql.Date(8,ql.August,2014)
        maturityDate = ql.Date(8,ql.August,2019)

        schedule = ql.Schedule(issueDate, maturityDate,
                               ql.Period(ql.Semiannual), ql.TARGET(),
                               ql.Following, ql.Following,
                               ql.DateGeneration.Backward, False)

        bond = ql.FloatingRateBond(settlementDays = 3,
                                   faceAmount = 100,
                                   schedule = schedule,
                                   index = index,
                                   paymentDayCounter = ql.Actual360())
```

The cash flows are calculated based on the forecast curve. Here they are, together with their dates. As expected, they each pay around 0.1% of the notional.

```
In [6]: dates = [ c.date() for c in bond.cashflows() ]
        cfs = [ c.amount() for c in bond.cashflows() ]
        DataFrame(list(zip(dates, cfs)),
                  columns = ('date','amount'),
                  index = range(1,len(dates)+1))
```

Out[6]:

	date	amount
1	February 9th, 2015	0.102778
2	August 10th, 2015	0.101112
3	February 8th, 2016	0.101112
4	August 8th, 2016	0.101112
5	February 8th, 2017	0.102223
6	August 8th, 2017	0.100556
7	February 8th, 2018	0.102223
8	August 8th, 2018	0.100556
9	February 8th, 2019	0.102223
10	August 8th, 2019	0.100556
11	August 8th, 2019	100.000000

If we try to use the function provided for calculating bond durations, though, we run into a problem. When we pass it the bond and a 0.2% semiannual yield, the result we get is:

```
In [7]: y = ql.InterestRate(0.002, ql.Actual360(), ql.Compounded, ql.Semiannual)
        print(ql.BondFunctions.duration(bond, y, ql.Duration.Modified))
```

```
Out[7]: 4.8609591731332165
```

which is about the time to maturity. Shouldn't we get the time to next coupon instead?

What happened?

The function above is too generic. It calculates the modified duration as $-\frac{1}{P}\frac{dP}{dy}$; however, it doesn't know what kind of bond it has been passed and what kind of cash flows are paid, so it can only consider the yield for discounting and not for forecasting. If you looked into the C++ code, you'd see that the bond price P above is calculated as the sum of the discounted cash flows, as in the following:

```
In [8]: y = ql.SimpleQuote(0.002)
        yield_curve = ql.FlatForward(bond.settlementDate(), ql.QuoteHandle(y),
                                     ql.Actual360(), ql.Compounded, ql.Semiannual)

        dates = [ c.date() for c in bond.cashflows() ]
        cfs = [ c.amount() for c in bond.cashflows() ]
        discounts = [ yield_curve.discount(d) for d in dates ]
        P = sum(cf*b for cf,b in zip(cfs,discounts))

        print(P)

Out[8]: 100.03665363580889
```

(Incidentally, we can see that this matches the calculation in the `dirtyPrice` method of the `Bond` class.)

```
In [9]: bond.setPricingEngine(
            ql.DiscountingBondEngine(ql.YieldTermStructureHandle(yield_curve)))
        print(bond.dirtyPrice())

Out[9]: 100.03665363580889
```

Finally, the derivative $\frac{dP}{dy}$ in the duration formula in approximated as $\frac{P(y+dy) - P(y-dy)}{2dy}$, so that we get:

```
In [10]: dy = 1e-5

         y.setValue(0.002 + dy)
         cfs_p = [ c.amount() for c in bond.cashflows() ]
         discounts_p = [ yield_curve.discount(d) for d in dates ]
         P_p = sum(cf*b for cf,b in zip(cfs_p,discounts_p))
         print(P_p)

         y.setValue(0.002 - dy)
         cfs_m = [ c.amount() for c in bond.cashflows() ]
         discounts_m = [ yield_curve.discount(d) for d in dates ]
         P_m = sum(cf*b for cf,b in zip(cfs_m,discounts_m))
         print(P_m)

         y.setValue(0.002)

Out[10]: 100.03179102561501
         100.0415165074028

In [11]: print(-(1/P)*(P_p - P_m)/(2*dy))

Out[11]: 4.8609591756253225
```

which is the same figure returned by `BondFunctions.duration`.

The problem is that the above doesn't use the yield curve for forecasting, so it's not really considering the bond as a floating-rate bond. It's using it as a fixed-rate bond, whose coupon rates happen to equal the current forecasts for the Euribor 6M fixings. This is clear if we look at the coupon amounts and discounts we stored during the calculation:

```
In [12]: DataFrame(list(zip(dates, cfs, discounts,
                             cfs_p, discounts_p, cfs_m, discounts_m)),
                   columns = ('date','amount','discounts',
                              'amount (+)','discounts (+)',
                              'amount (-)','discounts (-)',),
                   index = range(1,len(dates)+1))
```

Out[12]:

	date	amount	discounts	amount (+)	discounts (+)	amount (-)	discounts (-)
1	February 9th, 2015	0.102778	0.999339	0.102778	0.999336	0.102778	0.999343
2	August 10th, 2015	0.101112	0.998330	0.101112	0.998322	0.101112	0.998338
3	February 8th, 2016	0.101112	0.997322	0.101112	0.997308	0.101112	0.997335
4	August 8th, 2016	0.101112	0.996314	0.101112	0.996296	0.101112	0.996333
5	February 8th, 2017	0.102223	0.995297	0.102223	0.995273	0.102223	0.995320
6	August 8th, 2017	0.100556	0.994297	0.100556	0.994269	0.100556	0.994325
7	February 8th, 2018	0.102223	0.993282	0.102223	0.993248	0.102223	0.993315
8	August 8th, 2018	0.100556	0.992284	0.100556	0.992245	0.100556	0.992322
9	February 8th, 2019	0.102223	0.991270	0.102223	0.991227	0.102223	0.991314
10	August 8th, 2019	0.100556	0.990275	0.100556	0.990226	0.100556	0.990323
11	August 8th, 2019	100.000000	0.990275	100.000000	0.990226	100.000000	0.990323

where you can see how the discount factors changed when the yield was modified, but the coupon amounts stayed the same.

The solution

Unfortunately, there's no easy way to fix the `BondFunctions.duration` method so that it does the right thing. What we can do, instead, is to repeat the calculation above while setting up the bond and the curves so that the yield is used correctly. In particular, we have to link the forecast curve to the flat yield curve being modified…

```
In [13]: forecast_curve.linkTo(yield_curve)
```

...so that changing the yield will also affect the forecast rate of the coupons.

```
In [14]: y.setValue(0.002 + dy)
         P_p = bond.dirtyPrice()
         cfs_p = [ c.amount() for c in bond.cashflows() ]
         discounts_p = [ yield_curve.discount(d) for d in dates ]
         print(P_p)

         y.setValue(0.002 - dy)
         P_m = bond.dirtyPrice()
         cfs_m = [ c.amount() for c in bond.cashflows() ]
         discounts_m = [ yield_curve.discount(d) for d in dates ]
         print(P_m)

         y.setValue(0.002)

Out[14]: 100.03632329080955
         100.03698398354918
```

Now the coupon amounts change with the yield (except, of course, the first coupon, whose amount was already fixed)...

```
In [15]: DataFrame(list(zip(dates, cfs, discounts, cfs_p,
                             discounts_p, cfs_m, discounts_m)),
                  columns = ('date','amount','discounts',
                             'amount (+)','discounts (+)',
                             'amount (-)','discounts (-)',),
                  index = range(1,len(dates)+1))

Out[15]:
```

	date	amount	discounts	amount (+)	discounts (+)	amount (-)	discounts (-)
1	February 9th, 2015	0.102778	0.999339	0.102778	0.999336	0.102778	0.999343
2	August 10th, 2015	0.101112	0.998330	0.101617	0.998322	0.100606	0.998338
3	February 8th, 2016	0.101112	0.997322	0.101617	0.997308	0.100606	0.997335
4	August 8th, 2016	0.101112	0.996314	0.101617	0.996296	0.100606	0.996333
5	February 8th, 2017	0.102223	0.995297	0.102734	0.995273	0.101712	0.995320
6	August 8th, 2017	0.100556	0.994297	0.101059	0.994269	0.100053	0.994325
7	February 8th, 2018	0.102223	0.993282	0.102734	0.993248	0.101712	0.993315

	date	amount	discounts	amount (+)	discounts (+)	amount (-)	discounts (-)
8	August 8th, 2018	0.100556	0.992284	0.101059	0.992245	0.100053	0.992322
9	February 8th, 2019	0.102223	0.991270	0.102734	0.991227	0.101712	0.991314
10	August 8th, 2019	0.100556	0.990275	0.101059	0.990226	0.100053	0.990323
11	August 8th, 2019	100.000000	0.990275	100.000000	0.990226	100.000000	0.990323

...and the duration is calculated correctly, thus approximating the four months to the next coupon.

```
In [16]: print(-(1/P)*(P_p - P_m)/(2*dy))
```

```
Out[16]: 0.33022533022465994
```

This also holds if the discounting curve is dependent, but not the same as the forecast curve; e.g., as in the case of an added credit spread:

```
In [17]: discount_curve = ql.ZeroSpreadedTermStructure(
             forecast_curve,
             ql.QuoteHandle(ql.SimpleQuote(0.001)))
         bond.setPricingEngine(
             ql.DiscountingBondEngine(ql.YieldTermStructureHandle(discount_curve)))
```

This causes the price to decrease due to the increased discount factors...

```
In [18]: P = bond.dirtyPrice()
         cfs = [ c.amount() for c in bond.cashflows() ]
         discounts = [ discount_curve.discount(d) for d in dates ]
         print(P)
```

```
Out[18]: 99.55107926688962
```

...but the coupon amounts are still the same.

```
In [19]: DataFrame(list(zip(dates, cfs, discounts)),
                   columns = ('date','amount','discount'),
                   index = range(1,len(dates)+1))
```

```
Out[19]:
```

	date	amount	discount
1	February 9th, 2015	0.102778	0.999009
2	August 10th, 2015	0.101112	0.997496
3	February 8th, 2016	0.101112	0.995984
4	August 8th, 2016	0.101112	0.994475
5	February 8th, 2017	0.102223	0.992952
6	August 8th, 2017	0.100556	0.991456
7	February 8th, 2018	0.102223	0.989938
8	August 8th, 2018	0.100556	0.988446
9	February 8th, 2019	0.102223	0.986932
10	August 8th, 2019	0.100556	0.985445
11	August 8th, 2019	100.000000	0.985445

The price derivative is calculated in the same way as above…

```
In [20]: y.setValue(0.002 + dy)
         P_p = bond.dirtyPrice()
         print(P_p)

         y.setValue(0.002 - dy)
         P_m = bond.dirtyPrice()
         print(P_m)

         y.setValue(0.002)

Out[20]: 99.55075966035385
         99.55139887578544

In [21]: print(-(1/P)*(P_p - P_m)/(2*dy))

Out[21]: 0.3210489711903113
```

…and yields a similar result.

Translating QuantLib Python examples to C++

It's easy enough to translate the Python code shown in this book into the corresponding C++ code. As an example, I'll go through a bit of code from the notebook on instruments and pricing engines.

```
In [1]: import QuantLib as ql
```

This line imports the `QuantLib` module and provides a shorter alias that can be used to qualify the classes and functions it contains. The C++ equivalent would be:

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

namespace ql = QuantLib;
```

In C++, however, I wouldn't include the global `quantlib.hpp` header, which would inflate your compilation times; instead, you can include the specific headers for the classes you'll use.

Moreover, in C++ is not as discouraged as in Python to import the whole contents of a namespace in a source file, that is, to use

```
using namespace QuantLib;
```

However, the above should not be used in header files; ask the nearest C++ guru if you're not sure why.

```
In [2]: today = ql.Date(7, ql.March, 2014)
        ql.Settings.instance().evaluationDate = today
```

The code above has a couple of caveats. The first line is easy enough to translate; you'll have to declare the type to the variable (or use `auto` if you're compiling in C++11 mode). The second line is trickier. To begin with, the syntax to call static methods differs in Python and C++, so you'll have to replace the dot before `instance` by a double colon (the same goes for the namespace qualifications). Then, `evaluationDate` is a property in Python but a method in C++; it was changed in the Python module to be more idiomatic, since it's not that usual in Python to assign to the result of a method. Luckily, you won't find many such cases. The translated code is:

```
ql::Date today(7, ql::March, 2014);
ql::Settings::instance().evaluationDate() = today;
```

Next:

```
In [3]: option = ql.EuropeanOption(ql.PlainVanillaPayoff(ql.Option.Call, 100.0),
                                   ql.EuropeanExercise(ql.Date(7, ql.June, 2014)))
```

Again, you'll have to declare the type of the variable. Furthermore, the constructor of `EuropeanOption` takes its arguments by pointer, or more precisely, by `boost::shared_ptr`. This is hidden in Python, since there's no concept of pointer in the language; the SWIG wrappers take care of exporting `boost::shared_ptr<T>` simply as `T`. The corresponding C++ code (note also the double colon in `Option::Call`):

```
ql::EuropeanOption option(
    boost::make_shared<ql::PlainVanillaPayoff>(ql::Option::Call, 100.0),
    boost::make_shared<ql::EuropeanExercise(ql::Date(7, ql::June, 2014)));
```

(A note: in the remainder of the example, I'll omit the `boost::` and `ql::` namespaces for brevity.)

```
In [4]: u = ql.SimpleQuote(100.0)
        r = ql.SimpleQuote(0.01)
        sigma = ql.SimpleQuote(0.20)
```

Quotes, too, are stored and passed around as `shared_ptr` instances; this is the case for most polymorphic classes (when in doubt, you can look at the C++ headers and check the signatures of the functions you want to call). The above becomes:

```
shared_ptr<SimpleQuote> u = make_shared<SimpleQuote>(100.0);
shared_ptr<SimpleQuote> r = make_shared<SimpleQuote>(0.01);
shared_ptr<SimpleQuote> sigma = make_shared<SimpleQuote>(0.20);
```

Depending on what you need to do with them, the variables might also be declared as `shared_ptr<Quote>`. I used the above, since I'll need to call a method of `SimpleQuote` in a later part of the code.

```
In [5]: riskFreeCurve = ql.FlatForward(0, ql.TARGET(),
                                       ql.QuoteHandle(r), ql.Actual360())
        volatility = ql.BlackConstantVol(0, ql.TARGET(),
                                         ql.QuoteHandle(sigma), ql.Actual360())
```

The `Handle` template class couldn't be exported as such, because Python doesn't have templates. Thus, the SWIG wrappers have to declare separate classes `QuoteHandle`, `YieldTermStructureHandle` and so on. In C++, you can go back to the original syntax.

```
shared_ptr<YieldTermStructure> riskFreeCurve =
    make_shared<FlatForward>(0, TARGET(),
                             Handle<Quote>(r), Actual360());
shared_ptr<BlackVolTermStructure> volatility =
    make_shared<BlackConstantVol>(0, TARGET(),
                                  Handle<Quote>(sigma), Actual360());
```

Next,

```
In [6]: process = ql.BlackScholesProcess(ql.QuoteHandle(u),
                                          ql.YieldTermStructureHandle(riskFreeCurve),
                                          ql.BlackVolTermStructureHandle(volatility))
```

turns into

```
shared_ptr<BlackScholesProcess> process =
    make_shared<BlackConstantVol>(
        Handle<Quote>(u),
        Handle<YieldTermStructure>(riskFreeCurve),
        Handle<BlackVolTermStructure>(volatility));
```

and

```
In [7]: engine = ql.AnalyticEuropeanEngine(process)
```

into

```
shared_ptr<PricingEngine> engine =
    make_shared<AnalyticEuropeanEngine>(process);
```

So far, we've been calling constructors. Method invocation works the same in Python and C++, except that in C++ we might be calling methods through a pointer. Therefore,

```
In [8]: option.setPricingEngine(engine)

In [9]: print(option.NPV())

In [10]: print(option.delta())
         print(option.gamma())
         print(option.vega())
```

where `option` is an object in C++, becomes

```
option.setPricingEngine(engine);

cout << option.NPV() << endl;

cout << option.delta() << endl;
cout << option.gamma() << endl;
cout << option.vega() << endl;
```

whereas

```
In [11]: u.setValue(105.0)
```

since `u` is a (smart) pointer, turns into

```
u->setValue(105.0);
```

Of course, the direct translation I've been doing only applies to the QuantLib code; I'm not able to point you to libraries that replace the graphing functionality in `matplotlib`, or the data-analysis facilities in `pandas`, or the parallel math functions in `numpy`. However, I hope that the above can still enable you to extract value from this cookbook, even if you're programming in C++.