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 for sale at http://leanpub.com/quantlibpythoncookbook

This version was published on 2019-03-17

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

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]:
```

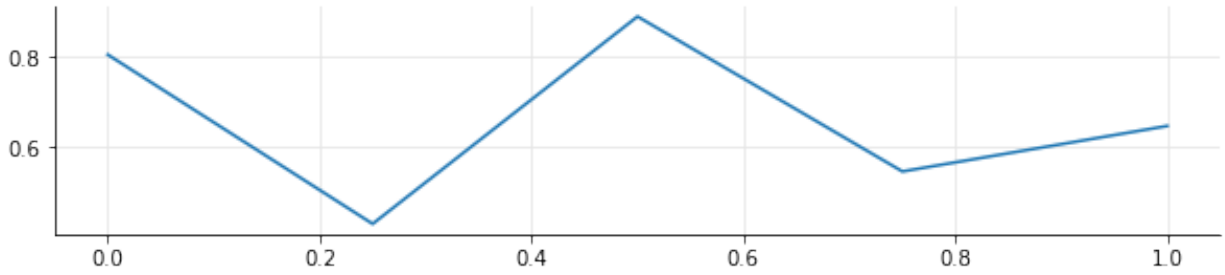
[4] http://jupyter.org/

	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.

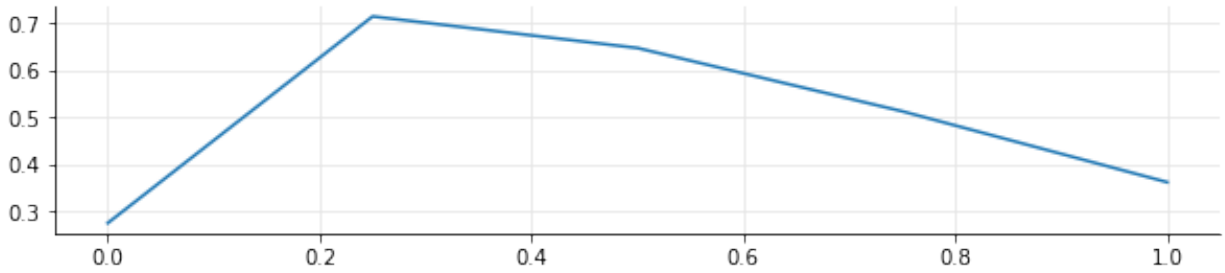
Notebooks can also include plots, as in the following cell:

```
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 0x7fef5c24fda0>]
```



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.

Basics

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]: from QuantLib import *
        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 = 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()  == 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 + Period(1, Weeks)))
        print("Add a month    : {0}".format(date + Period(1, Months)))
        print("Add a year     : {0}".format(date + Period(1, 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 == Date(31, 3, 2015))
        print(date > Date(30, 3, 2015))
        print(date < Date(1, 4, 2015))
        print(date != 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 = Date(31, 3, 2015)
        us_calendar = UnitedStates()
        italy_calendar = Italy()

        period = Period(60, 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 = 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 = Date(1, 1, 2015)
         termination_date = Date(1, 1, 2016)
         tenor = Period(Monthly)
         calendar = UnitedStates()
         business_convention = Following
         termination_business_convention = Following
         date_generation = DateGeneration.Forward
         end_of_month = False

         schedule = 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 = Date(1, 1, 2015)
         termination_date = Date(1, 1, 2016)
         first_date = Date(15, 1, 2015)
         schedule = Schedule(effective_date,
                             termination_date,
                             tenor,
                             calendar,
                             business_convention,
                             termination_business_convention,
```

```
                            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 = Date(1, 1, 2015)
         termination_date = Date(1, 1, 2016)
         penultimate_date = Date(15, 12, 2015)
         schedule = Schedule(effective_date,
                             termination_date,
                             tenor,
                             calendar,
                             business_convention,
                             termination_business_convention,
                             DateGeneration.Forward,
                             end_of_month,
                             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 = Date(1, 2, 2015)
         effective_date = Date(15, 12, 2014)
         termination_date = Date(1, 1, 2016)
         schedule = Schedule(effective_date,
                             termination_date,
                             tenor,
                             calendar,
                             business_convention,
                             termination_business_convention,
                             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 = Date(1, 1, 2015)
         penultimate_date = Date(1, 12, 2015)
         termination_date = Date(15, 1, 2016)
         schedule = Schedule(effective_date,
                             termination_date,
                             tenor,
                             calendar,
                             business_convention,
                             termination_business_convention,
                             DateGeneration.Forward,
                             end_of_month,
                             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 = [Date(2,1,2015), Date(2, 2,2015),
                  Date(2,3,2015), Date(1,4,2015),
                  Date(1,5,2015), Date(1,6,2015),
                  Date(1,7,2015), Date(3,8,2015),
                  Date(1,9,2015), Date(1,10,2015),
                  Date(2,11,2015), Date(1,12,2015),
                  Date(4,1,2016)]
         rolling_convention = Following

         schedule = 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 = ActualActual()
         compound_type = Compounded
         frequency = Annual

         interest_rate = 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 = 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]: from QuantLib import *

In [2]: today = Date(7, March, 2014)
        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 = EuropeanOption(PlainVanillaPayoff(Option.Call, 100.0),
                                EuropeanExercise(Date(7, 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 = SimpleQuote(100.0)
        r = SimpleQuote(0.01)
        sigma = 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 = FlatForward(0, TARGET(), QuoteHandle(r), Actual360())
        volatility = BlackConstantVol(0, TARGET(), QuoteHandle(sigma), 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 = BlackScholesProcess(QuoteHandle(u),
                                      YieldTermStructureHandle(riskFreeCurve),
                                      BlackVolTermStructureHandle(volatility))
```

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

```
In [7]: engine = 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.

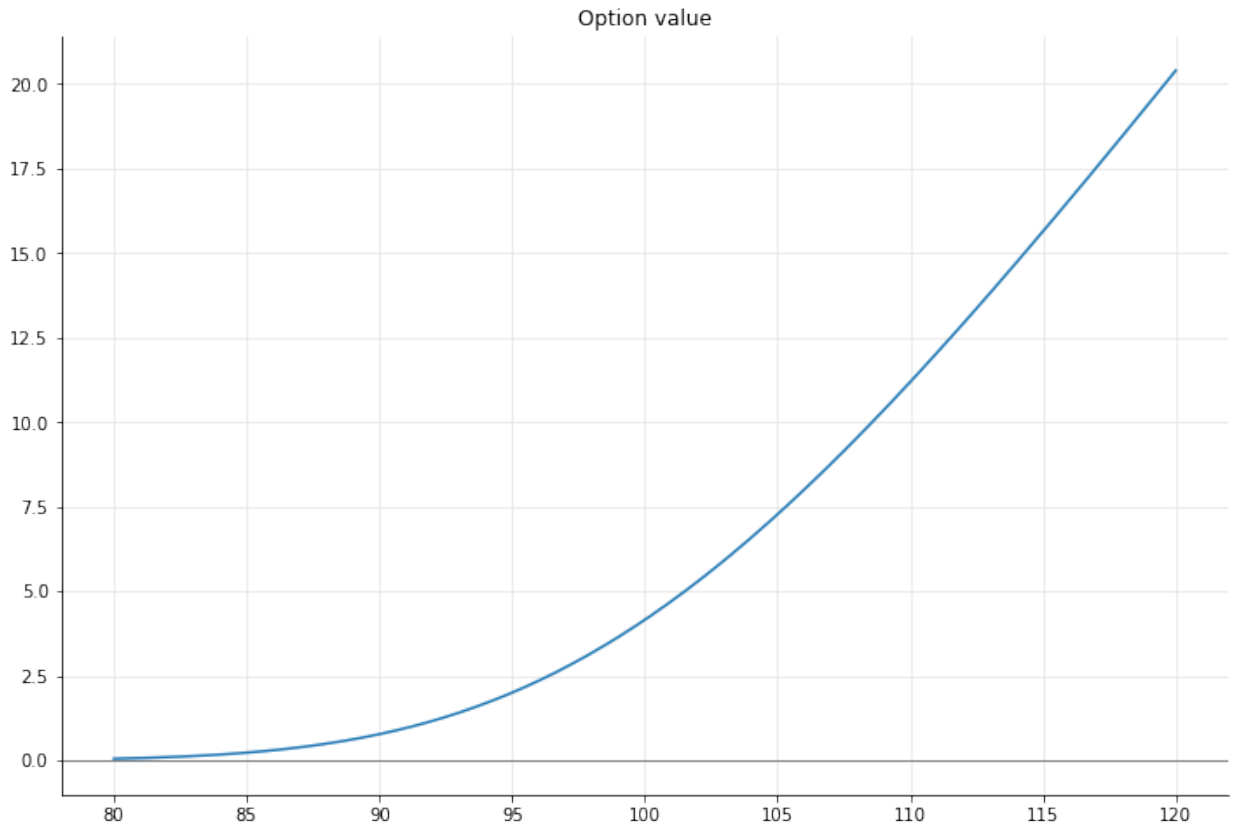
```
In [11]: u.setValue(105.0)
         print(option.NPV())

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…

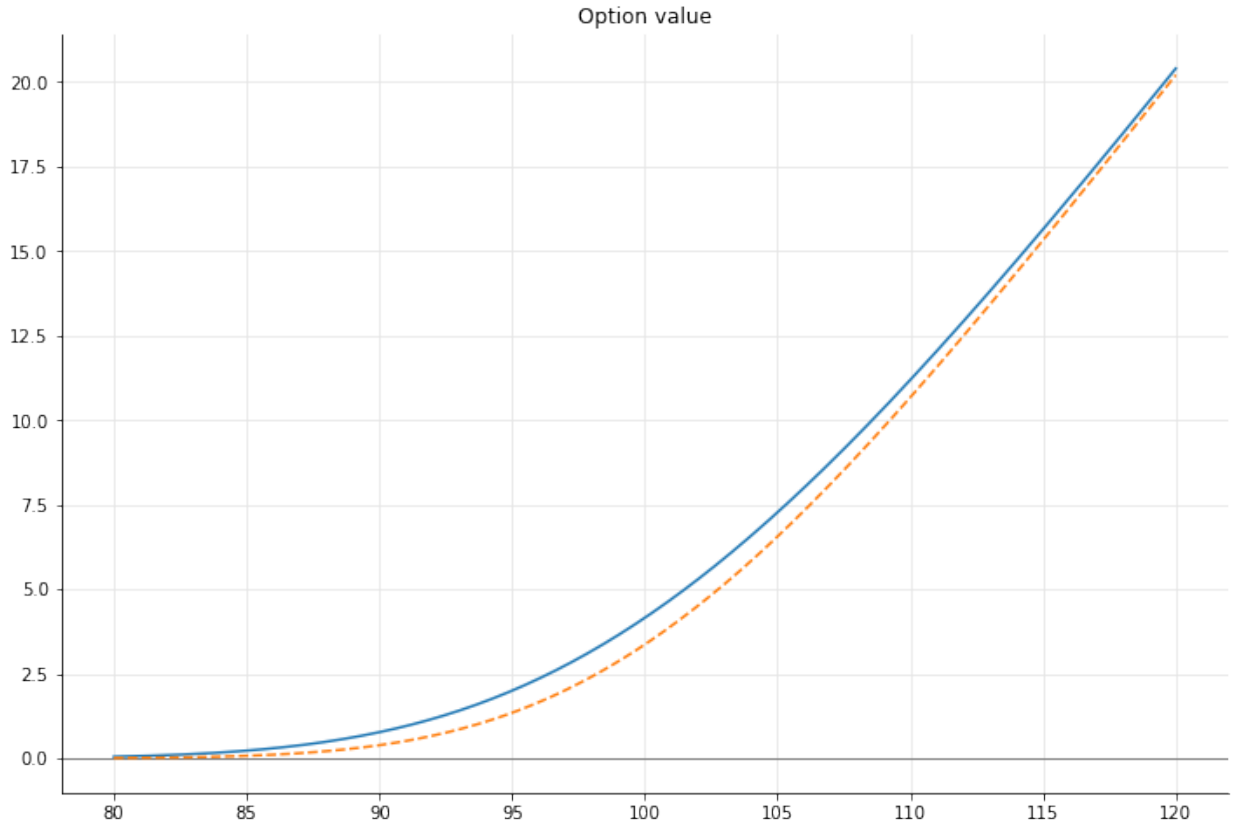
```
In [20]: u.setValue(105.0)
         r.setValue(0.01)
         sigma.setValue(0.20)
         print(option.NPV())

Out[20]: 7.27556357927846
```

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

```
In [21]: Settings.instance().evaluationDate = Date(7, 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]: Settings.instance().evaluationDate = Date(7, 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]: 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 = HestonModel(
             HestonProcess(YieldTermStructureHandle(riskFreeCurve),
                           YieldTermStructureHandle(FlatForward(0, TARGET(),
                                                                0.0, Actual360())),
                           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 = 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 = 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]: 7.306010762284822
         CPU times: user 1.71 s, sys: 0 ns, total: 1.71 s
         Wall time: 1.71 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]: 7.306010762284822
         CPU times: user 10 ms, sys: 0 ns, total: 10 ms
         Wall time: 976 µs
```

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]: 6.597869654923489
         CPU times: user 1.69 s, sys: 30 ms, total: 1.72 s
         Wall time: 1.71 s
```

3. Numerical Greeks calculation

In this notebook, I'll build on the facilities provided by the `Instrument` class (that is, its ability to detect changes in its inputs and recalculate accordingly) to show how to calculate numerical Greeks when the engine doesn't provide them.

Setup

As usual, we import the QuantLib module and set the evaluation date:

```
In [1]: from QuantLib import *

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

A somewhat exotic option

As an example, we'll use a knock-in barrier option:

```
In [3]: option = BarrierOption(Barrier.UpIn,
                               120.0,   # barrier
                               0.0,     # rebate
                               PlainVanillaPayoff(Option.Call, 100.0),
                               EuropeanExercise(Date(8, January, 2015)))
```

For the purpose of this example, the market data are the underlying value, the risk-free rate and the volatility. We wrap them in quotes, so that the instrument will be notified of any changes...

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

...and from the quotes we build the flat curves and the process that the engine requires. As explained in a later notebook, we build the term structures so that they move with the evaluation date; this will be useful further on.

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

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

Finally, we build the engine (the library provides one based on an analytic formula) and set it to the option.

```
In [7]: option.setPricingEngine(AnalyticBarrierEngine(process))
```

Now we can ask the option for its value...

```
In [8]: print(option.NPV())

Out[8]: 1.3657980739109867
```

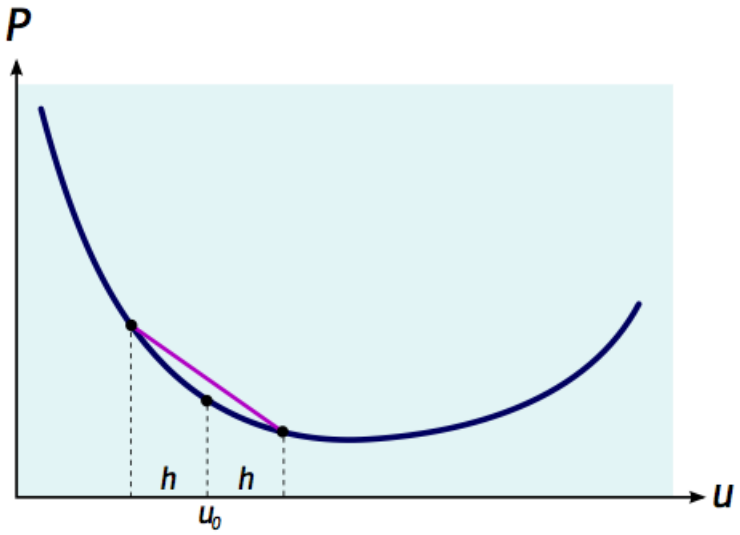
...but we're not so lucky when it comes to Greeks:

```
In [9]: print(option.delta())

Out[9]: ---------------------------------------------------------------------------
        RuntimeError                              Traceback (most recent call last)
        <ipython-input-9-dcaa26b2b456> in <module>()
        ----> 1 print(option.delta())

        /usr/local/lib/python3.6/dist-packages/QuantLib/QuantLib.py in delta(self)
          11432
          11433     def delta(self):
        > 11434         return _QuantLib.BarrierOption_delta(self)
          11435
          11436     def gamma(self):

        RuntimeError: delta not provided
```

The engine doesn't provide the delta, so asking for it raises an error.

Numerical calculation

What does a quant have to do? We can use numerical differentiation to approximate the Greeks, as shown in the next figure: that is, we can approximate the derivative by calculating the option value for two slightly different values of the underlying and by taking the slope between the resulting points.



The relevant formulas are:

$$\Delta = \frac{P(u_0 + h) - P(u_0 - h)}{2h} \qquad \Gamma = \frac{P(u_0 + h) - 2P(u_0) + P(u_0 - h)}{h^2}$$

where $P(u)$ is the price of the option for a given value of the underlying u.

Thanks to the framework we set in place, getting the perturbed prices is easy enough. We just have to set the relevant quote to the new value and ask the option for its price again. Thus, we choose a small increment and start. First, we save the current value of the option…

```
In [10]: u0 = u.value() ; h = 0.01

In [11]: P0 = option.NPV() ; print(P0)

Out[11]: 1.3657980739109867
```

…then we increase the underlying value and get the new option value…

```
In [12]: u.setValue(u0+h)
         P_plus = option.NPV() ; print(P_plus)

Out[12]: 1.3688112201958083
```

…then we do the same after decreasing the underlying value.

```
In [13]: u.setValue(u0-h)
         P_minus = option.NPV() ; print(P_minus)

Out[13]: 1.3627900998610207
```

Finally, we set the underlying value back to its current value.

```
In [14]: u.setValue(u0)
```

Applying the formulas above give us the desired Greeks:

```
In [15]: Delta = (P_plus - P_minus)/(2*h)
         Gamma = (P_plus - 2*P0 + P_minus)/(h*h)
         print(Delta)
         print(Gamma)

Out[15]: 0.3010560167393761
         0.05172234855521651
```

The approach is usable for any Greek. We can use the two-sided formula above, or the one-sided formula below if we want to minimize the number of evaluations:

$$\frac{\partial P}{\partial x} = \frac{P(x_0 + h) - P(x_0)}{h}$$

For instance, here we calculate Rho and Vega:

```
In [16]: r0 = r.value() ; h = 0.0001
         r.setValue(r0+h) ; P_plus = option.NPV()
         r.setValue(r0)
         Rho = (P_plus - P0)/h ; print(Rho)

Out[16]: 6.531038494277386

In [17]: sigma0 = sigma.value() ; h = 0.0001
         sigma.setValue(sigma0+h) ; P_plus = option.NPV()
         sigma.setValue(sigma0)
         Vega = (P_plus - P0)/h ; print(Vega)

Out[17]: 26.52519924198904
```

The approach for the Theta is a bit different, although it still relies on the fact that the option reacts to the change in the market data. The problem is that we don't have the time to maturity available as a quote, as was the case for the other quantities. Instead, since we set up the term structures so that they move with the evaluation date, we just have to set it to tomorrow's date to get the corresponding option value:

```
In [18]: Settings.instance().evaluationDate = today+1
         P1 = option.NPV()
         h = 1.0/365
         Theta = (P1-P0)/h ; print(Theta)

Out[18]: -10.770888399441302
```

4. Market quotes

In this notebook, I'll show a pitfall to avoid when multiple quotes need to be updated.

```
In [1]: %matplotlib inline
        import numpy as np
        import utils

In [2]: from QuantLib import *

In [3]: today = Date(17, October, 2016)
        Settings.instance().evaluationDate = today
```

Setting the stage

For illustration purposes, I'll create a bond curve using the same data and algorithm shown in one of the QuantLib C++ examples; namely, I'll give to the curve the functional form defined by the Nelson-Siegel model and I'll fit it to a number of bond. Here are the maturities in years and the coupons of the bonds I'll use:

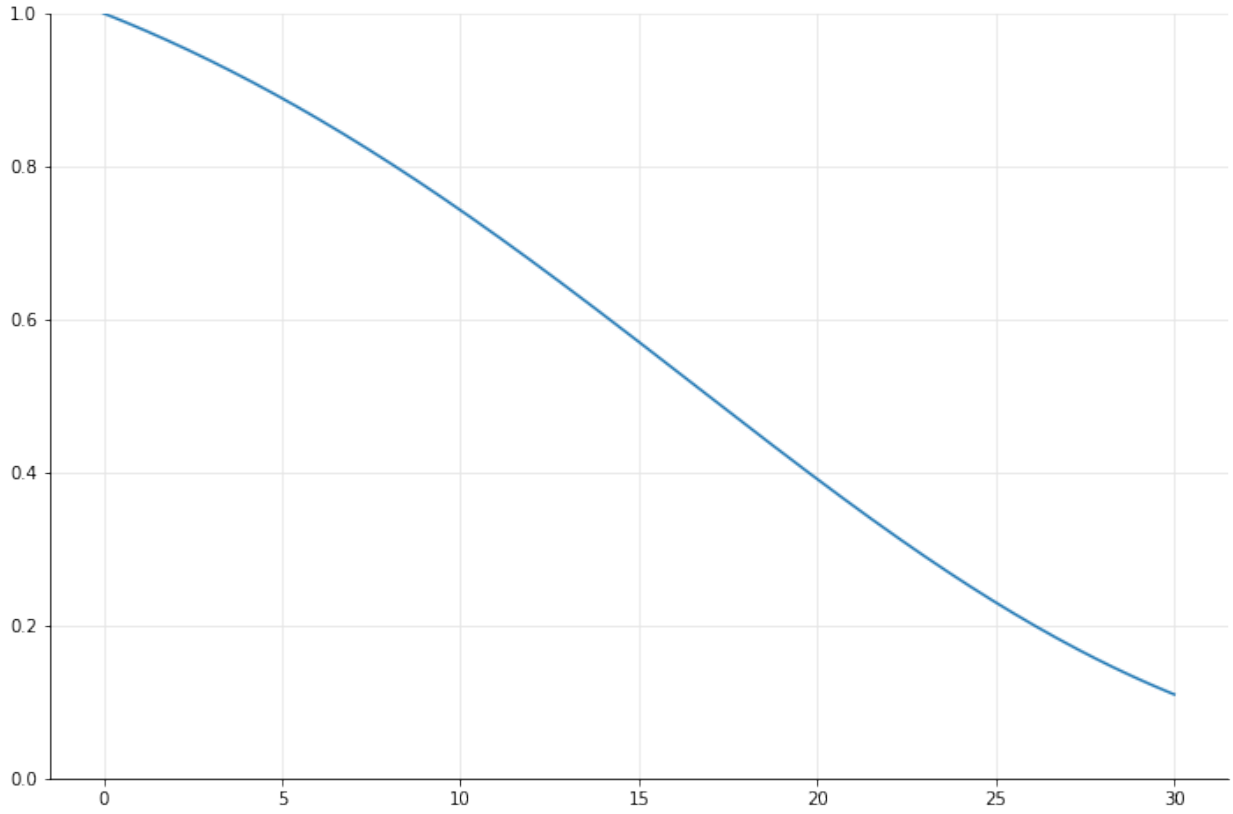
```
In [4]: data = [ (2, 0.02),  (4, 0.0225),  (6, 0.025),  (8, 0.0275),
                 (10, 0.03), (12, 0.0325), (14, 0.035), (16, 0.0375),
                 (18, 0.04), (20, 0.0425), (22, 0.045), (24, 0.0475),
                 (26, 0.05), (28, 0.0525), (30, 0.055)]
```

For simplicity, I'll use the same start date, frequency and conventions for all the bonds; this doesn't affect the point I'm going to make in the rest of the notebook. I'll also assume that all bonds currently price at 100. I'll skip over the details of building the curve now; the one thing you'll need to remember is that it depends on the quotes modeling the bond prices.

```
In [5]: calendar = TARGET()
        settlement = calendar.advance(today, 3, Days)
        quotes = []
        helpers = []
        for length, coupon in data:
            maturity = calendar.advance(settlement, length, Years)
            schedule = Schedule(settlement, maturity, Period(Annual),
                                calendar, ModifiedFollowing, ModifiedFollowing,
                                DateGeneration.Backward, False)
            quote = SimpleQuote(100.0)
            quotes.append(quote)
            helpers.append(FixedRateBondHelper(QuoteHandle(quote), 3, 100.0,
                                               schedule, [coupon], SimpleDayCounter(),
                                               ModifiedFollowing))

        curve = FittedBondDiscountCurve(0, calendar, helpers,
                                        SimpleDayCounter(), NelsonSiegelFitting())
```

Here is a visualization of the curve as discount factors versus time in years:

```
In [6]: sample_times = np.linspace(0.0, 30.0, 301)
        sample_discounts = [ curve.discount(t) for t in sample_times ]

        f, ax = utils.plot()
        ax.set_ylim(0.0, 1.0)
        ax.plot(sample_times, sample_discounts);
```



Also, here's a bond priced by discounting its coupons on the curve:

```
In [7]: schedule = Schedule(today, calendar.advance(today, 15, Years),
                            Period(Semiannual), calendar,
                            ModifiedFollowing, ModifiedFollowing,
                            DateGeneration.Backward, False)
        bond = FixedRateBond(3, 100.0, schedule, [0.04], Actual360())
        bond.setPricingEngine(DiscountingBondEngine(YieldTermStructureHandle(curve)))
        print(bond.cleanPrice())

Out[7]: 105.77449628297312
```

"It looked like a good idea at the time"

Now, let's add an observer that checks whether the bond is out of date, and if so recalculates the bond and outputs its new price. In Python, I can do this by defining a function to be triggered by the notifications, by passing it to the observer I'm creating, and (this last step is as in C++) by registering the observer with the bond.

As a reminder of how the whole thing works: the changes will come from the market quotes, but the observer doesn't need to be concerned with that and only registers with the object it's ultimately

interested in; in this case, the bond whose price it wants to monitor. A change in any of the market quotes will cause the quote to notify the helper, which in turn will notify the curve, and so on to the pricing engine, the bond and finally our observer.

```
In [8]: prices = []
        def print_price():
            p = bond.cleanPrice()
            prices.append(p)
            print(p)
        o = Observer(print_price)
        o.registerWith(bond)
```

The function also appends the new price to a list that can be used later as a history of the prices. Let's see if it works:

```
In [9]: quotes[2].setValue(101.0)

Out[9]: 105.77449628297312
        105.8656042875337
```

Whoa, what was that? The function was called twice, which surprised me too when I wrote this notebook. It turns out that, due to a glitch of multiple inheritance, the curve sends two notifications to the instrument. After the first, the instrument recalculates but the curve doesn't (which explains why the price doesn't change); after the second, the curve updates and the price changes. This should be fixed in a future release, but again it doesn't change the point of the notebook.

Let's set the quote back to its original value.

```
In [10]: quotes[2].setValue(100.0)

Out[10]: 105.8656042875337
         105.77449634664224
```

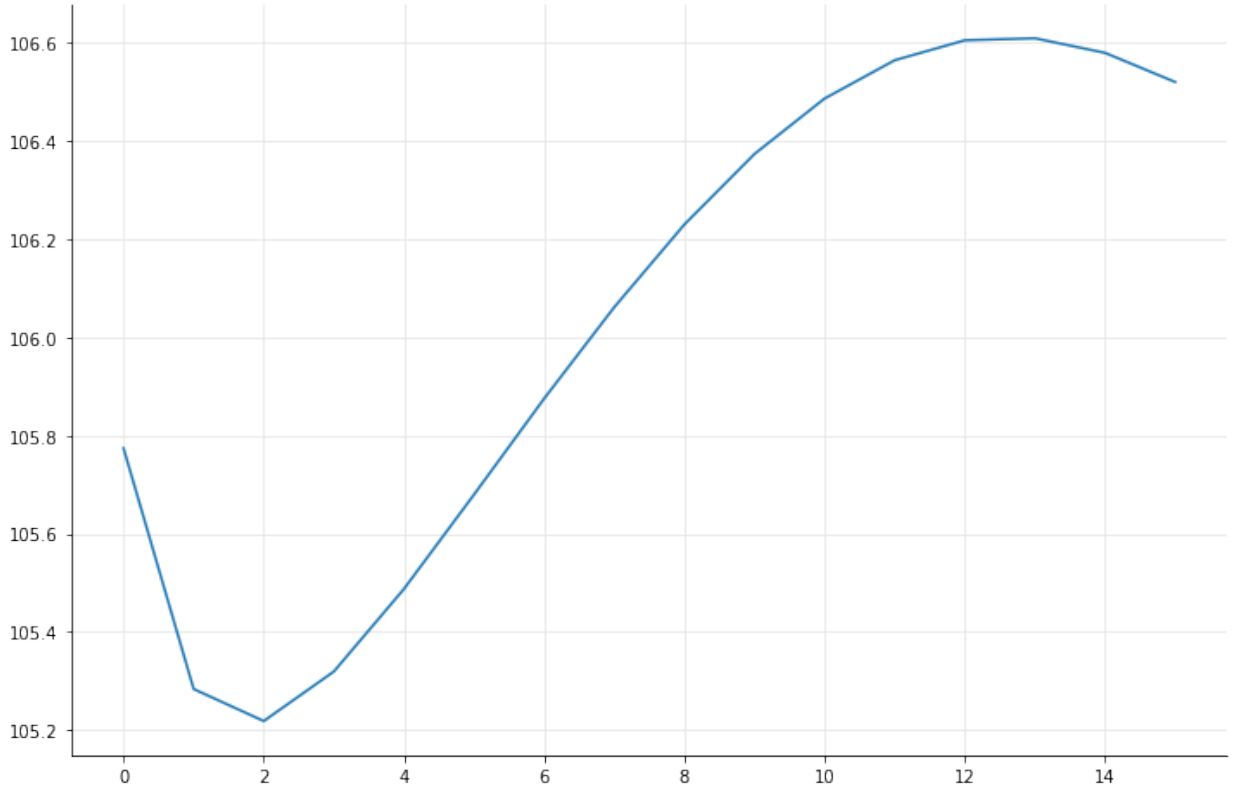
Now, let's say the market moves up and, accordingly, all the bonds prices increase to 101. Therefore, we need to update all the quotes.

```
In [11]: prices = []
         for q in quotes:
             q.setValue(101.0)
```

```
Out[11]: 105.77449634664224
         105.28388426272507
         105.28388426272507
         105.2186288679219
         105.2186288679219
         105.3195906444377
         105.3195906444377
         105.4878663448759
         105.4878663448759
         105.68032070200927
         105.68032070200927
         105.87580370787278
         105.87580370787278
         106.06201680440225
         106.06201680440225
         106.23044624497663
         106.23044624497663
         106.37409230798896
         106.37409230798896
         106.48708840758337
         106.48708840758337
         106.56505206364592
         106.56505206364592
         106.60570726105742
         106.60570726105742
         106.60980187075381
         106.60980187075381
         106.58011186582736
         106.58011186582736
         106.52070699740128
```

As you see, each of the updates sent a notification and thus triggered a recalculation. We can use the list of prices we collected (slicing it to skip duplicate values) to visualize how the price changed.

```
In [12]: unique_prices = prices[::2]+prices[-1::]
         _, ax = utils.plot()
         ax.plot(unique_prices, '-');
```



The first price is the original one, and the last price is the final one; but all those in between are calculated based on an incomplete set of changes in which some quotes were updated and some others weren't. Those are all incorrect, and (since they went both above and below the range of the real prices) outright dangerous in case there were any triggers on price levels that could have fired. Clearly, this is not the kind of behavior we want our code to have.

Alternatives?

There are workarounds we can apply. For instance, it's possible to freeze the bond temporarily, preventing it from forwarding notifications.

```
In [13]: bond.freeze()
```

Now, notifications won't be forwarded by the bond and thus won't reach our observer. In fact, the following loop won't print anything.

```
In [14]: for q in quotes:
             q.setValue(101.5)
```

When we restore the bond, it sends a single notification, which triggers only one recalculation and gives the correct final price.

```
In [15]: bond.unfreeze()

Out[15]: 106.85839373944943
```

When using C++, it's also possible to disable and re-enable notifications globally, which makes it more convenient.

But it all feels a bit convoluted anyway. The whole thing will be simpler if we discard the initial idea and don't force a recalculation for each notification.

Pull, don't push

It's preferable for updates to *not* trigger recalculation and just set some kind of dirty flag, just like the instruments in the library do. This way, you can control when the calculation occur.

To do so, let's remove the observer we have in place…

```
In [16]: del o
```

…and instead create one that raises a flag when it's notified.

```
In [17]: flag = {}
         flag['status'] = 'down'
         def set_flag():
             flag['status'] = 'up'
         o = Observer(set_flag)
         o.registerWith(bond)
```

The flag is initially down…

```
In [18]: print(flag)

Out[18]: {'status': 'down'}
```

…and quote changes cause it to be raised.

```
In [19]: for q in quotes:
             q.setValue(100.0)

In [20]: print(flag)

Out[20]: {'status': 'up'}
```

At this point, we can ask the bond for its final price.

```
In [21]: bond.cleanPrice()

Out[21]: 105.77449635334463
```

Better yet, we can let the instrument do that: let's remove the second observer, too, and just ask the instrument for its price after the changes. The instrument keeps track of whether it needs recalculation, so it doesn't need us to keep track of it.

```
In [22]: del o

In [23]: for q in quotes:
             q.setValue(101.0)

In [24]: bond.cleanPrice()

Out[24]: 106.52070687248381
```

So, less is more? In this case, it seems so.

5. Term structures and their reference dates

In this notebook, I show briefly how to set up term structures so that they track (or don't track) the global evaluation date.

Setup

Import the QuantLib module and set up the global evaluation date. You might want to take note of the date, since we'll be moving it around later on.

```
In [1]: from QuantLib import *

In [2]: Settings.instance().evaluationDate = Date(3, October, 2014)
```

Specifying the reference date of a term structure

In not-too-accurate terms, the reference date of a term structure is where it begins. It can be the evaluation date, but you might also want it to start on the spot date, for instance.

We have two possibilities to define a reference date for a curve—even though some particular classes only allow one of them.

The first is to define it by means of a (possibly null) offset from the current evaluation date; e.g., "two business days after the evaluation date" to define it as the spot date, or "no business days" to define it as the evaluation date itself. I'll do it here by building a sample curve over a few swaps.

Never mind the helper object that I'm building here…

```
In [3]: helpers = [ SwapRateHelper(QuoteHandle(SimpleQuote(rate/100.0)),
                                   Period(*tenor), TARGET(),
                                   Annual, Unadjusted,
                                   Thirty360(),
                                   Euribor6M())
                    for tenor, rate in [((2,Years), 0.201),
                                        ((3,Years), 0.258),
                                        ((5,Years), 0.464),
                                        ((10,Years), 1.151),
                                        ((15,Years), 1.588)] ]
```

…because the construction of the curve is the main point: note the `0` and `TARGET()` arguments, specifying the number of days and the calendar used to determine business days.

```
In [4]: curve1 = PiecewiseFlatForward(0, TARGET(), helpers, Actual360())
```

The second possibility is to specify the reference date explicitly. For instance, the `ForwardCurve` class takes a vector of specific dates and the corresponding rates and interpolates between them; the first passed date is taken as the reference date of the curve.

For comparison purposes, I'll ask the curve above for its nodes and use them to build a `ForwardCurve` instance:

```
In [5]: dates, rates = zip(*curve1.nodes())

In [6]: curve1.nodes()

Out[6]: ((Date(3,10,2014), 0.0019777694879293093),
         (Date(7,10,2016), 0.0019777694879293093),
         (Date(9,10,2017), 0.0036475517704509294),
         (Date(7,10,2019), 0.007660760701876805),
         (Date(7,10,2024), 0.018414773669420893),
         (Date(8,10,2029), 0.025311634328221498))
```

The curve built based on these data will be the same as the first, except that we're specifying its reference date explicitly as October 3rd (the first passed date).

```
In [7]: curve2 = ForwardCurve(dates, rates, Actual360())
```

Both curves are defined over the same range of dates...

```
In [8]: print("{0} to {1}".format(curve1.referenceDate(), curve1.maxDate()))
        print("{0} to {1}".format(curve2.referenceDate(), curve2.maxDate()))

Out[8]: October 3rd, 2014 to October 8th, 2029
        October 3rd, 2014 to October 8th, 2029
```

...and return the same rates, whether we ask for a given time (for instance, 5 years)...

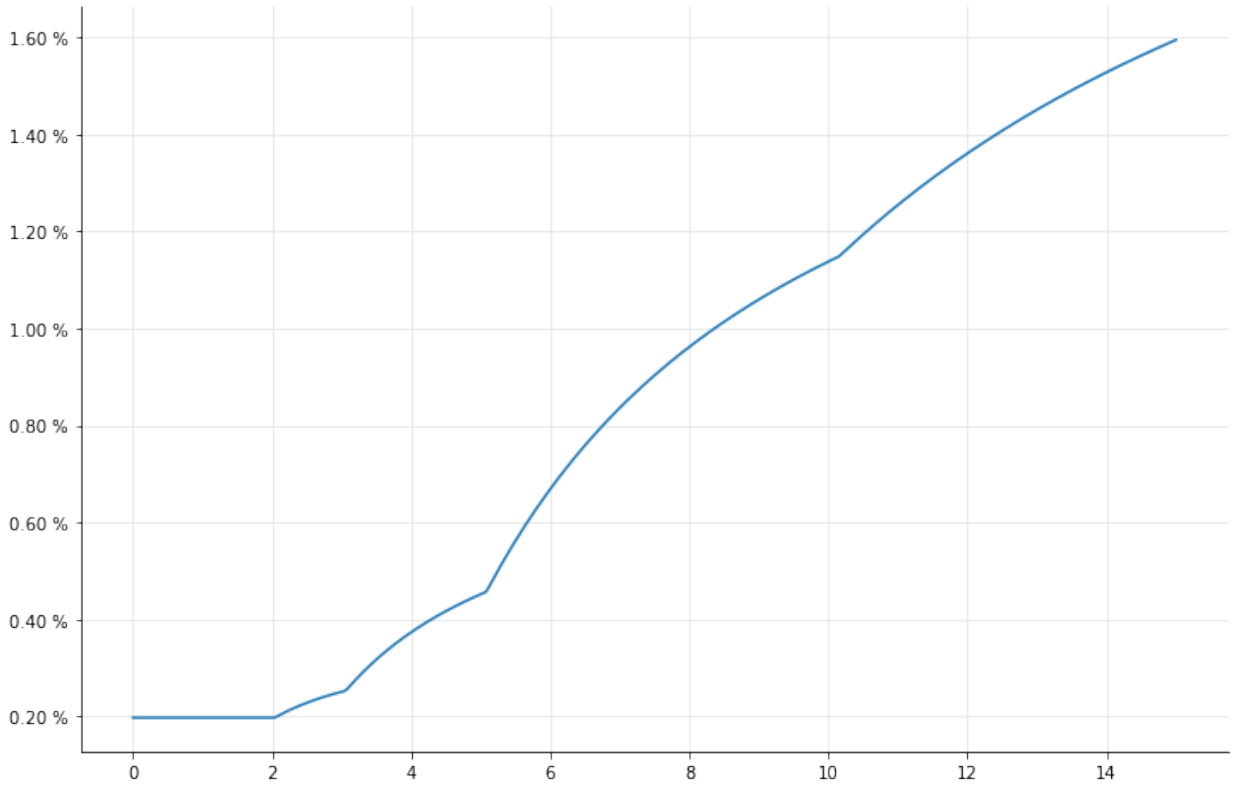
```
In [9]: print(curve1.zeroRate(5.0, Continuous))
        print(curve2.zeroRate(5.0, Continuous))

Out[9]: 0.452196 % Actual/360 continuous compounding
        0.452196 % Actual/360 continuous compounding
```

...or for a given date.

```
In [10]: print(curve1.zeroRate(Date(7, September, 2019), Actual360(), Continuous))
         print(curve2.zeroRate(Date(7, September, 2019), Actual360(), Continuous))

Out[10]: 0.452196 % Actual/360 continuous compounding
         0.452196 % Actual/360 continuous compounding
```

With the help of a couple more Python modules, we can also plot the whole curve by asking for rates over a set of times:

```
In [11]: %matplotlib inline
         import utils
         from matplotlib.ticker import FuncFormatter
         import numpy as np

In [12]: times = np.linspace(0.0, 15.0, 400)
         rates = [ curve1.zeroRate(t, Continuous).rate() for t in times ]
         _, ax = utils.plot()
         ax.yaxis.set_major_formatter(
             FuncFormatter(lambda r,pos: utils.format_rate(r,2)))
         ax.plot(times, rates);
```



Moving the evaluation date

To recap: we built the first curve specifying its reference date relative to the evaluation date, and the second curve specifying its reference date explicitly. Now, what happens if we change the evaluation date?

```
In [13]: Settings.instance().evaluationDate = Date(19, September, 2014)
```

As you might expect, the reference date of the first curve changes accordingly while that of the second curve doesn't.

We can see how the range of definition has now changed for the first curve, but not for the second:

```
In [14]: print("{0} to {1}".format(curve1.referenceDate(), curve1.maxDate()))
         print("{0} to {1}".format(curve2.referenceDate(), curve2.maxDate()))

Out[14]: September 19th, 2014 to September 24th, 2029
         October 3rd, 2014 to October 8th, 2029
```

And of course the rates have changed, too...

```
In [15]: print(curve1.zeroRate(5.0, Continuous))
         print(curve2.zeroRate(5.0, Continuous))

Out[15]: 0.452196 % Actual/360 continuous compounding
         0.452196 % Actual/360 continuous compounding
```

...if we look at them in the right way. The whole curve has moved back a couple of weeks, so if we ask for a given time we'll get the same rates; in other words, we're asking for the zero rate over five years after the reference date, and that remains the same for a rigid translation of the curve. If we ask for the zero rate at a given date, though, we'll see the effect:

```
In [16]: print(curve1.zeroRate(Date(7, September, 2019), Actual360(), Continuous))
         print(curve2.zeroRate(Date(7, September, 2019), Actual360(), Continuous))

Out[16]: 0.454618 % Actual/360 continuous compounding
         0.452196 % Actual/360 continuous compounding
```

Notifications

Finally, we can see how the two curves behave differently also with respect to notifications. Let's make two observers...

```
In [17]: def make_observer(i):
             def say():
                 s = "Observer %d notified" % i
                 print('-'*len(s))
                 print(s)
                 print('-'*len(s))
             return Observer(say)

         obs1 = make_observer(1)
         obs2 = make_observer(2)
```

...and check that they work correctly by connecting them to a few quotes. The first observer will receive notifications from the first and third quote, and the second observer will receive notifications from the second and third quote.

```
In [18]: q1 = SimpleQuote(1.0)
         obs1.registerWith(q1)

         q2 = SimpleQuote(2.0)
         obs2.registerWith(q2)

         q3 = SimpleQuote(3.0)
         obs1.registerWith(q3)
         obs2.registerWith(q3)
```

If I trigger a change in the first quote, the first observer is notified and outputs a message:

```
In [19]: q1.setValue(1.5)

Out[19]: --------------------
         Observer 1 notified
         --------------------
```

A change in the second quote causes a message from the second observer...

```
In [20]: q2.setValue(1.9)

Out[20]: --------------------
         Observer 2 notified
         --------------------
```

...and a change in the third quote causes both observers to react.

```
In [21]: q3.setValue(3.1)

Out[21]: --------------------
         Observer 2 notified
         --------------------
         --------------------
         Observer 1 notified
         --------------------
```

Now let's connect the observers to the curves. The first observer will receive notifications from the curve that moves with the evaluation date, and the second observer will receive notifications from the curve that doesn't move.

```
In [22]: obs1.registerWith(curve1)
         obs2.registerWith(curve2)
```

Now we can see what happens when the evaluation date changes again:

```
In [23]: Settings.instance().evaluationDate = Date(23, September, 2014)

Out[23]: --------------------
         Observer 1 notified
         --------------------
```

As you can see, only the moving curve sent a notification. The other did not, since it was not modified by the change of evaluation date.

6. Pricing over a range of days

Based on questions on *Stack Exchange* from Charles[1], bob.jonst[2], MCM[3] and lcheng[4].

```
In [1]: from QuantLib import *
        import numpy as np
        np.random.seed(42)
```

Let's say we have an instrument (a fixed-rate bond, for instance) that we want to price on a number of dates. I assume we also have the market quotes, or the curves, corresponding to each of the dates; in this case we only need interest rates, but the library works the same way for any quotes.

We'll store the resulting prices in a dictionary, with the date as the key.

```
In [2]: prices = {}
```

Producing a single price

To price the bond on a single date, we create the instrument itself...

```
In [3]: start_date = Date(8, February, 2016)
        maturity_date = start_date + Period(5, Years)
        schedule = Schedule(start_date, maturity_date, Period(Semiannual), TARGET(),
                            Following, Following, DateGeneration.Backward, False)
        coupons = [0.01]*10
        bond = FixedRateBond(3, 100, schedule, coupons, Thirty360())
```

...and the required discount curve. For brevity, here I'm interpolating precomputed rates; I might as well bootstrap the curve on a set of market rates.

[1] https://stackoverflow.com/questions/32869325/
[2] https://quant.stackexchange.com/questions/35961/
[3] https://quant.stackexchange.com/questions/38509
[4] https://quant.stackexchange.com/questions/36830/

```
In [4]: today = Date(9, May, 2018)
        nodes = [ today + Period(i, Years) for i in range(11) ]
        rates = [ 0.007, 0.010, 0.012, 0.013, 0.014,
                  0.016, 0.017, 0.018, 0.020, 0.021, 0.022 ]
        discount_curve = ZeroCurve(nodes, rates, Actual360())
```

Given the bond and the curve, we link them together through an engine, set the evaluation date and get the result.

```
In [5]: discount_handle = RelinkableYieldTermStructureHandle(discount_curve)
        bond.setPricingEngine(DiscountingBondEngine(discount_handle))

In [6]: Settings.instance().evaluationDate = today

In [7]: prices[today] = bond.cleanPrice()
        print(prices[today])

Out[7]: 99.18942082987543
```

Pricing on multiple days

We could repeat the above for all dates, but it goes against the grain of the library. The architecture (see chapter 2 of Implementing QuantLib[5] for details) was designed so that the instrument can react to changing market conditions; therefore, we can avoid recreating the instrument. We'll only change the discount curve and the evaluation date.

For instance, here I'll calculate the price for the business day before today:

```
In [8]: calendar = TARGET()
        yesterday = calendar.advance(today, -1, Days)
```

I'll generate random rates to avoid coming up with a new set; but the idea is to build the correct discount curve for the evaluation date.

```
In [9]: nodes = [ yesterday + Period(i, Years) for i in range(11) ]
        base_rates = np.array(rates)
        rates = base_rates * np.random.normal(loc=1.0, scale=0.005, size=base_rates.shape)
        discount_curve = ZeroCurve(nodes, list(rates), Actual360())
```

As I mentioned, I need to set the new evaluation date and to link the handle in the engine to the new discount curve...

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

```
In [10]: Settings.instance().evaluationDate = yesterday
         discount_handle.linkTo(discount_curve)
```

...after which the bond returns the updated price.

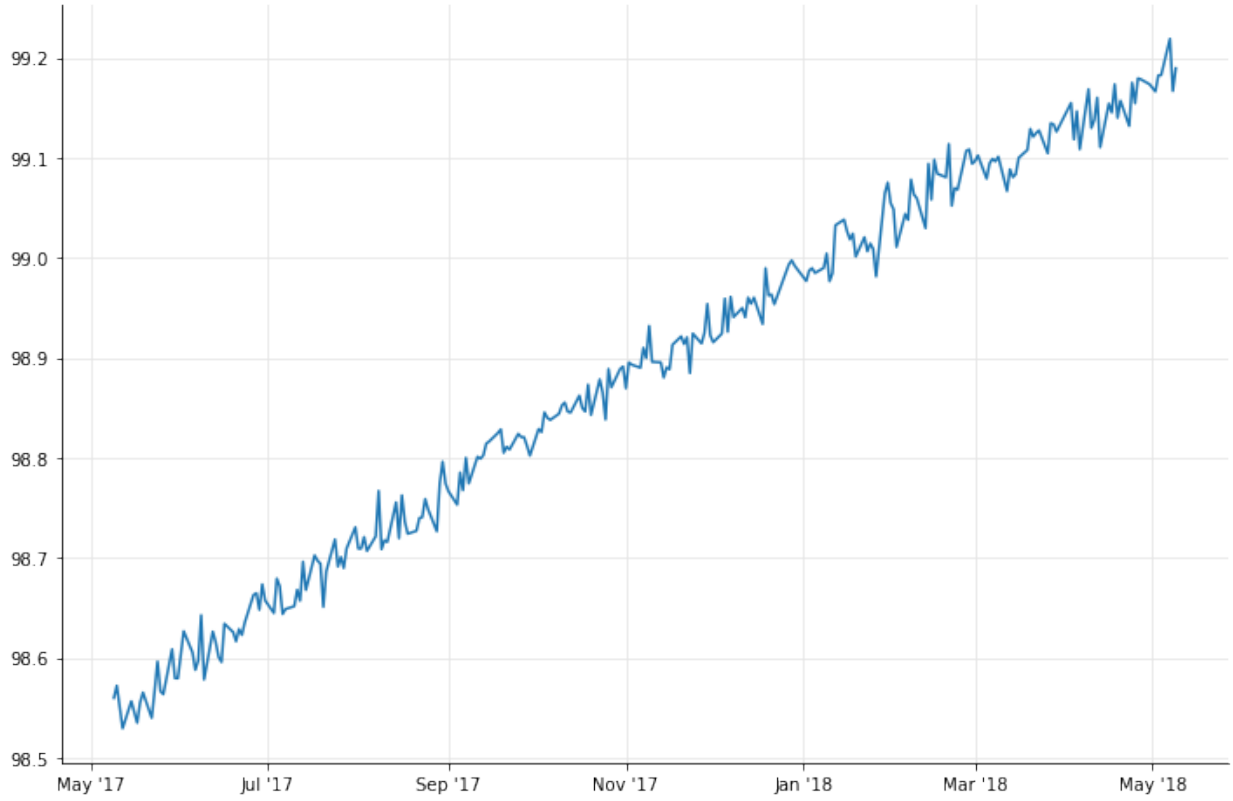
```
In [11]: prices[yesterday] = bond.cleanPrice()
         print(prices[yesterday])

Out[11]: 99.16663635835845
```

By repeating the process, I can generate prices for, say, the whole of last year. Again, I'm generating random rates to avoid tedious listings or external data files; you'll use the correct ones instead.

```
In [12]: first_date = calendar.advance(today, -1, Years)
         date = calendar.advance(yesterday, -1, Days)

         while date >= first_date:
             nodes = [ date + Period(i, Years) for i in range(11) ]
             rates = base_rates * np.random.normal(loc=1.0, scale=0.005, size=base_rates.shape)
             discount_curve = ZeroCurve(nodes, list(rates), Actual360())

             Settings.instance().evaluationDate = date
             discount_handle.linkTo(discount_curve)

             prices[date] = bond.cleanPrice()
             date = calendar.advance(date, -1, Days)
```

Here are the results. Through the random noise, you can see how the price increases as the bond gets nearer to maturity.

```
In [13]: %matplotlib inline
         import utils

In [14]: dates, values = zip(*sorted(prices.items()))

In [15]: fig, ax = utils.plot()
         ax.xaxis.set_major_formatter(utils.date_formatter())
         ax.plot_date([ utils.to_datetime(d) for d in dates ], values,'-');
```



Using quotes

If we work with quotes, we can also avoid rebuilding the curve. Let's say our discount curve is defined as a risk-free curve with an additional credit spread. The risk-free curve is bootstrapped from a number of market rates; for simplicity, here I'll use a set of overnight interest-rate swaps, but you'll use whatever makes sense in your case.

```
In [16]: index = Eonia()
         tenors = [ Period(i, Years) for i in range(1,11) ]
         rates = [ 0.010, 0.012, 0.013, 0.014, 0.016, 0.017, 0.018, 0.020, 0.021, 0.022 ]

         quotes = []
         helpers = []
         for tenor, rate in zip(tenors, rates):
             q = SimpleQuote(rate)
             h = OISRateHelper(2, tenor, QuoteHandle(q), index)
             quotes.append(q)
             helpers.append(h)
```

One thing to note: I'll setup the curve so that it moves with the evaluation date. This means that I won't pass an explicit reference date, but a number of business days and a calendar. Passing 0, as

in this case, will cause the reference date of the curve to equal the evaluation date; passing 2, for instance, would cause it to equal the corresponding spot date.

```
In [17]: risk_free_curve = PiecewiseFlatForward(0, TARGET(), helpers, Actual360())
```

Finally, I'll manage credit as an additional spread over the curve:

```
In [18]: spread = SimpleQuote(0.01)
         discount_curve = ZeroSpreadedTermStructure(YieldTermStructureHandle(risk_free_curve),
                                                    QuoteHandle(spread))
```

Now we can recalculate today's price...

```
In [19]: prices = {}

         Settings.instance().evaluationDate = today
         discount_handle.linkTo(discount_curve)

         prices[today] = bond.cleanPrice()
         print(prices[today])

Out[19]: 96.50362161659807
```

...and as before, we go back; except this time we don't need to build a new curve. Instead, we can set new values to the quotes and they will trigger the necessary recalculations.

```
In [20]: date = calendar.advance(today, -1, Days)

         base_rates = np.array(rates)

         while date >= first_date:
             rates = base_rates * np.random.normal(loc=1.0, scale=0.005, size=base_rates.shape)
             for q, r in zip(quotes, rates):
                 q.setValue(r)
             spread.setValue(spread.value()*np.random.normal(loc=1.0, scale=0.005))

             Settings.instance().evaluationDate = date

             prices[date] = bond.cleanPrice()
             date = calendar.advance(date, -1, Days)
```

Note that we didn't create any new object in the loop; we're only settings new values to the quotes.

Again, here are the results:

```
In [21]: dates, values = zip(*sorted(prices.items()))

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

A complication: past fixings

For instruments that depend on the floating rate, we might need some past fixings. This is not necessarily related to pricing on a range of dates: even on today's date, we need the fixing for the current coupon. Let's set the instrument up...

```
In [22]: forecast_handle = YieldTermStructureHandle(risk_free_curve)
         index = Euribor6M(forecast_handle)

         bond = FloatingRateBond(3, 100, schedule, index, Thirty360())
         bond.setPricingEngine(DiscountingBondEngine(discount_handle))

In [23]: Settings.instance().evaluationDate = today
         for q, r in zip(quotes, base_rates):
             q.setValue(r)
         spread.setValue(0.01)
```

...and try to price it. No joy.

```
In [24]: print(bond.cleanPrice())

Out[24]: ---------------------------------------------------------------------------
         RuntimeError                              Traceback (most recent call last)
         <ipython-input-24-74ed33c38331> in <module>()
         ----> 1 print(bond.cleanPrice())

         /usr/local/lib/python3.6/dist-packages/QuantLib/QuantLib.py in cleanPrice(self, *args)
           14307
           14308     def cleanPrice(self, *args):
         > 14309         return _QuantLib.Bond_cleanPrice(self, *args)
           14310
           14311     def dirtyPrice(self, *args):

         RuntimeError: Missing Euribor6M Actual/360 fixing for February 6th, 2018
```

Being in the past, the fixing can't be retrieved from the curve. We have to store it into the index, after which the calculation works:

```
In [25]: index.addFixing(Date(6, February,2018), 0.005)

         print(bond.cleanPrice())

Out[25]: 97.11939323923686
```

When pricing on a range of dates, though, we need to take into account the fact that the current coupon changes as we go back in time. These two dates will work...

```
In [26]: Settings.instance().evaluationDate = Date(1, March, 2018)
         print(bond.cleanPrice())

         Settings.instance().evaluationDate = Date(15, February, 2018)
         print(bond.cleanPrice())

Out[26]: 96.84331874622794
         96.79054303973298
```

...but this one causes the previous coupon to be evaluated, and that requires a new fixing:

```
In [27]: Settings.instance().evaluationDate = Date(1, February, 2018)
         print(bond.cleanPrice())

Out[27]: ---------------------------------------------------------------------------
         RuntimeError                              Traceback (most recent call last)
         <ipython-input-27-33dc024b8a28> in <module>()
               1 Settings.instance().evaluationDate = Date(1, February, 2018)
         ----> 2 print(bond.cleanPrice())

         /usr/local/lib/python3.6/dist-packages/QuantLib/QuantLib.py in cleanPrice(self, *args)
           14307
           14308     def cleanPrice(self, *args):
         > 14309         return _QuantLib.Bond_cleanPrice(self, *args)
           14310
           14311     def dirtyPrice(self, *args):

         RuntimeError: Missing Euribor6M Actual/360 fixing for August 4th, 2017
```

Once we add it, the calculation works again.

```
In [28]: index.addFixing(Date(4, August, 2017), 0.004)
         print(bond.cleanPrice())

Out[28]: 96.98060241422583
```

(If you're wondering how the calculation worked before, since this coupon belonged to the bond: on the other evaluation dates, this coupon was expired and the engine could skip it without needing to calculate its amount. Thus, its fixing didn't need to be retrieved.)

More complications: future prices

What if we go forward in time, instead of pricing on past dates?

For one thing, we'll need to forecast curves in some way. One way is to imply them from today's curves: I talk about implied curves in another notebook, so I won't repeat myself here. Let's assume we have implied rates and we can set them. Once we do, we can price in the future just as easily as we do in the past. As I write this, it's May 19th 2018, and June 1st is in the future:

```
In [29]: Settings.instance().evaluationDate = Date(1, June, 2018)

         print(bond.cleanPrice())

Out[29]: 97.2126812565699
```

However, there's another problem, as pointed out by Mariano Zeron[6] in a post to the QuantLib mailing list. If we go further in the future, the bond will require—so to speak—future past fixings.

```
In [30]: Settings.instance().evaluationDate = Date(1, June, 2019)

         print(bond.cleanPrice())

Out[30]: ---------------------------------------------------------------------------
         RuntimeError                              Traceback (most recent call last)
         <ipython-input-30-ae84687e04f4> in <module>()
               1 Settings.instance().evaluationDate = Date(1, June, 2019)
               2
         ----> 3 print(bond.cleanPrice())

         /usr/local/lib/python3.6/dist-packages/QuantLib/QuantLib.py in cleanPrice(self, *args)
           14307
           14308     def cleanPrice(self, *args):
         > 14309         return _QuantLib.Bond_cleanPrice(self, *args)
           14310
           14311     def dirtyPrice(self, *args):

         RuntimeError: Missing Euribor6M Actual/360 fixing for February 6th, 2019
```

Here the curve starts on June 1st 2019, and cannot retrieve the fixing at the start of the corresponding coupon.

One way out of this might be to forecast fixings off the current curve and store them:

```
In [31]: Settings.instance().evaluationDate = Date(1, June, 2018)

         future_fixing = index.fixing(Date(6,February,2019))
         print(future_fixing)
         index.addFixing(Date(6,February,2019), future_fixing)

Out[31]: 0.011387399107860378
```

This way, they will be retrieved in the same way as real past fixings.

[6] https://sourceforge.net/p/quantlib/mailman/message/35270917/

```
In [32]: Settings.instance().evaluationDate = Date(1, June, 2019)

         print(bond.cleanPrice())

Out[32]: 98.30830224923507
```

Of course, you might forecast them in a better way: that's up to you. And if you're worried that this might interfere with pricing on today's date, don't: stored fixings are only used if they're in the past with respect to the evaluation date. The fixing I'm storing below for February 3rd 2021 will be retrieved if the evaluation date is later...

```
In [33]: index.addFixing(Date(3,February,2021), 0.02)

         Settings.instance().evaluationDate = Date(1, June, 2021)
         print(index.fixing(Date(3,February,2021)))

Out[33]: 0.02
```

...but it will be forecast from the curve when it's after the evaluation date:

```
In [34]: Settings.instance().evaluationDate = Date(1, June, 2020)
         print(index.fixing(Date(3,February,2021)))

Out[34]: 0.011367299732914539
```

7. A note on random numbers and dimensionality

Setup

Import QuantLib and the graphing module.

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        from QuantLib import *
```

Also, define a helper function to make the notebook less verbose.

```
In [2]: def set_unit_square(ax):
            ax.axis('scaled')
            ax.set_xlim([0,1])
            ax.set_ylim([0,1])
```

Covering a unit square

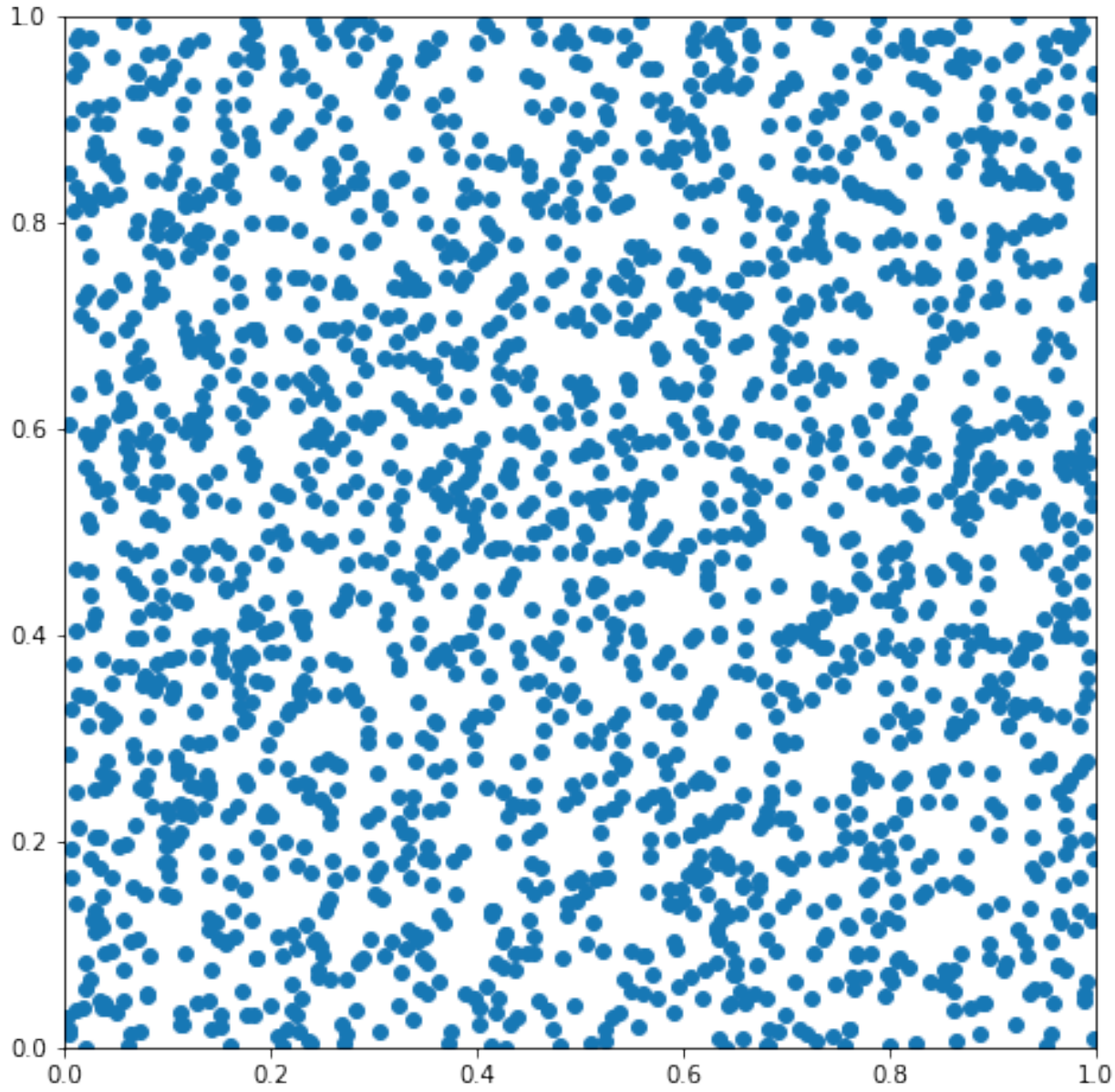
Let's say we want to extract points inside a unit square; that is, pairs of points in the domain $(0,1) \times (0,1)$. The dimensionality of the problem is 2, since we need 2 numbers, x and y, per each sample.

With pseudo-random numbers, it doesn't matter much: we can just extract single numbers and form pairs from them.

```
In [3]: rng = MersenneTwisterUniformRng(42)
```

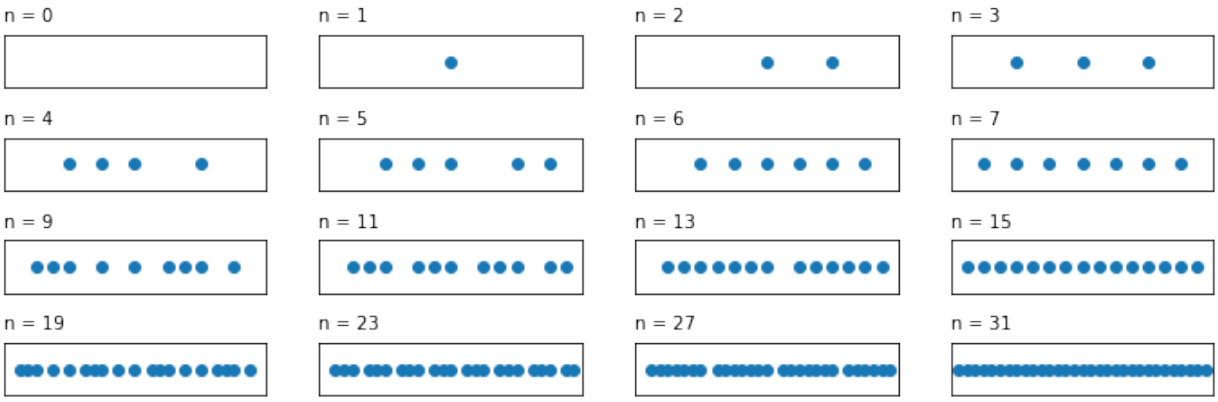
```
In [4]: xs = []
        ys = []
        for i in range(2047):
            xs.append(rng.next().value())
            ys.append(rng.next().value())
```

```
In [5]: fig = plt.figure(figsize=(8,8))
        ax = fig.add_subplot(1,1,1)
        set_unit_square(ax)
        ax.plot(xs,ys,'o');
```



The same doesn't hold for quasi-random numbers, for which each sample is correlated to the one that follows it in order to cover the domain evenly. We can see this by plotting the sequence of Sobol numbers generated to cover the 1-dimensional unit interval:

```
In [6]: fig = plt.figure(figsize=(12,4))
        for i, n in enumerate([0,1,2,3, 4,5,6,7, 9,11,13,15, 19,23,27,31]):
            rng = SobolRsg(1)
            xs = [ rng.nextSequence().value()[0] for j in range(n) ]
            ax = fig.add_subplot(4, 4, i+1)
            ax.axis('scaled')
            ax.set_xlim([0,1])
            ax.set_ylim([-0.1,0.1])
            ax.set_xticks([])
            ax.set_yticks([])
            ax.plot(xs,[0]*len(xs),'o')
            ax.text(0.0, 0.15, 'n = %d' % n)
```

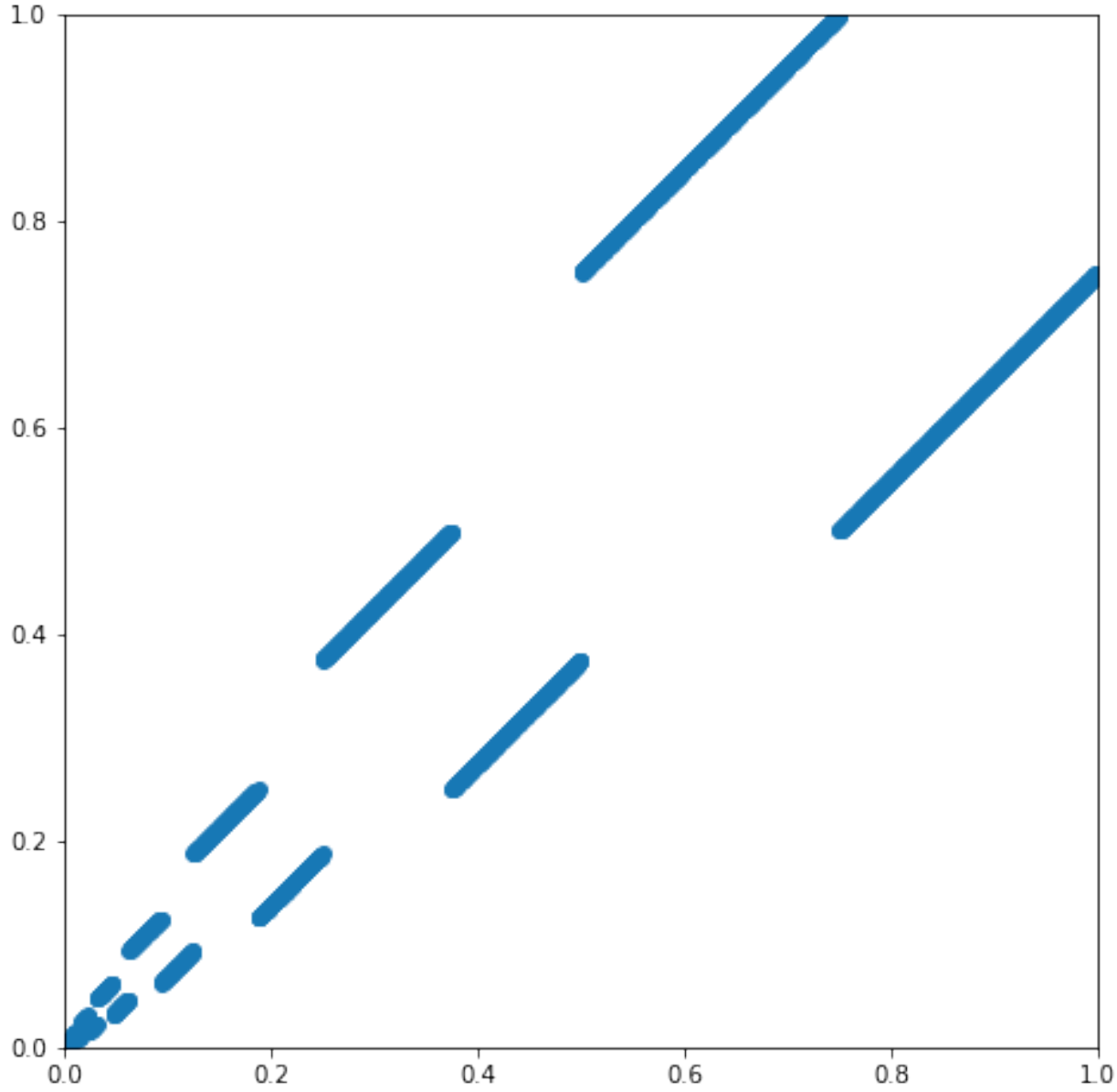


The points are not added randomly at all, but in a predetermined sequence. This ruins the random properties of the sequence when used with the wrong dimensionality. (You can also see how an even coverage is only obtained for a number of samples of the form $n = 2^i - 1$ for some i.)

```
In [7]: rng = SobolRsg(1)
```

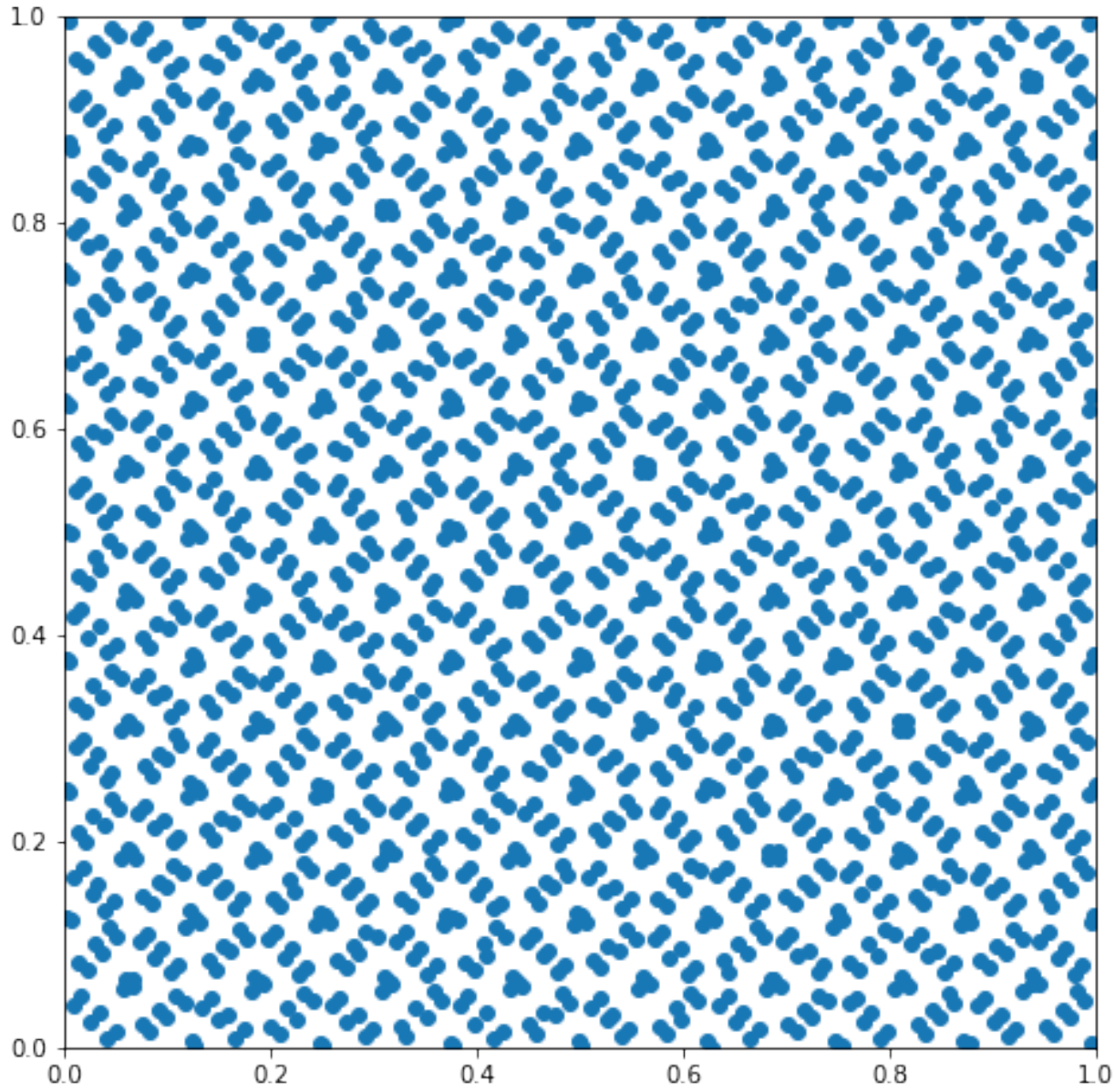
```
In [8]: xs = []
        ys = []
        for i in range(2047):
            xs.append(rng.nextSequence().value()[0])
            ys.append(rng.nextSequence().value()[0])
```

```
In [9]: fig = plt.figure(figsize=(8,8))
        ax = fig.add_subplot(1,1,1)
        set_unit_square(ax)
        ax.plot(xs,ys,'o');
```



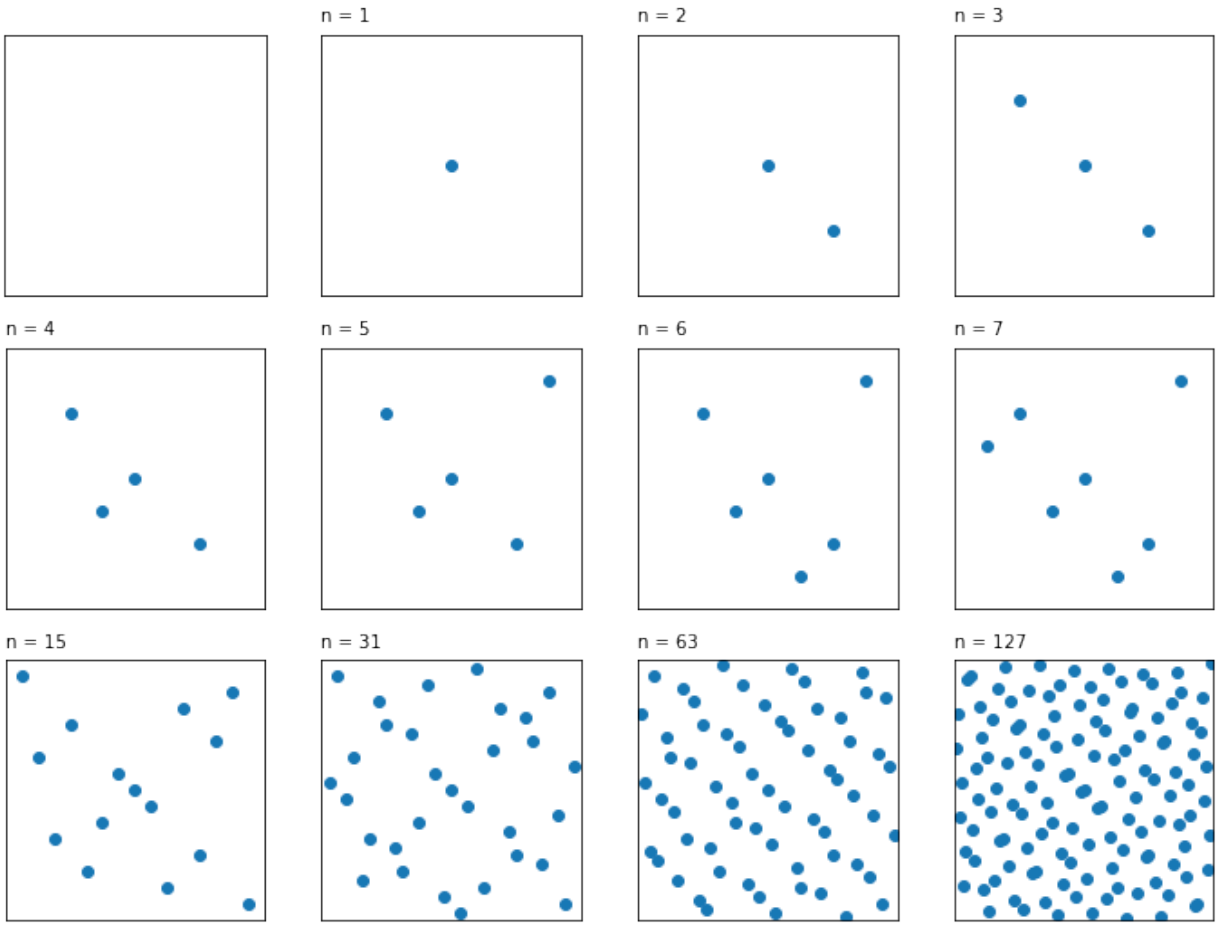
To cover the domain correctly, we have to use the right dimensionality.

```
In [10]: rng = SobolRsg(2)

In [11]: xs = []
         ys = []
         for i in range(2047):
             x,y = rng.nextSequence().value()
             xs.append(x)
             ys.append(y)

In [12]: fig = plt.figure(figsize=(8,8))
         ax = fig.add_subplot(1,1,1)
         set_unit_square(ax)
         ax.plot(xs,ys,'o');
```



The pattern above covers the square evenly, and also causes projections on the two axes to have good coverage (which wouldn't happen, for instance, with a regular placement in rows and columns; the projections of most points would coincide). It is also interesting to see how the coverage is built as the number of samples increase:

```
In [13]: fig = plt.figure(figsize=(12,9))
         for i, n in enumerate([0,1,2,3, 4,5,6,7, 15,31,63,127]):
             rng = SobolRsg(2)
             ax = fig.add_subplot(3, 4, i+1)
             ax.set_xticks([])
             ax.set_yticks([])
             if n == 0:
                 continue
             points = [ rng.nextSequence().value() for j in range(n) ]
             xs,ys = zip(*points)
             ax.axis('scaled')
             ax.set_xlim([0,1])
             ax.set_ylim([0,1])
             ax.plot(xs,ys,'o')
             ax.text(0.0, 1.05, 'n = %d' % n)
```



Dimensionality of Monte Carlo simulations

The classes in the QuantLib Monte Carlo framework will check the dimensionality of the generators they're given and will warn you if it's not correct. It's still up to you to find the correct one, while writing your engines (for details on that, you can check chapter 6 of *Implementing QuantLib*[1]).

For instance, let's say that you want to simulate three correlated stocks; and for sake of simplicity, let's say they follow the Black-Scholes process. You'll build a process for each of them…

```
In [14]: today = Date(27,January,2018)
         Settings.instance().evaluationDate = today
         risk_free = YieldTermStructureHandle(FlatForward(today, 0.01, Actual360()))

         processes = [
             BlackScholesProcess(QuoteHandle(SimpleQuote(S)),
                                 risk_free,
                                 BlackVolTermStructureHandle(
                                     BlackConstantVol(today, TARGET(), sigma, Actual360())))
             for S, sigma in [(100, 0.20),
                              ( 80, 0.25),
                              (110, 0.18)] ]
```

…and a single multi-dimensional process that correlates them. In this case, the resulting process has three random drivers.

```
In [15]: rho = [[1.0, 0.6, 0.8],
                [0.6, 1.0, 0.4],
                [0.8, 0.4, 1.0]]

         process = StochasticProcessArray(processes, rho)
         print(process.factors())

Out[15]: 3
```

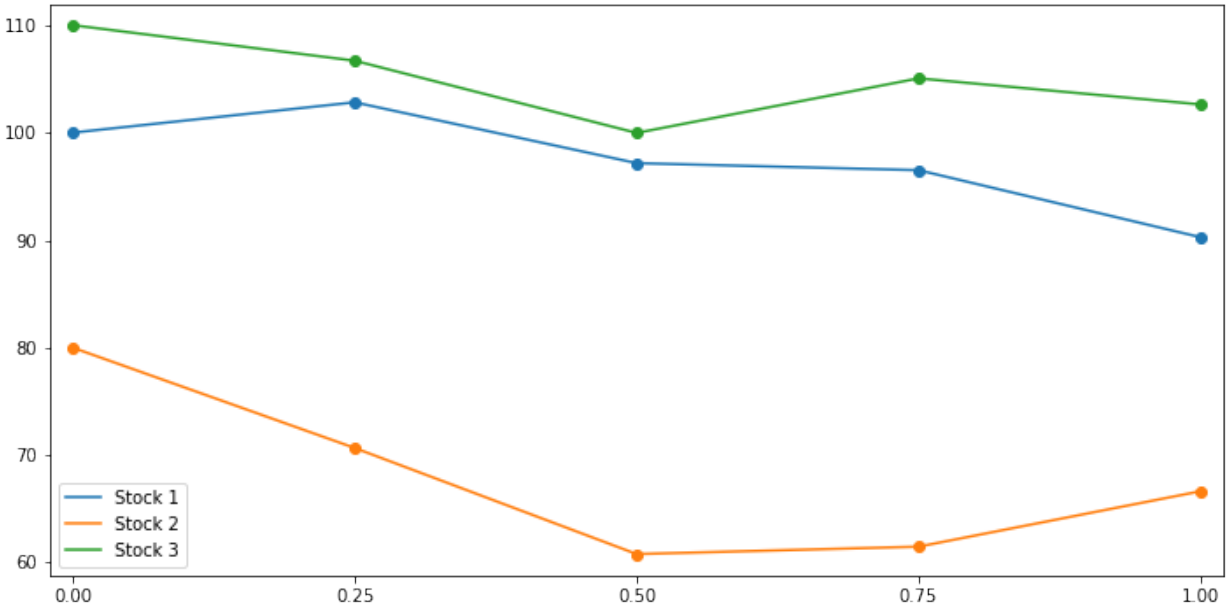
Now, let's say that we want to simulate paths over four steps, starting from today and ending one year from now. Each sample of the Monte Carlo simulation will need three random number for each step, for a total of 12 random numbers. This is the dimensionality of the problem; and, as I mentioned, the framework will check it and complain if it doesn't match. (Please bear with me as I build the several classes needed for random-numbers generation. If find yourself doing this, you might want to write a helper function, like I do here.)

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

```
In [16]: def rng(dimensionality):
             return GaussianRandomSequenceGenerator(
                      UniformRandomSequenceGenerator(
                        dimensionality,
                        UniformRandomGenerator(42)))

         times = [0.25, 0.50, 0.75, 1.0]
         generator = GaussianMultiPathGenerator(process, times, rng(10))

Out[16]: ---------------------------------------------------------------------------
         RuntimeError                              Traceback (most recent call last)
         <ipython-input-16-2f7dd2d480ec> in <module>()
               6
               7 times = [0.25, 0.50, 0.75, 1.0]
         ----> 8 generator = GaussianMultiPathGenerator(process, times, rng(10))

         /usr/local/lib/python3.6/dist-packages/QuantLib/QuantLib.py in __init__(self, process\
, times, generator, brownianBridge)
           18392
           18393     def __init__(self, process, times, generator, brownianBridge=False):
         > 18394         this = _QuantLib.new_GaussianMultiPathGenerator(process, times, gener\
ator, brownianBridge)
           18395         try:
           18396             self.this.append(this)

         RuntimeError: dimension (10) is not equal to (3 * 4) the number of factors times the \
number of time steps
```

As you might expect, the thing works with the correct dimensionality:

```
In [17]: generator = GaussianMultiPathGenerator(process, times, rng(12))

In [18]: sample = generator.next().value()

In [19]: fig = plt.figure(figsize=(12,6))
         ax = fig.add_subplot(1,1,1)
         ts = [0.0] + times
         y_min = 80
         y_max = 110
         for i in range(3):
             p, = ax.plot(ts, sample[i], label='Stock %d' % (i+1))
             ax.plot(ts, sample[i], 'o', color=p.get_color())
             y_min = min(y_min, min(sample[i]))
             y_max = max(y_max, max(sample[i]))
         ax.set_xlim(0.0-0.02, 1.0+0.02)
         ax.set_xticks(ts)
         ax.set_ylim(y_min-2, y_max+2)
         ax.legend(loc='best');
```


110
100
90
80
70
60
0.00
0.25
0.50
0.75
1.00
Stock 1
Stock 2
Stock 3

Interest-rate curves

8. 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]: from QuantLib import *

In [3]: today = Date(11, December, 2012)
        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 = [ DepositRateHelper(QuoteHandle(SimpleQuote(rate/100)),
                                      Period(1,Days), fixingDays,
                                      TARGET(), Following, False, 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 = Eonia()

In [6]: helpers += [ OISRateHelper(2, Period(*tenor),
                                   QuoteHandle(SimpleQuote(rate/100)), eonia)
                     for rate, tenor in [(0.070, (1,Weeks)), (0.069, (2,Weeks)),
                                         (0.078, (3,Weeks)), (0.074, (1,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 += [ DatedOISRateHelper(start_date, end_date,
                                        QuoteHandle(SimpleQuote(rate/100)), eonia)
                     for rate, start_date, end_date in [
                         (0.046, Date(16,January,2013), Date(13,February,2013)),
                         (0.016, Date(13,February,2013), Date(13,March,2013)),
                         (-0.007, Date(13,March,2013), Date(10,April,2013)),
                         (-0.013, Date(10,April,2013), Date(8,May,2013)),
                         (-0.014, Date(8,May,2013), Date(12,June,2013))] ]
```

Finally, we add OIS quotes up to 30 years.

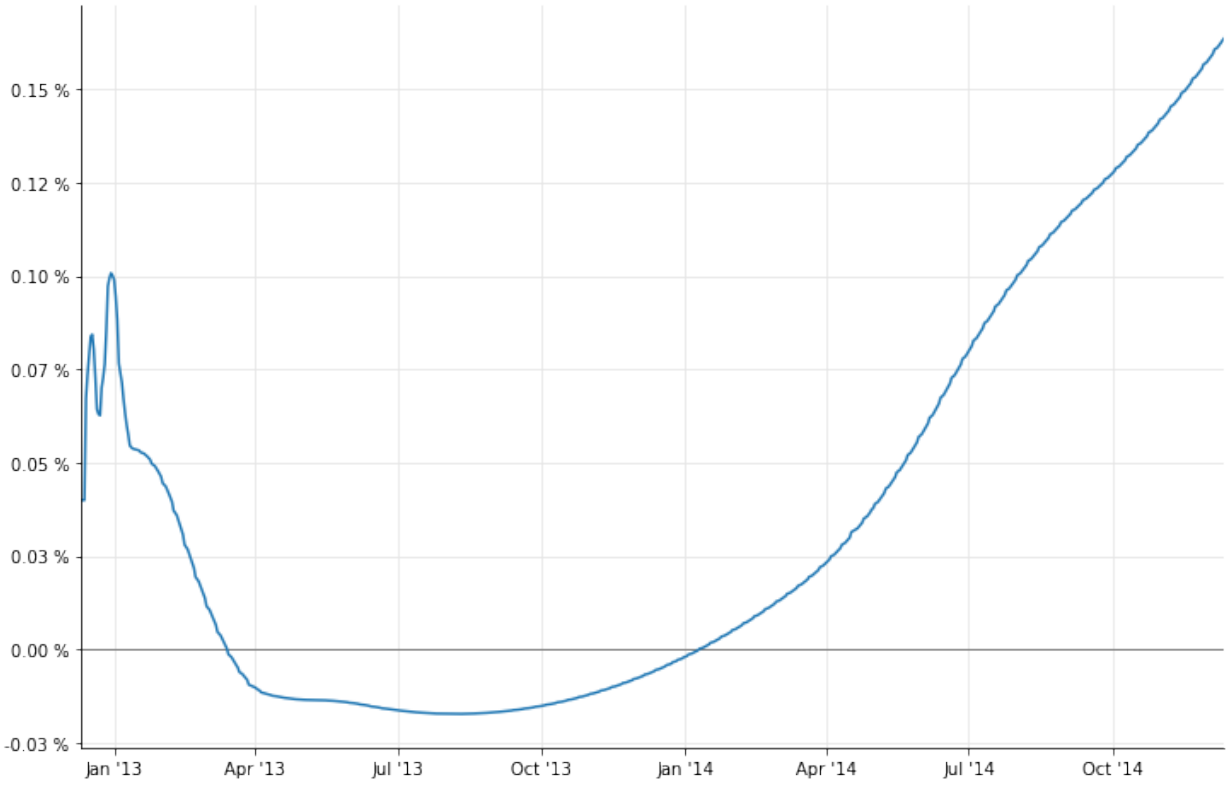
```
In [8]: helpers += [ OISRateHelper(2, Period(*tenor),
                                   QuoteHandle(SimpleQuote(rate/100)), eonia)
                     for rate, tenor in [(0.002, (15,Months)), (0.008, (18,Months)),
                                         (0.021, (21,Months)), (0.036, (2,Years)),
                                         (0.127, (3,Years)), (0.274, (4,Years)),
                                         (0.456, (5,Years)), (0.647, (6,Years)),
                                         (0.827, (7,Years)), (0.996, (8,Years)),
                                         (1.147, (9,Years)), (1.280, (10,Years)),
                                         (1.404, (11,Years)), (1.516, (12,Years)),
                                         (1.764, (15,Years)), (1.939, (20,Years)),
                                         (2.003, (25,Years)), (2.038, (30,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 = PiecewiseLogCubicDiscount(0, TARGET(),
                                                  helpers, 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 + Period(2,Years)
         dates = [ Date(serial) for serial in range(today.serialNumber(),
                                                    end.serialNumber()+1) ]
         rates_c = [ eonia_curve_c.forwardRate(d, TARGET().advance(d,1,Days),
                                               Actual360(), 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

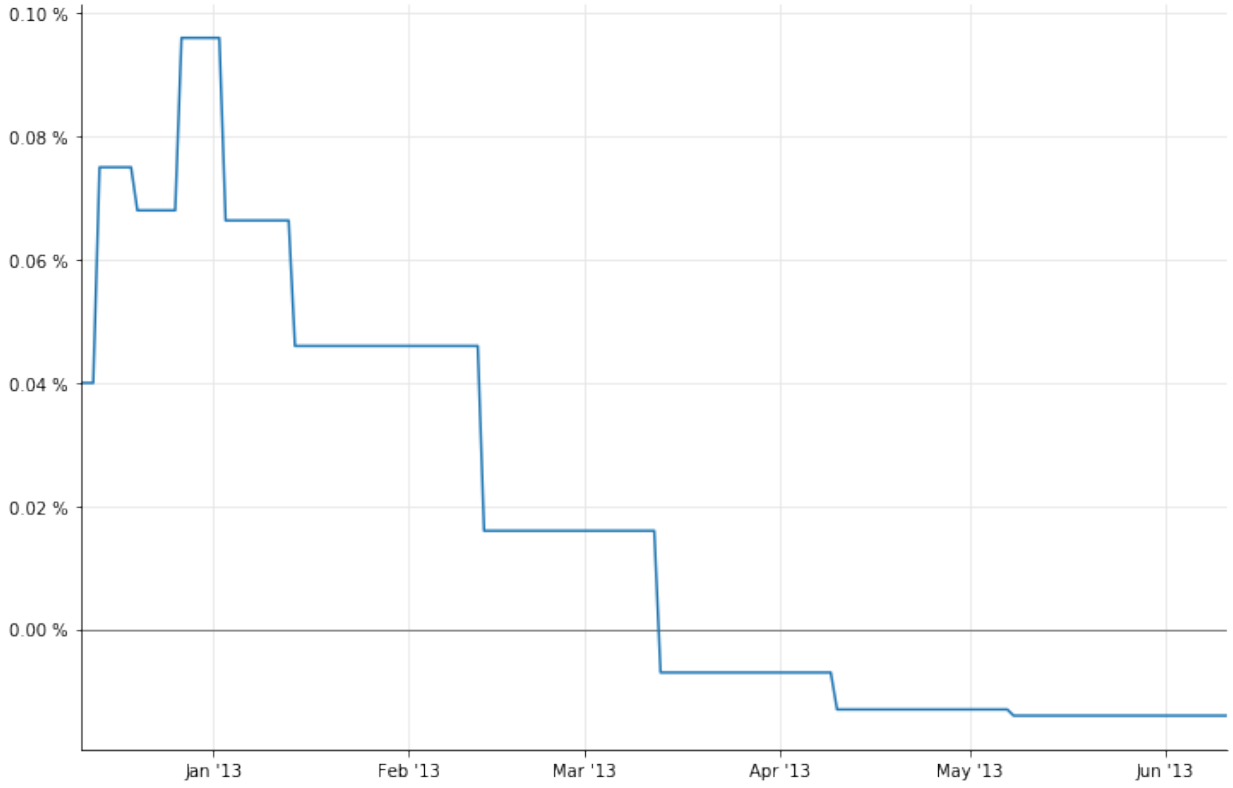
As explained in section 4.8 of the paper, the turn-of-year effect is a jump in interest rates due to an increased demand for liquidity at the end of the year. The jump is embedded in any quoted rates that straddles the end of the year and must be treated separately; the `YieldTermStructure` class allows 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 = PiecewiseFlatForward(0, TARGET(),
                                                helpers, 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 + Period(6,Months)
         dates = [ Date(serial) for serial in range(today.serialNumber(),
                                                    end.serialNumber()+1) ]
         rates_ff = [ eonia_curve_ff.forwardRate(d, TARGET().advance(d,1,Days),
                                                 Actual360(), 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…

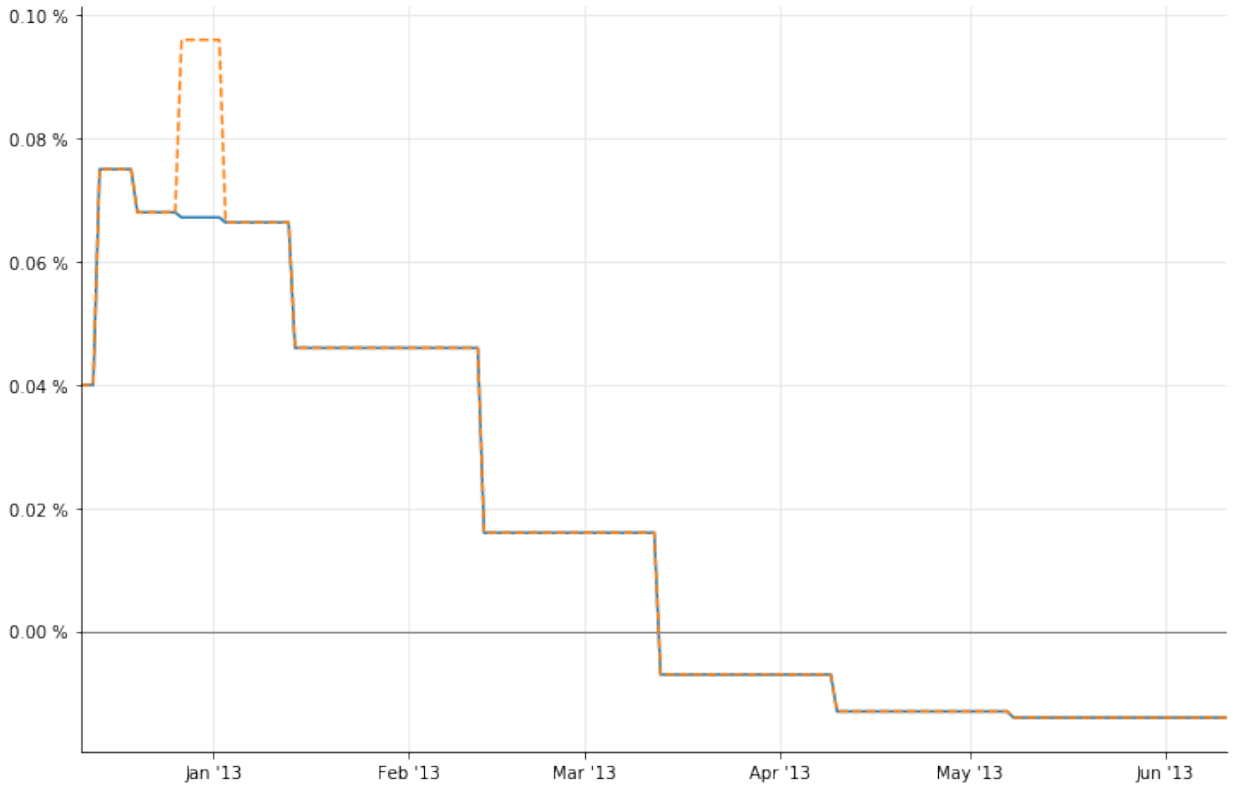
```
In [17]: nodes[6] = (nodes[6][0], (nodes[5][1]+nodes[7][1])/2.0)
         nodes[:9]

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 = 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, TARGET().advance(d,1,Days),
                                                Actual360(), 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 = Date(31,December,2012) - Period(1,Weeks)
         d2 = Date(31,December,2012) + Period(1,Weeks)

In [22]: F = eonia_curve_ff.forwardRate(d1, d2, Actual360(), Simple).rate()
         F_1 = temp_curve.forwardRate(d1, d2, Actual360(), 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.

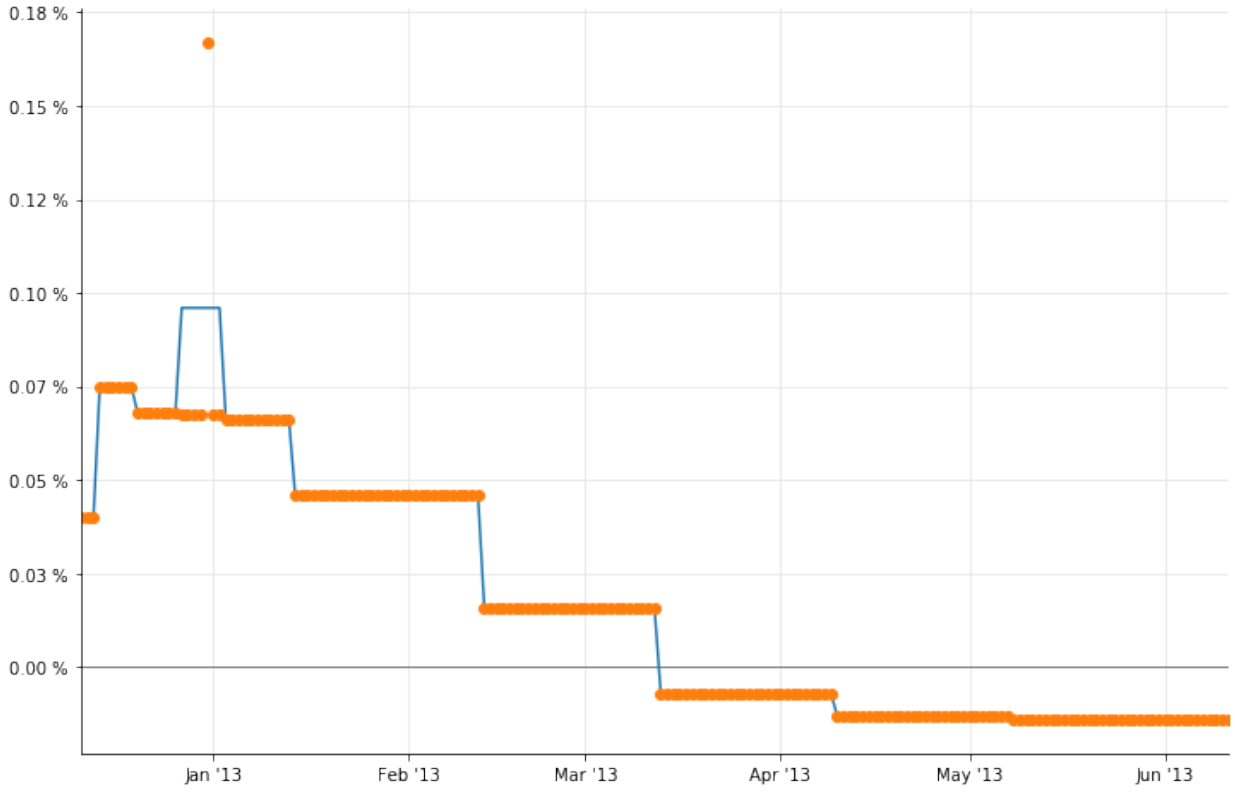
```
In [23]: t12 = eonia_curve_ff.dayCounter().yearFraction(d1,d2)
         t_j = eonia_curve_ff.dayCounter().yearFraction(Date(31,December,2012),
                                                        Date(2,January,2013))
         J = (F-F_1)*t12/t_j
         print(utils.format_rate(J,digits=3))

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 = [QuoteHandle(SimpleQuote(B))]
         jump_dates = [Date(31,December,2012)]
         eonia_curve_j = PiecewiseFlatForward(0, TARGET(),
                                              helpers, Actual365Fixed(),
                                              jumps, jump_dates)
```

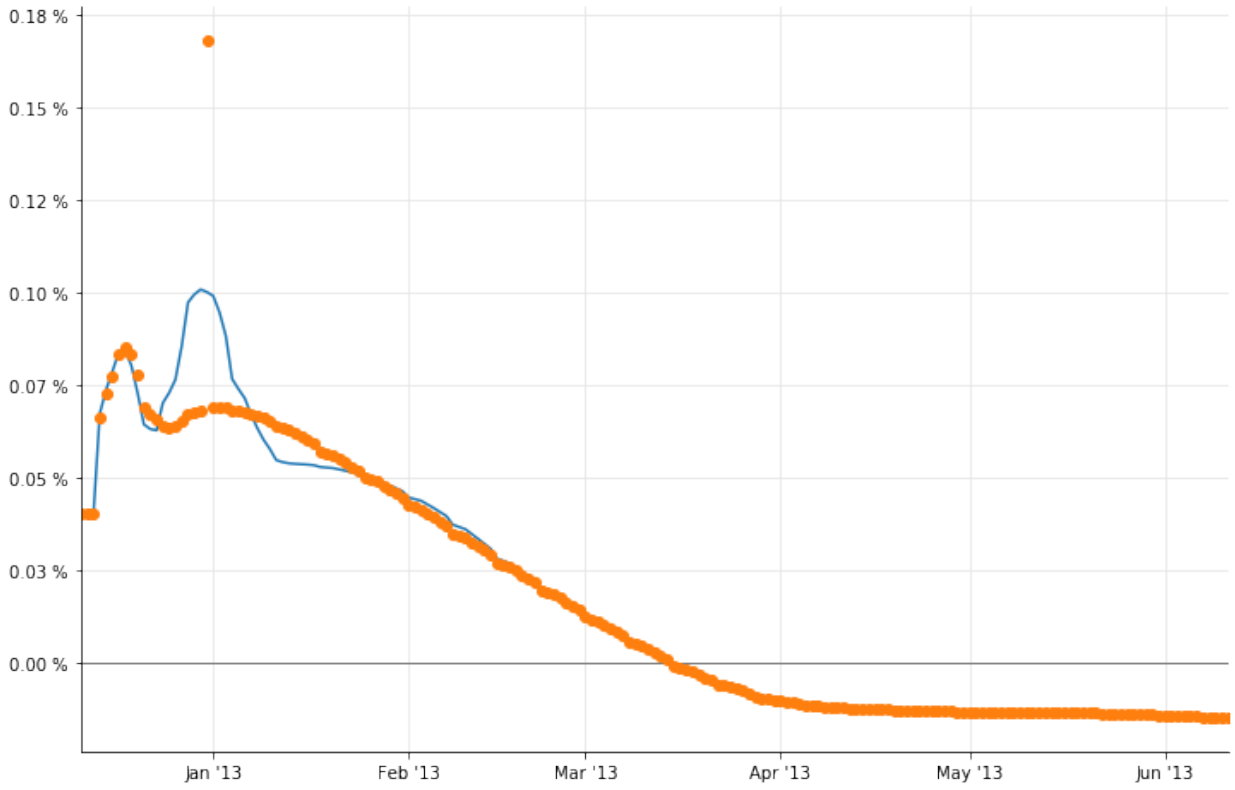
Retrieving daily overnight rates from the new curve and plotting them, we can see the jump quite clearly:

```
In [25]: rates_j = [ eonia_curve_j.forwardRate(d, TARGET().advance(d,1,Days),
                                                Actual360(), 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)
```



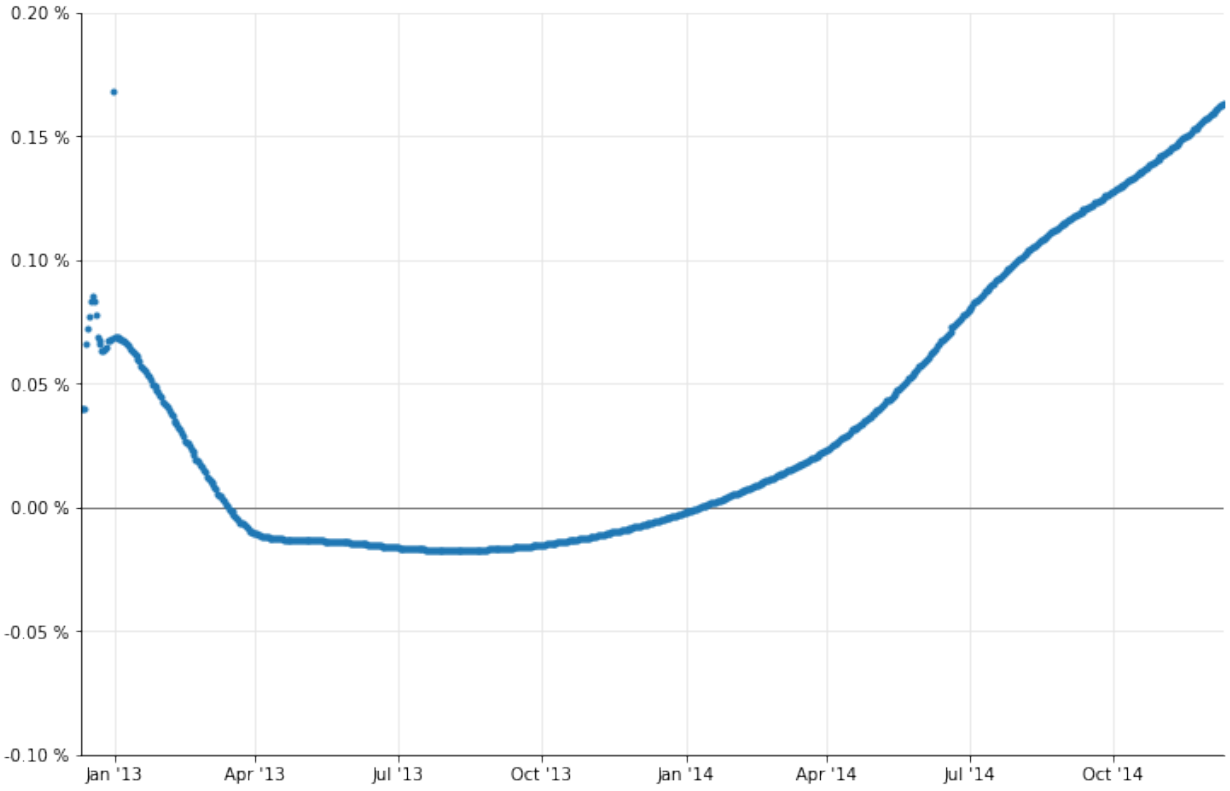
We can now go back to log-cubic discounts and add the jump.

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

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

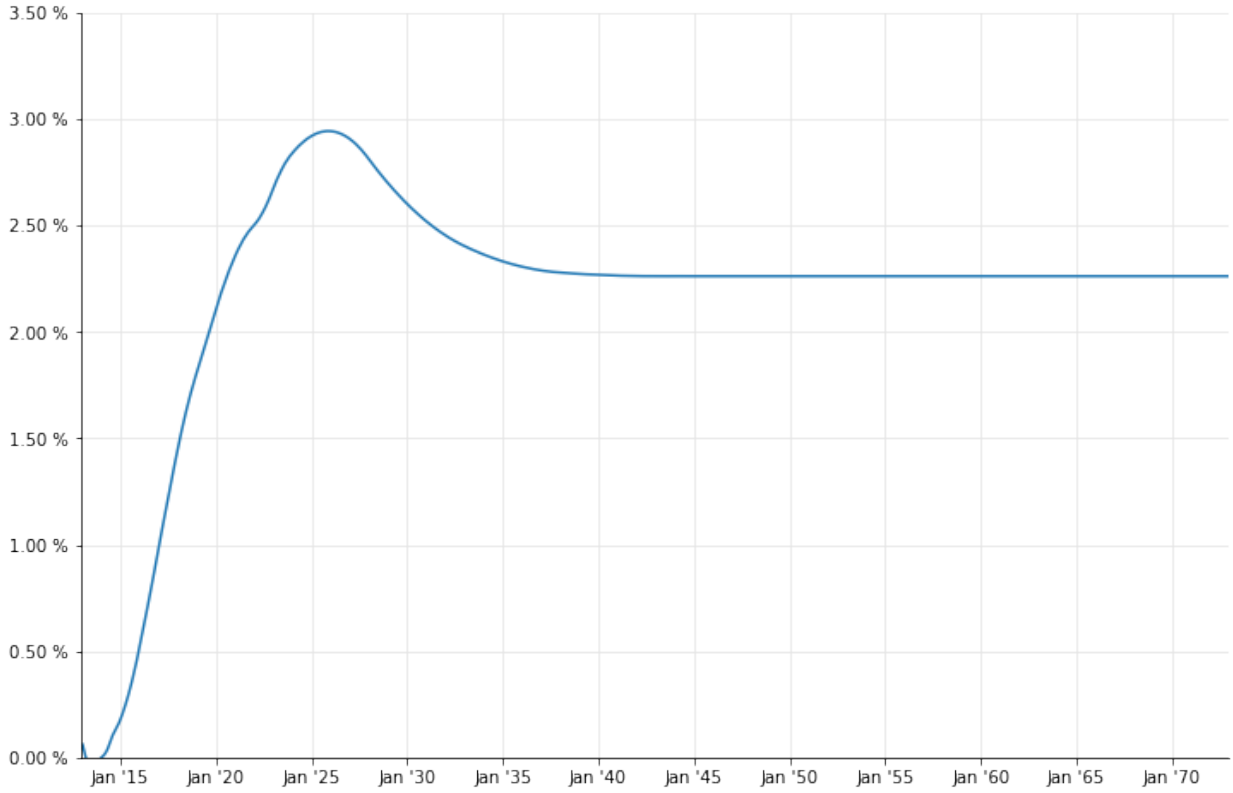


As you can see, the large bump is gone now. The two plots in figure 26 can be reproduced as follows (omitting the jump at the end of 2013 for brevity, and the flat forwards for clarity):

```
In [30]: dates = [ today+Period(i,Days) for i in range(0, 365*2+1) ]
         rates = [ eonia_curve.forwardRate(d, TARGET().advance(d,1,Days),
                                           Actual360(), 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+Period(i,Months) for i in range(0, 12*60+1) ]
         rates = [ eonia_curve.forwardRate(d, TARGET().advance(d,1,Days),
                                           Actual360(), 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

9. Euribor curve bootstrapping

In this notebook, I'll go over the second part of 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 numpy as np
        import utils
```

```
In [2]: from QuantLib import *
```

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

Discounting curve

The bootstrap of the Eonia curve was analyzed in another notebook, so I'll just instantiate the curve here without further explanation.

```
In [4]: eonia = Eonia()
```

```
In [5]: helpers = [ DepositRateHelper(QuoteHandle(SimpleQuote(rate/100)),
                                      Period(1,Days), fixingDays,
                                      TARGET(), Following, False, Actual360())
                    for rate, fixingDays in [(0.04, 0), (0.04, 1), (0.04, 2)] ]
```

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

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

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

In [9]: jumps = [QuoteHandle(SimpleQuote(math.exp(-J*2.0/360)))
                 for J in [0.00102, 0.00086]]
        jump_dates = [Date(31,December,2012), Date(31,December,2013)]

In [10]: eonia_curve = PiecewiseLogCubicDiscount(2, TARGET(), helpers,
                                                 Actual365Fixed(), jumps, jump_dates)
         eonia_curve.enableExtrapolation()
```

6-months Euribor

As we'll see, most of the Euribor curves for different tenors have their own quirks.

I'll start from the 6-months Euribor curve, which is somewhat simpler due to having a number of quoted rates directly available for bootstrapping. The first instrument used in the paper if the TOM 6-months FRA, which can be instantiated as a 6-months deposit with 3 fixing days; its rate (and those of all other FRAs) is retrieved from figure 6 in the paper.

```
In [11]: helpers = [ DepositRateHelper(QuoteHandle(SimpleQuote(0.312/100)),
                                       Period(6,Months), 3,
                                       TARGET(), Following, False, Actual360()) ]
```

Then comes a strip of 6-months FRA up to 2 years maturity:

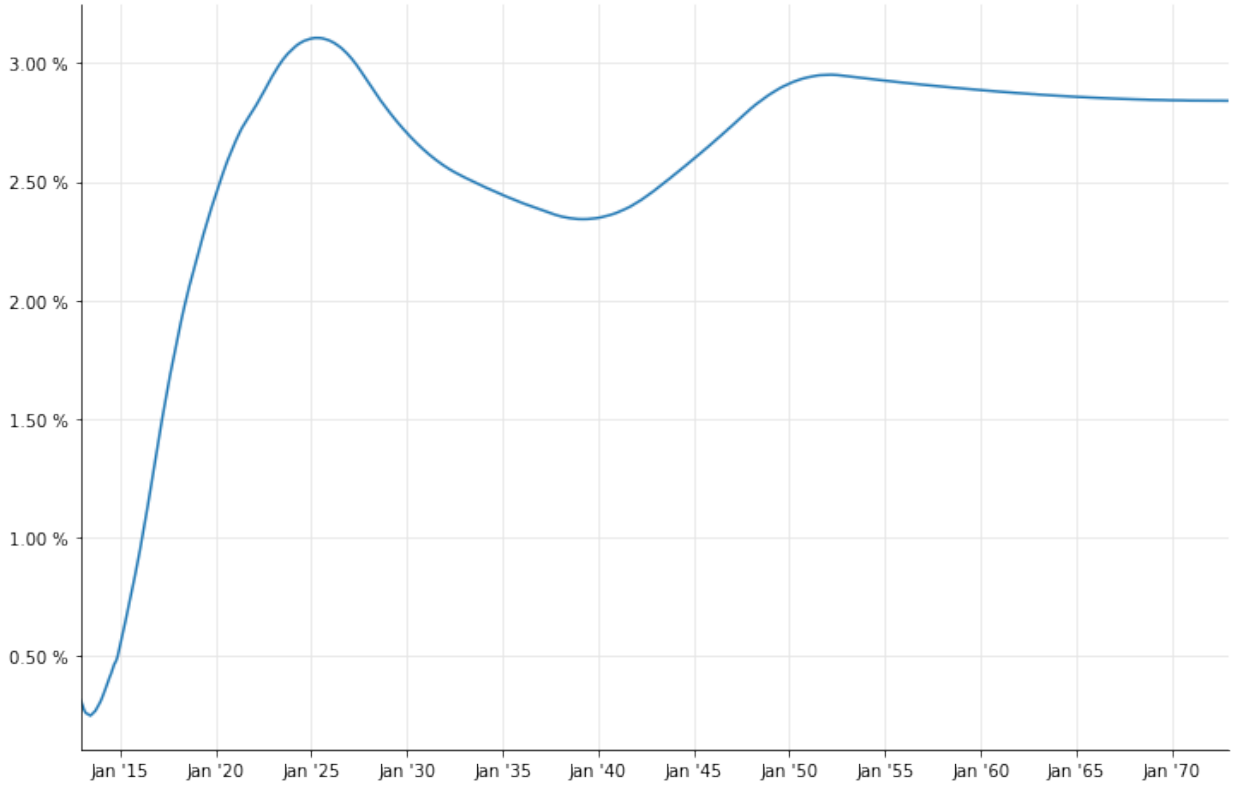
```
In [12]: euribor6m = Euribor6M()

In [13]: helpers += [ FraRateHelper(QuoteHandle(SimpleQuote(rate/100)),
                                    start, euribor6m)
                      for rate, start in [(0.293, 1), (0.272, 2), (0.260, 3),
                                          (0.256, 4), (0.252, 5), (0.248, 6),
                                          (0.254, 7), (0.261, 8), (0.267, 9),
                                          (0.279, 10), (0.291, 11), (0.303, 12),
                                          (0.318, 13), (0.335, 14), (0.352, 15),
                                          (0.371, 16), (0.389, 17), (0.409, 18)] ]
```

Finally, we have a series of swap rates with maturities from 3 to 60 years, listed in figure 9. As the paper explains, the curve being bootstrapped will be used only for forecasting the 6-months Euribor fixings paid by the floating leg; all the payments will be discounted by means of the OIS curve, which is wrapped in a `Handle` and passed as an extra argument to the `SwapRateHelper` constructor.

```
In [14]: discount_curve = RelinkableYieldTermStructureHandle()
         discount_curve.linkTo(eonia_curve)

In [15]: helpers += [ SwapRateHelper(QuoteHandle(SimpleQuote(rate/100)),
                                     Period(tenor, Years), TARGET(),
                                     Annual, Unadjusted,
                                     Thirty360(Thirty360.BondBasis),
                                     euribor6m, QuoteHandle(), Period(0, Days),
                                     discount_curve)
                      for rate, tenor in [(0.424, 3), (0.576, 4), (0.762, 5),
                                          (0.954, 6), (1.135, 7), (1.303, 8),
                                          (1.452, 9), (1.584, 10), (1.809, 12),
                                          (2.037, 15), (2.187, 20), (2.234, 25),
                                          (2.256, 30), (2.295, 35), (2.348, 40),
                                          (2.421, 50), (2.463, 60)] ]
```

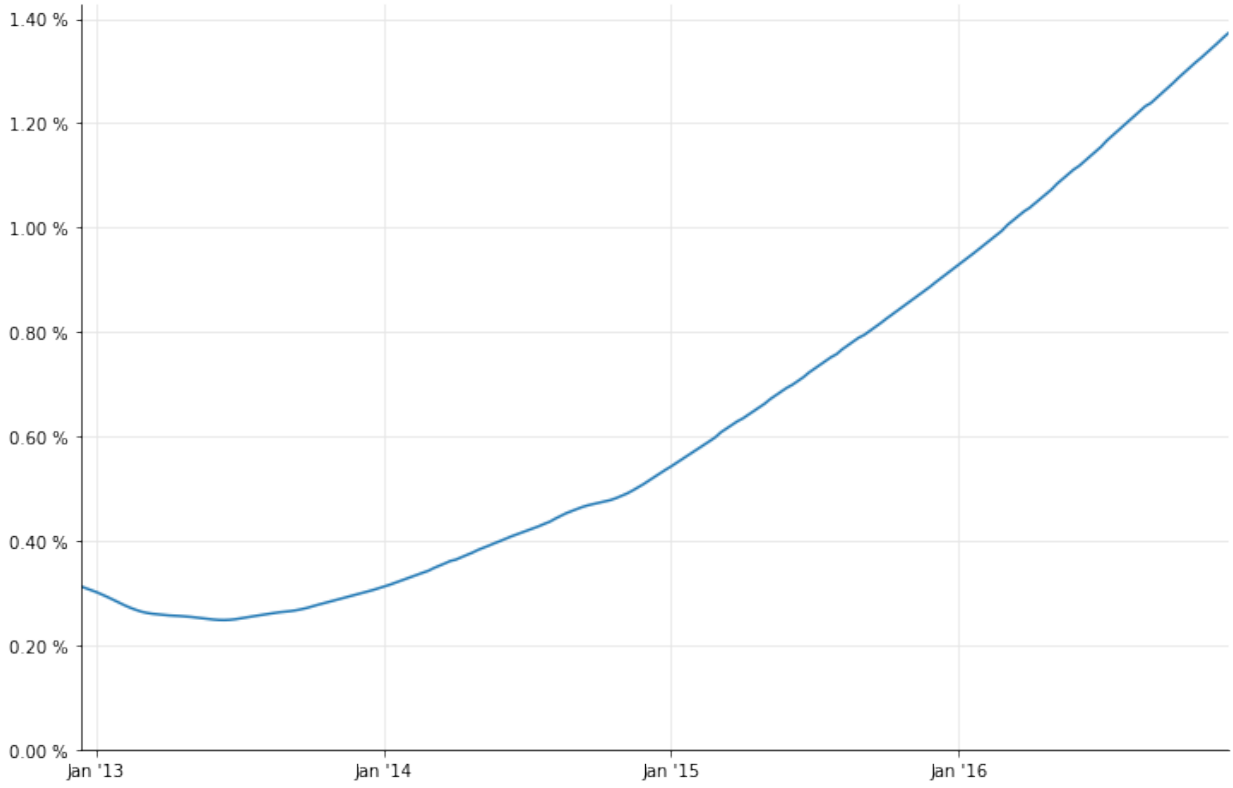
This will give us a decent Euribor curve, that we can display it by sampling 6-months forward rates at a number of dates.

```
In [16]: euribor6m_curve = PiecewiseLogCubicDiscount(2, TARGET(), helpers,
                                                    Actual365Fixed())
         euribor6m_curve.enableExtrapolation()

In [17]: spot = euribor6m_curve.referenceDate()
         dates = [ spot+Period(i,Months) for i in range(0, 60*12+1) ]
         rates = [ euribor6m_curve.forwardRate(d, euribor6m.maturityDate(d),
                                               Actual360(), Simple).rate()
                   for d in dates ]
         _, ax = utils.plot()
         utils.plot_curve(ax, dates, [(rates, '-')], format_rates=True)
```



This seems to work, and at the scale of the plot it seems to match figure 32 in the paper; but looking closely at the first part of the curve, you can see a glitch (some kind of dip) in the last part of 2014, when the FRA strip ends.

```
In [18]: dates = [ spot+Period(i,Weeks) for i in range(0, 52*4+1) ]
         rates = [ euribor6m_curve.forwardRate(d, euribor6m.maturityDate(d),
                                               Actual360(), Simple).rate()
                   for d in dates ]
         _, ax = utils.plot()
         utils.plot_curve(ax, dates, [(rates,'-')], ymin=0.0, format_rates=True)
```



Synthetic deposits

In short, the reason is that the short end of the curve (which is required for pricing FRAs; for instance, the 1x7 FRA required the discount factor at 1 month from now) is extrapolated backwards from the first quoted pillar at 6 months and is not quite correct. This leads to oscillations as soon as the curve is out of the tight strip of FRA quotes.

One way to correct this is to add synthetic deposits with short tenors, as explained in section 4.4.2 of the paper. To begin with, let's save the original curve to another variable for later comparison.

```
In [19]: euribor6m_curve_0 = euribor6m_curve
```

As detailed in the paper, one can model the basis between the Euribor market quotes and the corresponding OIS-based rates as a polynomial; that is, following equation 88,

$$R_x(T_1, T_2)\tau(T_1, T_2) = R_{on}(T_1, T_2)\tau(T_1, T_2) + \Delta(T_1, T_2)$$

In the paper, the expression for $\Delta(T_1, T_2)$ is given by equation 90, that is,

$$\Delta(T_1, T_2) = \alpha \cdot (T_2 - T_1) + \frac{1}{2}\beta \cdot (T_2 - T_1)^2 + \frac{1}{3}\gamma \cdot (T_2 - T_1)^3 + \dots$$

However, the above leads to problems when trying to solve for more than one coefficient. Following a later formulation[1], I'll express the instantaneous basis instead as

$$\delta(t) = \alpha + \beta \cdot t + \gamma \cdot t^2 + \dots$$

which leads to

$$\Delta(T_1, T_2) = \int_{T_1}^{T_2} \delta(t) = \alpha \cdot (T_2 - T_1) + \frac{1}{2}\beta \cdot (T_2^2 - T_1^2) + \frac{1}{3}\gamma \cdot (T_2^3 - T_1^3) + \dots$$

Once the basis is known, we can calculate synthetic deposit rates $R(0,T)$ for any maturity T.

Depending on how many polynomial coefficients we want to determine, we'll need a corresponding number of market quotes; by replacing their values and those of the OIS rates in equation 88 we can solve for α, β and any other coefficient.

For a constant polynomial, we'll need one quote to determine α; we can use the TOM 6-months deposit that the Euribor curve reprices exactly.

```
In [20]: d = TARGET().advance(spot, 1, Days)
         F_x = euribor6m_curve_0.forwardRate(d, TARGET().advance(d, 6, Months),
                                             Actual360(), Simple).rate()
         F_on = eonia_curve.forwardRate(d, TARGET().advance(d, 6, Months),
                                        Actual360(), Simple).rate()
         day_counter = euribor6m.dayCounter()
         T_x = day_counter.yearFraction(d, TARGET().advance(d, 6, Months))
         alpha = (F_x - F_on)
         print(alpha)

Out[20]: 0.002949286970370156
```

From the basis, we can instantiate synthetic deposits for a number of maturities below 6 months…

```
In [21]: synth_helpers = []
         for n, units in [(1,Days), (1,Weeks), (2,Weeks), (3,Weeks),
                          (1, Months), (2, Months), (3, Months),
                          (4, Months), (5, Months)]:
             t = day_counter.yearFraction(spot, TARGET().advance(spot, n, units))
             F_on = eonia_curve.forwardRate(spot, TARGET().advance(spot, n, units),
                                            Actual360(), Simple).rate()
             F = F_on + alpha
             print("{0}: {1}".format(Period(n,units), utils.format_rate(F, 4)))
             synth_helpers.append(DepositRateHelper(QuoteHandle(SimpleQuote(F)),
                                  Period(n, units), 2,
```

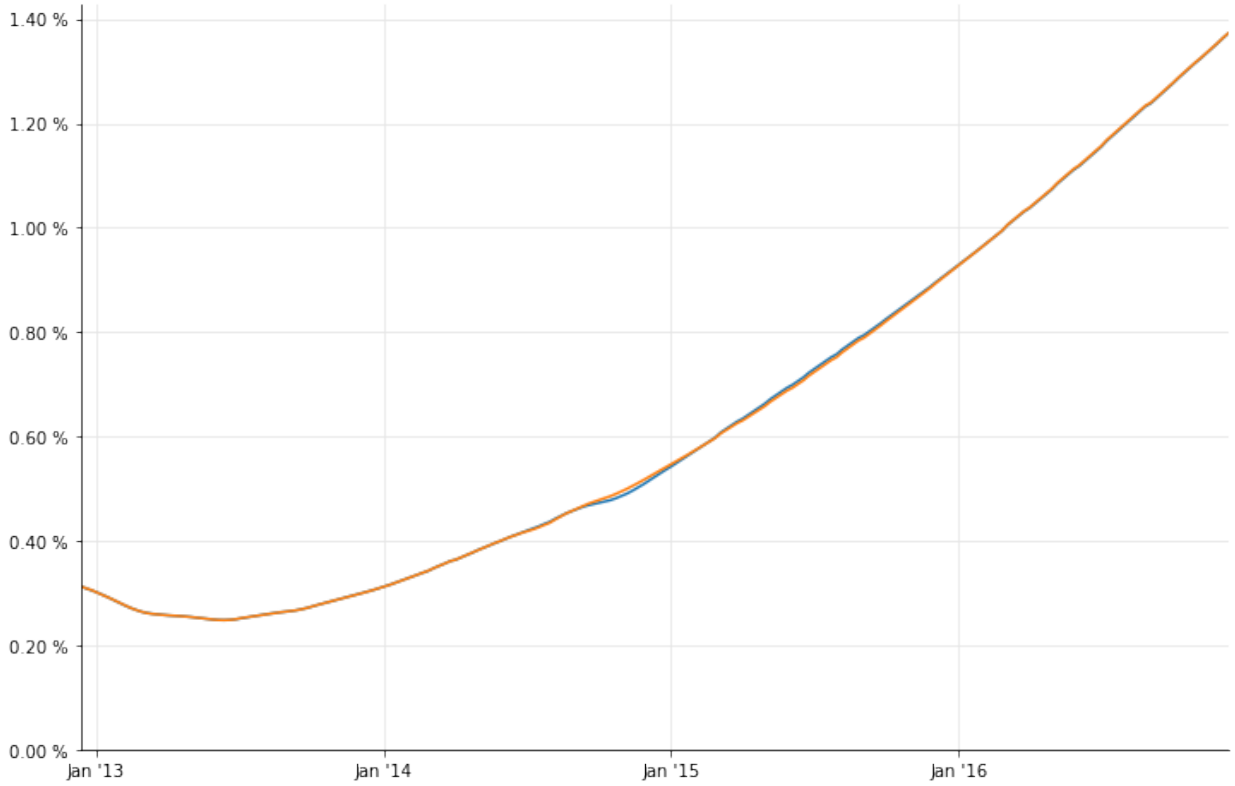
[1] https://speakerdeck.com/nando1970/eonia-jumps-and-proper-euribor-forwarding

```
                                    TARGET(), Following, False, Actual360()))
```

```
Out[21]: 1D: 0.3349 %
         1W: 0.3649 %
         2W: 0.3639 %
         3W: 0.3729 %
         1M: 0.3689 %
         2M: 0.3559 %
         3M: 0.3419 %
         4M: 0.3272 %
         5M: 0.3188 %
```

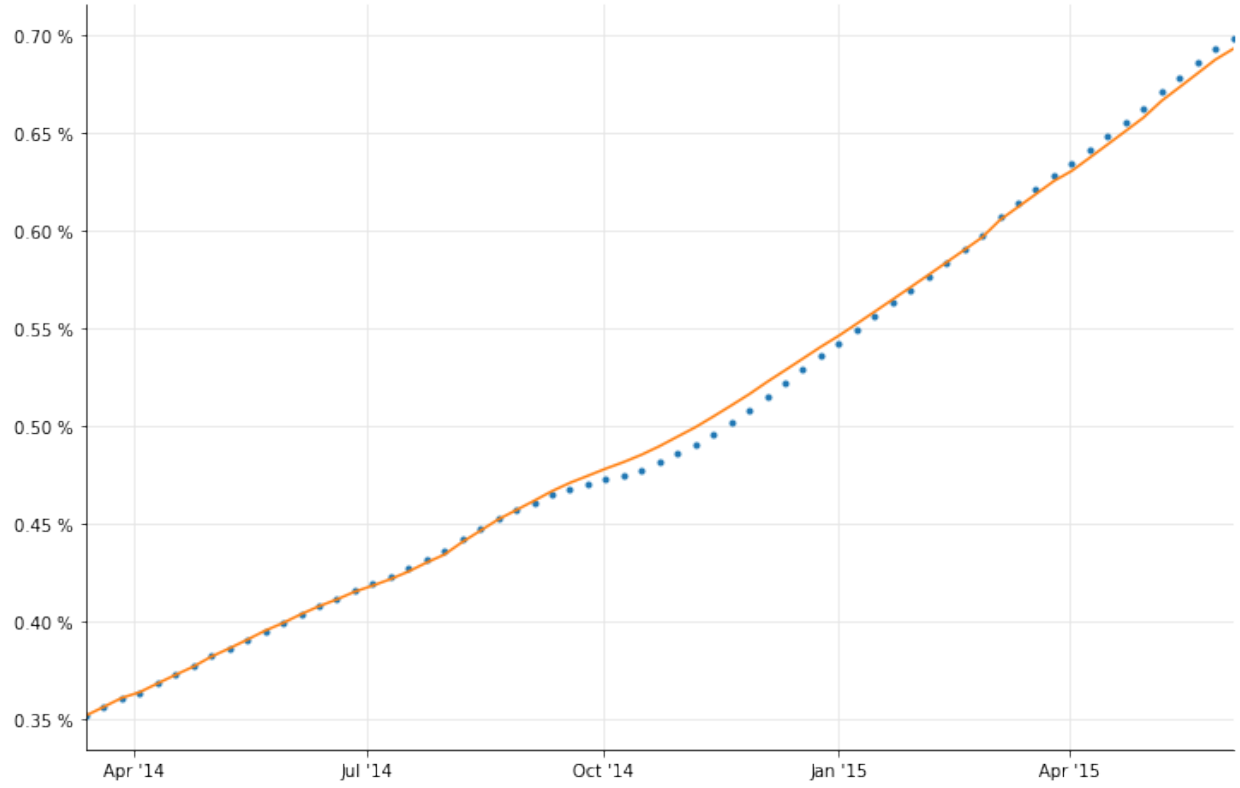
...after which we can create a new curve, which seems to have a smaller dip:

```
In [22]: euribor6m_curve = PiecewiseLogCubicDiscount(2, TARGET(),
                                                     helpers+synth_helpers,
                                                     Actual365Fixed())
         euribor6m_curve.enableExtrapolation()

In [23]: dates = [ spot+Period(i,Weeks) for i in range(0, 52*4+1) ]
         rates_0 = [ euribor6m_curve_0.forwardRate(d, euribor6m.maturityDate(d),
                                                   Actual360(), Simple).rate()
                     for d in dates ]
         rates = [ euribor6m_curve.forwardRate(d, euribor6m.maturityDate(d),
                                               Actual360(), Simple).rate()
                   for d in dates ]
         _, ax = utils.plot()
         utils.plot_curve(ax, dates, [(rates_0,'-'), (rates,'-')], ymin=0.0, format_rates=True)
```



By choosing to sample at different dates, we can zoom into the affected area. The original curve is the dotted line; the new curve is the solid one.

```
In [24]: dates = [ spot+Period(i,Weeks) for i in range(65, 130) ]
         rates_0 = [ euribor6m_curve_0.forwardRate(d, euribor6m.maturityDate(d),
                                                   Actual360(), Simple).rate()
                     for d in dates ]
         rates = [ euribor6m_curve.forwardRate(d, euribor6m.maturityDate(d),
                                               Actual360(), Simple).rate()
                   for d in dates ]
         _, ax = utils.plot()
         utils.plot_curve(ax, dates, [(rates_0,'.'), (rates,'-')], format_rates=True)
```



If we wanted to determine more coefficients for the basis, we'd have to select more quotes and solve a linear system. For instance, to determine both α and β, we can use the TOM 6-months and the 1x7 FRAs:

```
In [25]: start = TARGET().advance(spot, 1, Days)
         end = TARGET().advance(start, 6, Months)
         F_x = euribor6m_curve_0.forwardRate(start, end, Actual360(), Simple).rate()
         F_on = eonia_curve.forwardRate(start, end, Actual360(), Simple).rate()
         T_x0 = day_counter.yearFraction(start, end)
         Delta0 = F_x - F_on

         start = TARGET().advance(spot, 1, Months)
         end = TARGET().advance(start, 6, Months)
         F_x = euribor6m_curve_0.forwardRate(start, end, Actual360(), Simple).rate()
         F_on = eonia_curve.forwardRate(start, end, Actual360(), Simple).rate()
         T_x1 = day_counter.yearFraction(start, end)
         Delta1 = F_x - F_on

         t1 = day_counter.yearFraction(spot, start)
         t2 = day_counter.yearFraction(spot, end)

         L = np.array([[T_x0, 0.5*T_x0**2], [T_x1, 0.5*(t2**2-t1**2)]])
         b = np.array([Delta0*T_x0, Delta1*T_x1])
```

```
alpha, beta = np.linalg.solve(L,b)
print(alpha)
print(beta)
```

```
Out[25]: 0.0030464085692271255
         -0.0003842173141594401
```

Again, we can create synthetic deposits...

```
In [26]: synth_helpers = []
         for n, units in [(1,Days), (1,Weeks), (2,Weeks), (3,Weeks),
                          (1, Months), (2, Months), (3, Months),
                          (4, Months), (5, Months)]:
             t = day_counter.yearFraction(spot, TARGET().advance(spot, n, units))
             F_on = eonia_curve.forwardRate(spot, TARGET().advance(spot, n, units),
                                            Actual360(), Simple).rate()
             F = F_on + alpha + 0.5*beta*t
             print("{0}: {1}".format(Period(n,units), utils.format_rate(F, 4)))
             synth_helpers.append(DepositRateHelper(QuoteHandle(SimpleQuote(F)),
                                  Period(n, units), 2,
                                  TARGET(), Following, False, Actual360()))
```

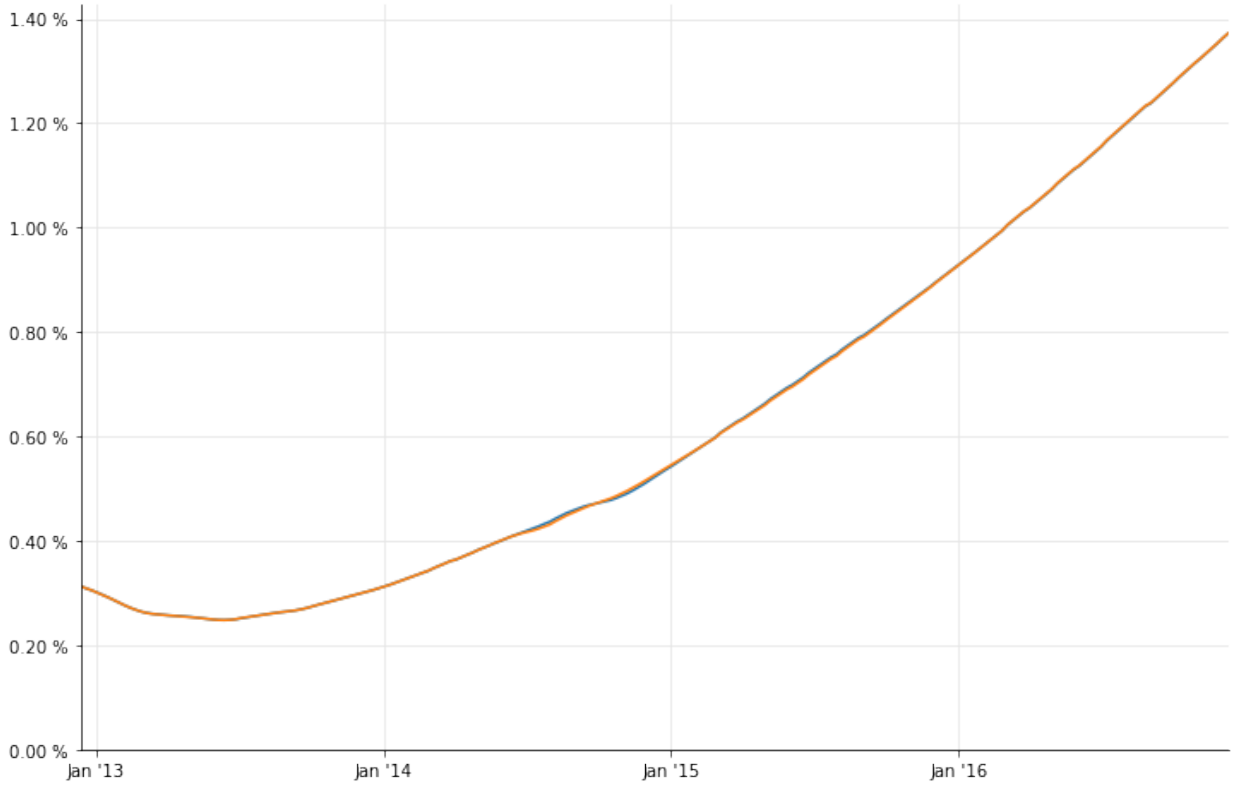
```
Out[26]: 1D: 0.3446 %
         1W: 0.3743 %
         2W: 0.3729 %
         3W: 0.3815 %
         1M: 0.3769 %
         2M: 0.3623 %
         3M: 0.3468 %
         4M: 0.3304 %
         5M: 0.3204 %
```

...and build a new curve. I'll leave it to you to decide whether this is an improvement over the degree-1 polynomial basis.

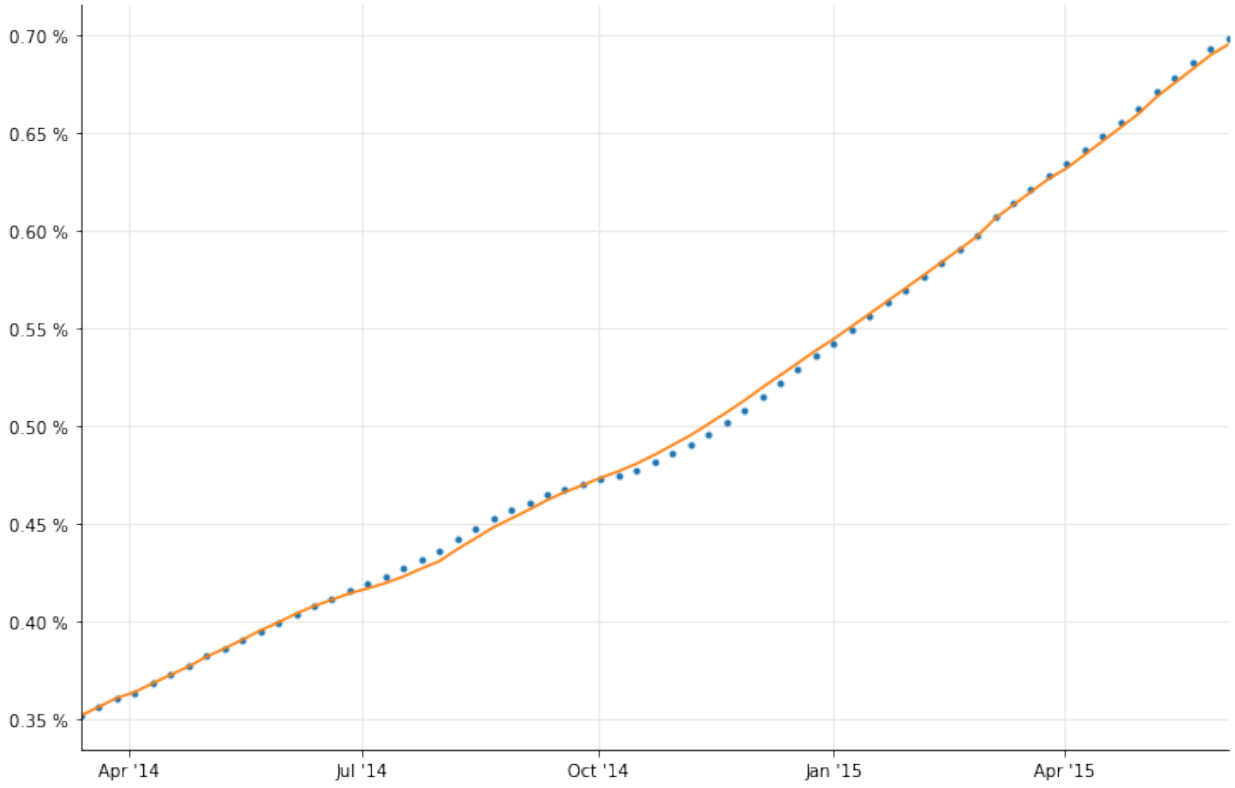
```
In [27]: euribor6m_curve = PiecewiseLogCubicDiscount(2, TARGET(),
                                                     helpers+synth_helpers,
                                                     Actual365Fixed())
         euribor6m_curve.enableExtrapolation()
```

```
In [28]: dates = [ spot+Period(i,Weeks) for i in range(0, 52*4+1) ]
         rates_0 = [ euribor6m_curve_0.forwardRate(d, euribor6m.maturityDate(d),
                                                   Actual360(), Simple).rate()
                     for d in dates ]
         rates = [ euribor6m_curve.forwardRate(d, euribor6m.maturityDate(d),
                                               Actual360(), Simple).rate()
```

```
                for d in dates ]
        _, ax = utils.plot()
        utils.plot_curve(ax, dates, [(rates_0,'-'), (rates,'-')], ymin=0.0, format_rates=True)
```



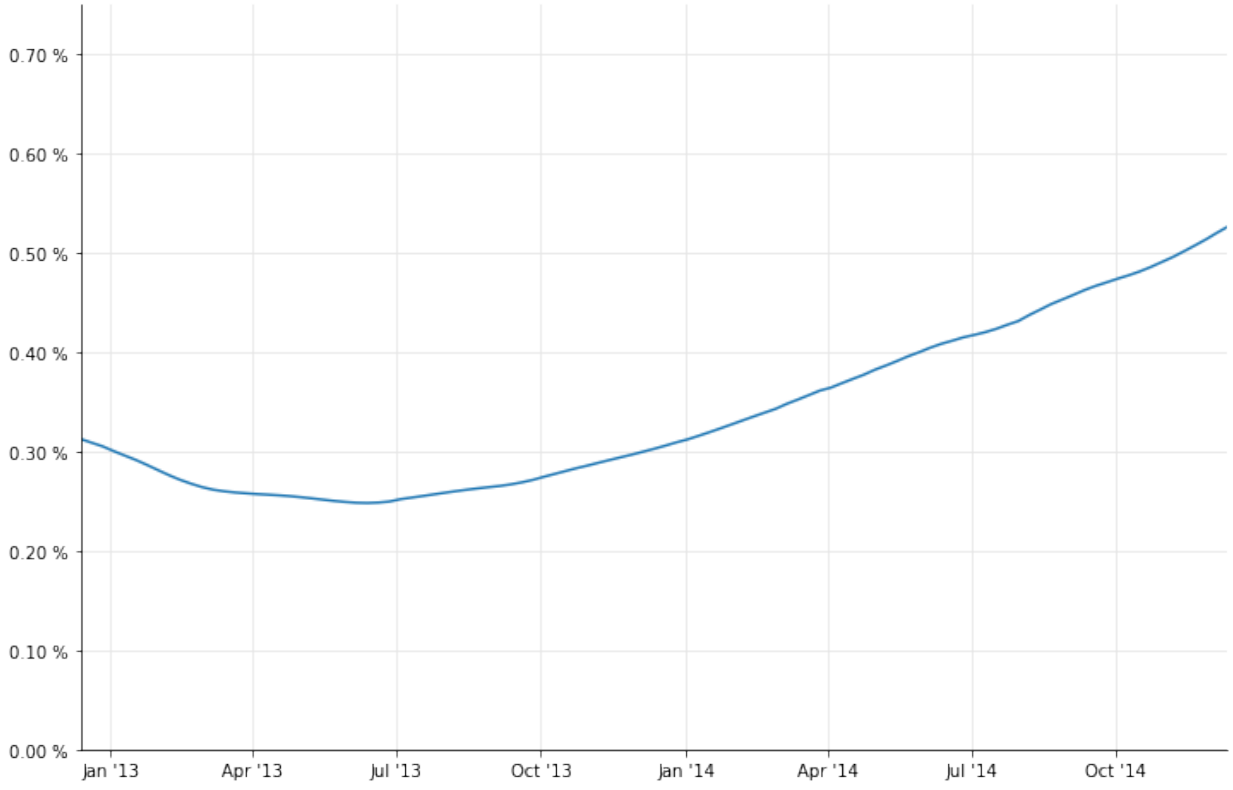

```
In [29]: dates = [ spot+Period(i,Weeks) for i in range(65, 130) ]
         rates_0 = [ euribor6m_curve_0.forwardRate(d, euribor6m.maturityDate(d),
                                                   Actual360(), Simple).rate()
                     for d in dates ]
         rates = [ euribor6m_curve.forwardRate(d, euribor6m.maturityDate(d),
                                               Actual360(), Simple).rate()
                   for d in dates ]
         _, ax = utils.plot()
         utils.plot_curve(ax, dates, [(rates_0,'.'), (rates,'-')], format_rates=True)
```



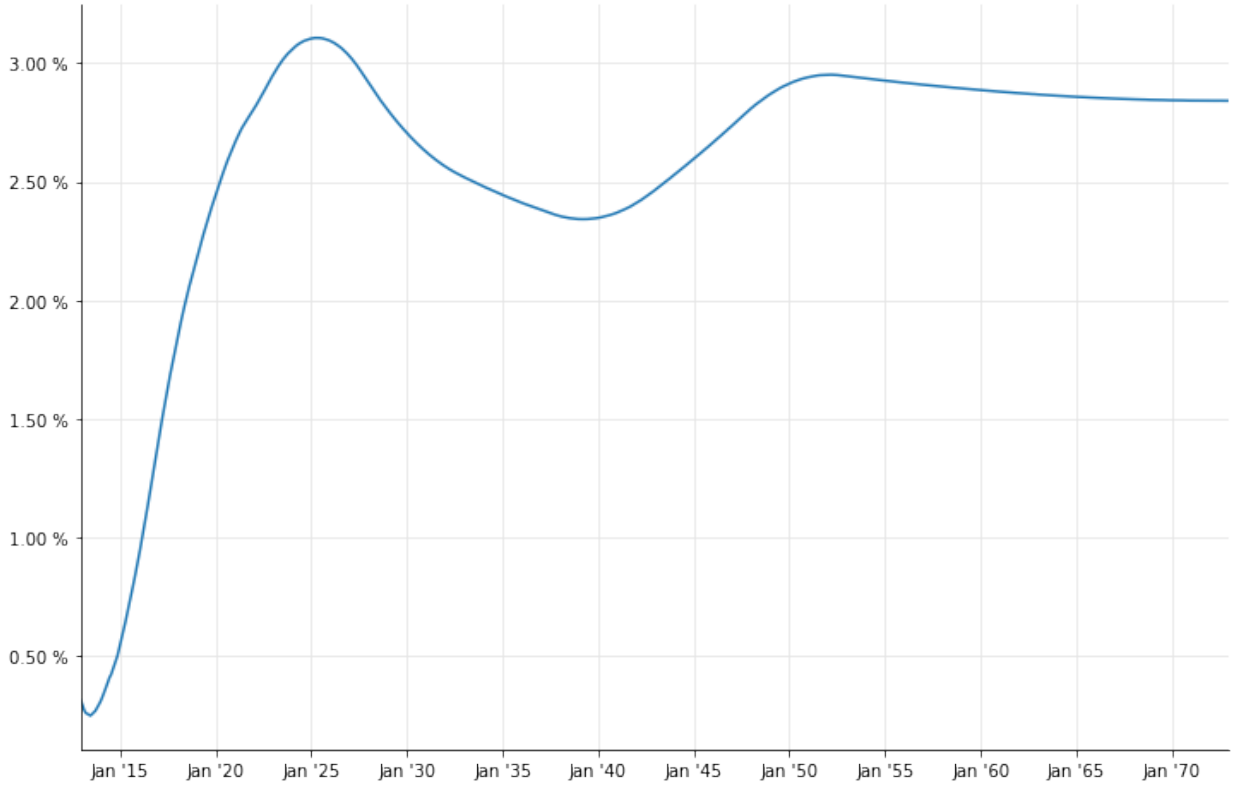
One thing to note: the values I'm getting for the synthetic deposits are not the same as those reported by the paper in figure 17. I still haven't found the reason for the discrepancy.

As for figure 32 in the paper, here's how we can reproduce it:

```
In [30]: spot = euribor6m_curve.referenceDate()
         dates = [ spot+Period(i,Weeks) for i in range(0, 2*52+1) ]
         rates = [ euribor6m_curve.forwardRate(d, euribor6m.maturityDate(d),
                                               Actual360(), Simple).rate()
                   for d in dates ]
         _, ax = utils.plot()
         utils.plot_curve(ax, dates, [(rates, '-')], ymin=0.0, ymax=0.0075, format_rates=True)
```




```
In [31]: spot = euribor6m_curve.referenceDate()
         dates = [ spot+Period(i,Months) for i in range(0, 60*12+1) ]
         rates = [ euribor6m_curve.forwardRate(d, euribor6m.maturityDate(d),
                                               Actual360(), Simple).rate()
                   for d in dates ]
         _, ax = utils.plot()
         utils.plot_curve(ax, dates, [(rates, '-')], format_rates=True)
```



12-months Euribor

For the 12-months curve, we'll start with the quoted 12-months deposit and 12x24 FRA (see figures 4 and 5).

```
In [32]: euribor12m = Euribor1Y()
         helpers = [ DepositRateHelper(QuoteHandle(SimpleQuote(0.54/100)),
                                       Period(12,Months), 2,
                                       TARGET(), Following, False, Actual360()) ]
         helpers += [ FraRateHelper(QuoteHandle(SimpleQuote(0.5070/100)),
                                    12, euribor12m) ]
```

Unfortunately, there are no quoted swap rates against 12-months Euribor. However, the market quotes 6- vs 12-months basis swaps; and more importantly, it quotes them as a portfolio of two IRS, payer and receiver, both accruing annual fixed coupons against Euribor 6M and 12M, respectively. The spread between the two fixed legs is quoted so that it sets the NPV of the portfolio at zero.

Given that the market also quotes the fair fixed rate for one of the two swaps, i.e., the one paying a fixed rate against Euribor 6M, it's straightforward to see that the fair fixed rate for the swap against Euribor 12M can be obtained by just adding the 6M rate to the basis spread: that is, if the NPV of a swap S_1 paying K against Euribor 6M is 0, and if the NPV of the portfolio of S_1 minus another swap S_2 paying $K + S$ against Euribor 12M is also 0, then the NPV of S_2 must be 0 as well.

This gives us quoted swap rates against Euribor 12M up to 30 years, which is the longest quoted maturity for basis swaps. The data are from figures 9 and 15.

```
In [33]: helpers += [
             SwapRateHelper(QuoteHandle(SimpleQuote((rate+basis)/100)),
                            Period(tenor, Years), TARGET(),
                            Annual, Unadjusted, Thirty360(Thirty360.BondBasis),
                            euribor12m, QuoteHandle(), Period(0, Days),
                            discount_curve)
             for rate, basis, tenor in [(0.424, 0.179, 3), (0.576, 0.164, 4),
                                        (0.762, 0.151, 5), (0.954, 0.139, 6),
                                        (1.135, 0.130, 7), (1.303, 0.123, 8),
                                        (1.452, 0.118, 9), (1.584, 0.113, 10),
                                        (1.809, 0.106, 12), (2.037, 0.093, 15),
                                        (2.187, 0.080, 20), (2.234, 0.072, 25),
                                        (2.256, 0.066, 30)] ]
```

Again, we'll be using synthetic helpers to improve the shape of the short end of the curve. The same procedure we used for the Euribor 6M curve lets us create deposits with a number of maturities below 1 year; I'll skip the calculation and just create helpers with the the resulting rates as reported by the paper.

```
In [34]: synth_helpers = [
             DepositRateHelper(QuoteHandle(SimpleQuote(rate/100)),
                               Period(*tenor), 2,
                               TARGET(), Following, False, Actual360())
             for rate, tenor in [(0.6537, (1,Months)), (0.6187, (3,Months)),
                                 (0.5772, (6,Months)), (0.5563, (9,Months))] ]
```

It is also possible to build synthetic FRAs: their construction is explained in the paper. I'll leave it to a later version of this chapter; for the time being, I'll just add the finished helpers.

```
In [35]: synth_helpers += [
             FraRateHelper(QuoteHandle(SimpleQuote(rate/100)),
                           months_to_start, euribor12m)
             for rate, months_to_start in [(0.4974, 3), (0.4783, 6), (0.4822, 9),
                                           (0.5481, 15), (0.6025, 18)] ]
```

Finally, we can extend the long end of the curve by creating synthetic swaps with maturities above 30 years. To calculate their rates, we add the swap rates against Euribor 6M (quoted up to 60 years) to the last quoted basis spread.

```
In [36]: last_basis = 0.066
         synth_helpers += [
             SwapRateHelper(QuoteHandle(SimpleQuote((rate+last_basis)/100)),
                            Period(tenor, Years), TARGET(),
                            Annual, Unadjusted, Thirty360(Thirty360.BondBasis),
                            euribor12m, QuoteHandle(), Period(0, Days),
                            discount_curve)
             for rate, tenor in [(2.295, 35), (2.348, 40),
                                 (2.421, 50), (2.463, 60)] ]
```

Bootstrapping over the whole set of real and synthetic quotes gives us our final Euribor 12M curve:

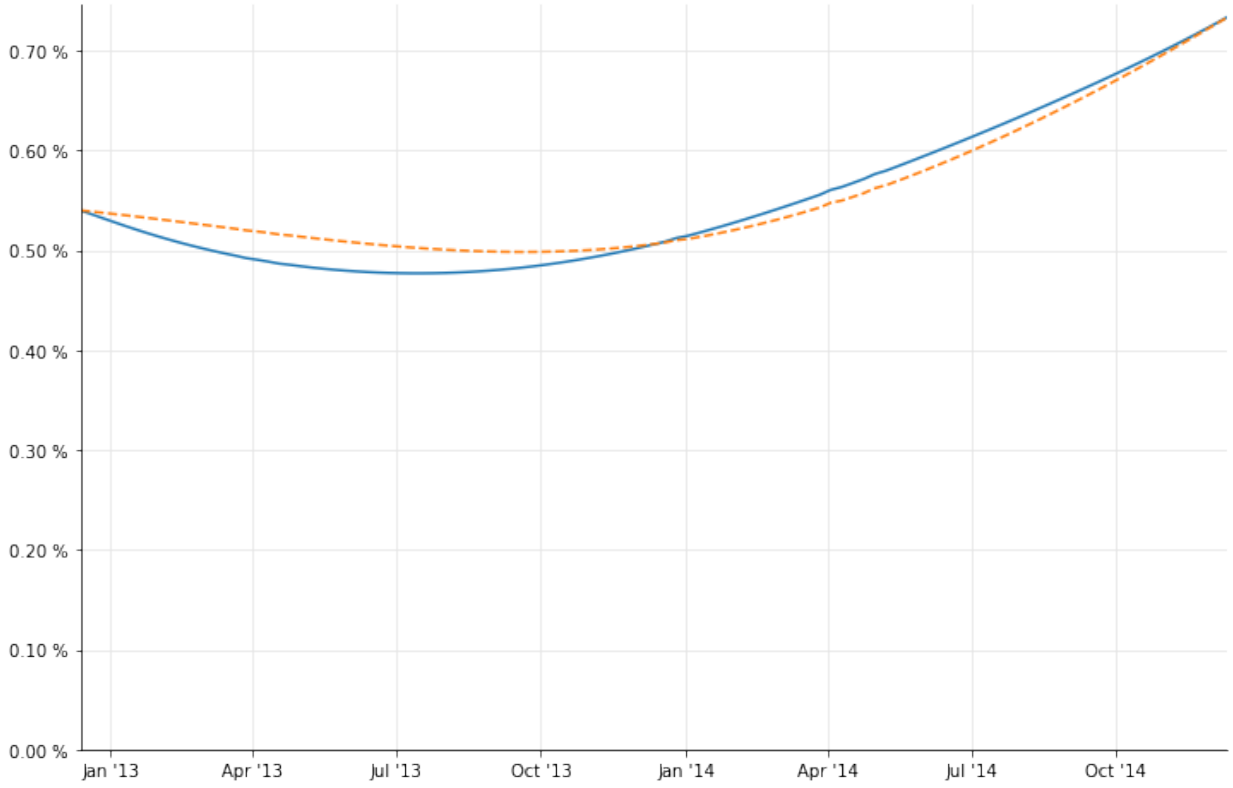
```
In [37]: euribor12m_curve = PiecewiseLogCubicDiscount(2, TARGET(),
                                                      helpers+synth_helpers,
                                                      Actual365Fixed())
         euribor12m_curve.enableExtrapolation()
```

For comparison, we can build another one excluding the synthetic helpers.

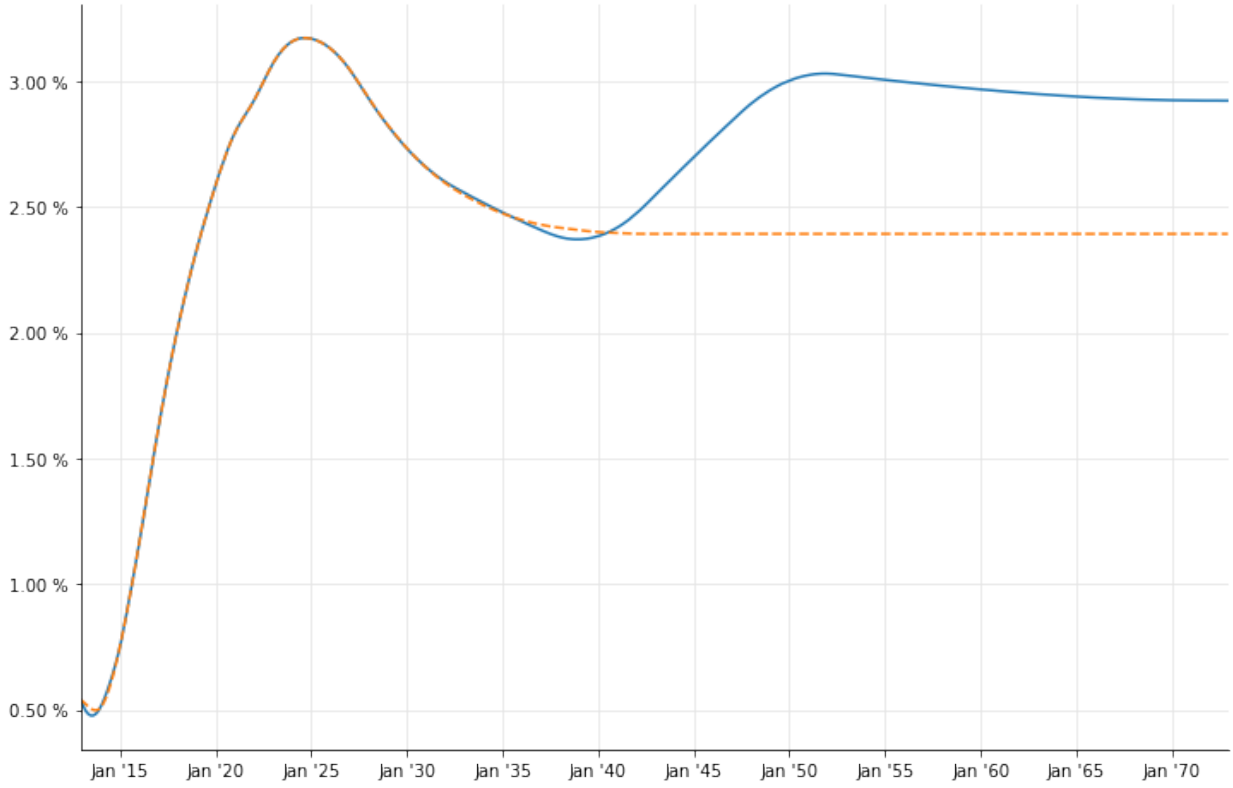
```
In [38]: euribor12m_curve_0 = PiecewiseLogCubicDiscount(2, TARGET(), helpers,
                                                        Actual365Fixed())
         euribor12m_curve_0.enableExtrapolation()
```

The two curves are plotted together in the two following graphs, which also reproduce figure 34 in the paper. The solid line corresponds to the complete curve, and the dashed line to the curve without the synthetic helpers. The differences are obvious, both in the short and in the long end.

```
In [39]: spot = euribor12m_curve.referenceDate()
         dates = [ spot+Period(i,Weeks) for i in range(0, 2*52+1) ]
         rates_0 = [ euribor12m_curve_0.forwardRate(d, euribor12m.maturityDate(d),
                                                    Actual360(), Simple).rate()
                     for d in dates ]
         rates = [ euribor12m_curve.forwardRate(d, euribor12m.maturityDate(d),
                                                Actual360(), Simple).rate()
                   for d in dates ]
         _, ax = utils.plot()
         utils.plot_curve(ax, dates, [(rates, '-'),(rates_0, '--')], ymin=0.0, format_rates=Tr\
ue)
```




```
In [40]: dates = [ spot+Period(i,Months) for i in range(0, 60*12+1) ]
         rates_0 = [ euribor12m_curve_0.forwardRate(d, euribor12m.maturityDate(d),
                                                    Actual360(), Simple).rate()
                     for d in dates ]
         rates = [ euribor12m_curve.forwardRate(d, euribor12m.maturityDate(d),
                                                Actual360(), Simple).rate()
                   for d in dates ]
         _, ax = utils.plot()
         utils.plot_curve(ax, dates, [(rates, '-'), (rates_0, '--')], format_rates=True)
```



3-months Euribor

For the 3-months Euribor, we can use a strip of very liquid futures after the 3-months deposit; their rates are listed in figures 7 and 4, respectively.

```
In [41]: euribor3m = Euribor3M()
         helpers = [ DepositRateHelper(QuoteHandle(SimpleQuote(0.179/100)),
                                       Period(3,Months), 3,
                                       TARGET(), Following, False, Actual360()) ]
         helpers += [
             FuturesRateHelper(QuoteHandle(SimpleQuote(100-rate)),
                               start_date, euribor3m, QuoteHandle())
             for rate, start_date in [(0.1775, Date(19, December, 2012)),
                                      (0.1274, Date(20, March, 2013)),
                                      (0.1222, Date(19, June, 2013)),
                                      (0.1269, Date(18, September, 2013)),
                                      (0.1565, Date(18, December, 2013)),
                                      (0.1961, Date(19, March, 2014)),
                                      (0.2556, Date(18, June, 2014)),
                                      (0.3101, Date(17, September, 2014))] ]
```

For the swaps, we combine quotes for the swaps against 6-months Euribor with quotes for the 3-months against 6-months basis swap, like we did for the 12-months curve; basis swap quotes for this

tenor are available up to 50 years, as shown in figure 15. In this case, though, the fixed rate against Euribor 3M is lower than the one against Euribor 6M; therefore, the basis must be subtracted from the quoted rate:

```
In [42]: helpers += [
             SwapRateHelper(QuoteHandle(SimpleQuote((rate-basis)/100)),
                            Period(tenor, Years), TARGET(),
                            Annual, Unadjusted, Thirty360(Thirty360.BondBasis),
                            euribor3m, QuoteHandle(), Period(0, Days),
                            discount_curve)
             for rate, basis, tenor in [(0.424, 0.1395, 3), (0.576, 0.1390, 4),
                                        (0.762, 0.1395, 5), (0.954, 0.1375, 6),
                                        (1.135, 0.1350, 7), (1.303, 0.1320, 8),
                                        (1.452, 0.1285, 9), (1.584, 0.1250, 10),
                                        (1.809, 0.1170, 12), (2.037, 0.1045, 15),
                                        (2.187, 0.0885, 20), (2.234, 0.0780, 25),
                                        (2.256, 0.0700, 30), (2.348, 0.0600, 40),
                                        (2.421, 0.0540, 50)] ]
```

Again, synthetic deposit rates can be calculated and added for short maturities...

```
In [43]: synth_helpers = [
             DepositRateHelper(QuoteHandle(SimpleQuote(rate/100)),
                               Period(*tenor), 2,
                               TARGET(), Following, False, Actual360())
             for rate, tenor in [(0.1865, (2,Weeks)), (0.1969, (3,Weeks)),
                                 (0.1951, (1,Months)), (0.1874, (2,Months))] ]
```

...and again, we can add a few synthetic swaps where quotes for the 3-months versus 6-months Euribor are not available. We can calculate a quote for the 35-years basis swap by interpolating between the 30- and 40-years quotes, and one for the 60-years swap by extrapolating the 50-years quote flatly, like we did for the 12-months Euribor. Note that in this case, the authors of the paper choose instead to extrapolate the previous quotes linearly; anyway, this gives a difference of less than half a basis point.

```
In [44]: synth_helpers += [
             SwapRateHelper(QuoteHandle(SimpleQuote((rate-basis)/100)),
                            Period(tenor, Years), TARGET(),
                            Annual, Unadjusted, Thirty360(Thirty360.BondBasis),
                            euribor3m, QuoteHandle(), Period(0, Days),
                            discount_curve)
             for rate, basis, tenor in [(2.295, 0.0650, 35), (2.463, 0.0540, 60)] ]
```

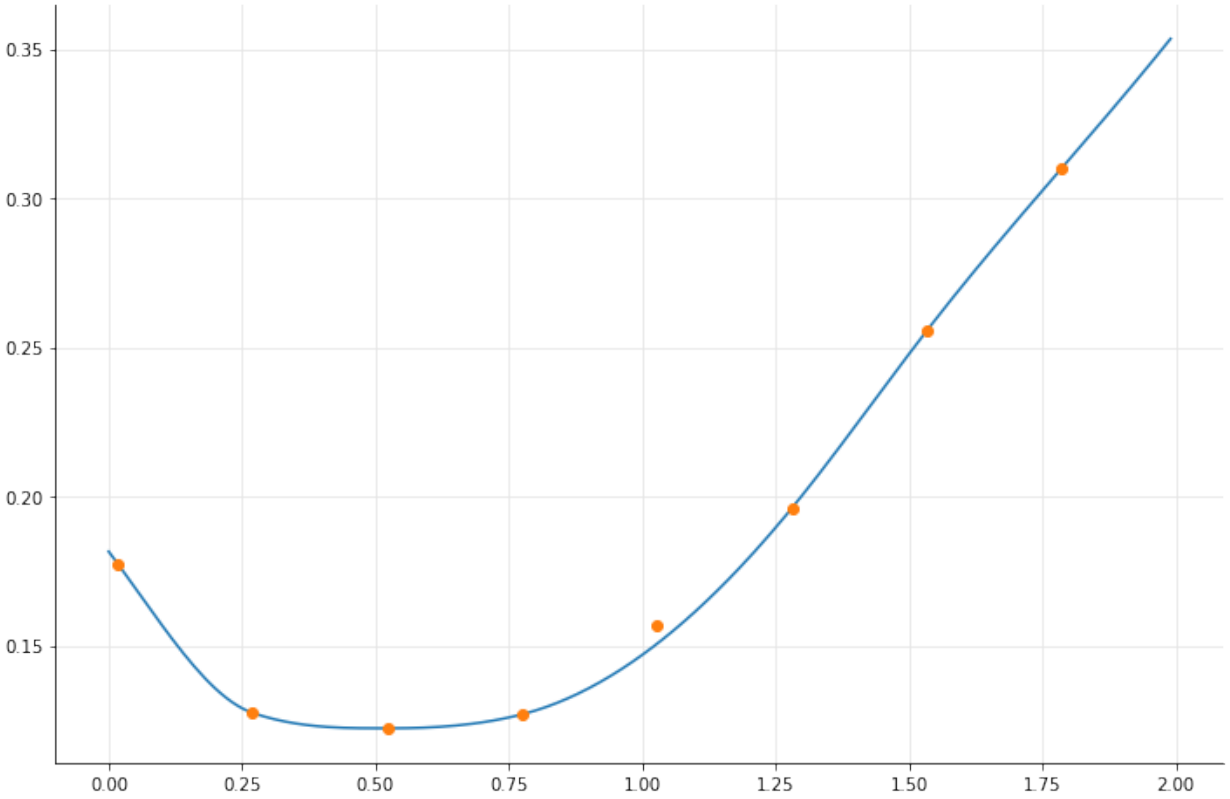
Turn of year

This is not the end of the story, though, since one of the futures we used turns out to be out of line with respect to the others in the strip.

```
In [45]: futures = [(0.1775, Date(19, December, 2012)),
                    (0.1274, Date(20, March, 2013)),
                    (0.1222, Date(19, June, 2013)),
                    (0.1269, Date(18, September, 2013)),
                    (0.1565, Date(18, December, 2013)),
                    (0.1961, Date(19, March, 2014)),
                    (0.2556, Date(18, June, 2014)),
                    (0.3101, Date(17, September, 2014))]
```

Not surprisingly, it's the one that spans the end of the year and thus includes the corresponding jump; that is, the one at index 4 in the list, starting on December 18th. This can be seen clearly enough by fitting a spline between the other futures and plotting the quoted value against the curve:

```
In [46]: spot = euribor6m_curve.referenceDate()
         day_counter = euribor3m.dayCounter()
         quotes, times = zip(*[(q, day_counter.yearFraction(spot, d))
                               for q,d in futures])
         f = MonotonicCubicNaturalSpline(times[:4]+times[5:],
                                         quotes[:4]+quotes[5:])

In [47]: _, ax = utils.plot()
         ts, fs = zip(*[(t,f(t, True)) for t in np.arange(0.0, 2.0, 0.01)])
         ax.plot(ts,fs)
         _ = ax.plot(times,quotes,'o')
```



We can also ask the interpolation for the estimated value and compare it with the real one:

```
In [48]: print(utils.format_rate(quotes[4]))
         print(utils.format_rate(f(times[4])))

Out[48]: 15.65 %
         15.06 %
```

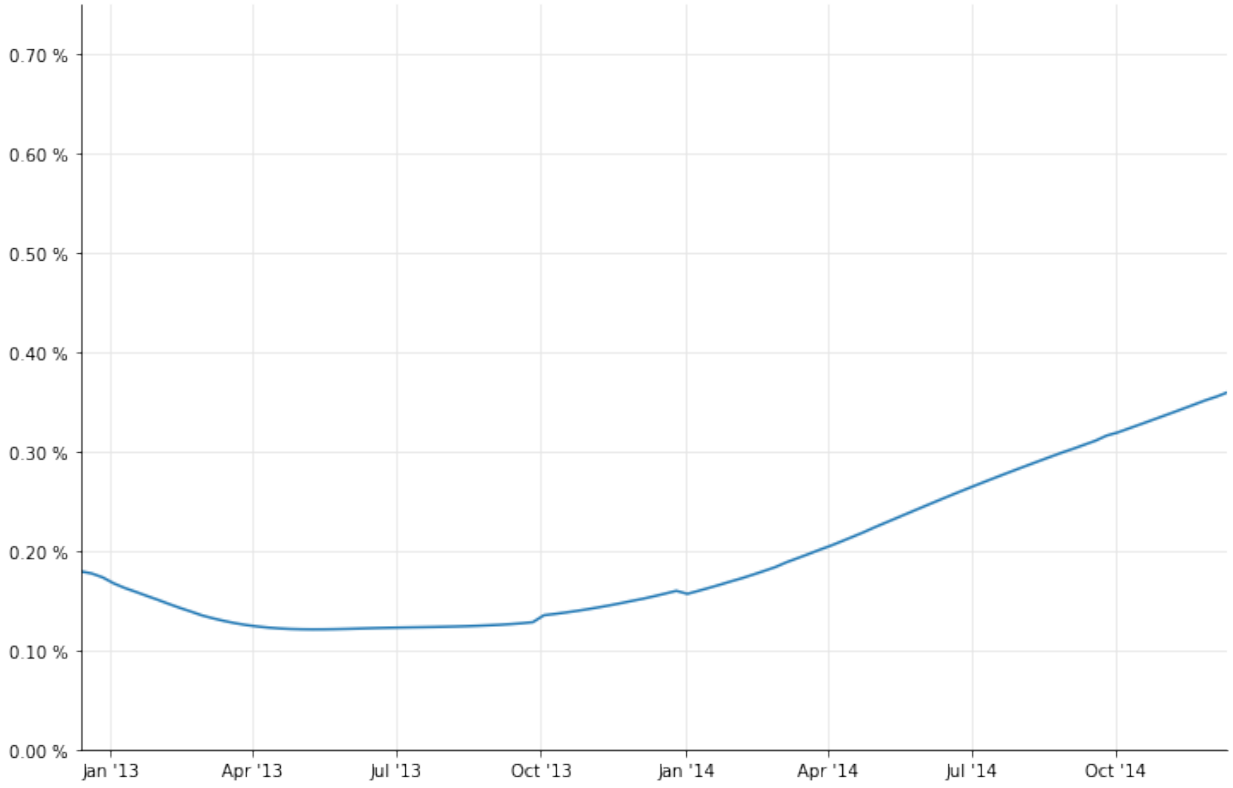
To account for the jump, we can estimate the corresponding discount factor $e^{-J*\tau}$ (where both J and τ are calculated with respect to the tenor of the futures) and add it to the curve.

```
In [49]: J = (quotes[4] - f(times[4]))/100
         tau = day_counter.yearFraction(Date(18,December,2013), Date(18,March,2014))
         print(utils.format_rate(J))
         print(tau)

Out[49]: 0.01 %
         0.25

In [50]: jumps = [QuoteHandle(SimpleQuote(math.exp(-J*tau)))]
         jump_dates = [Date(31,December,2013)]
         euribor3m_curve = PiecewiseLogCubicDiscount(2, TARGET(),
```

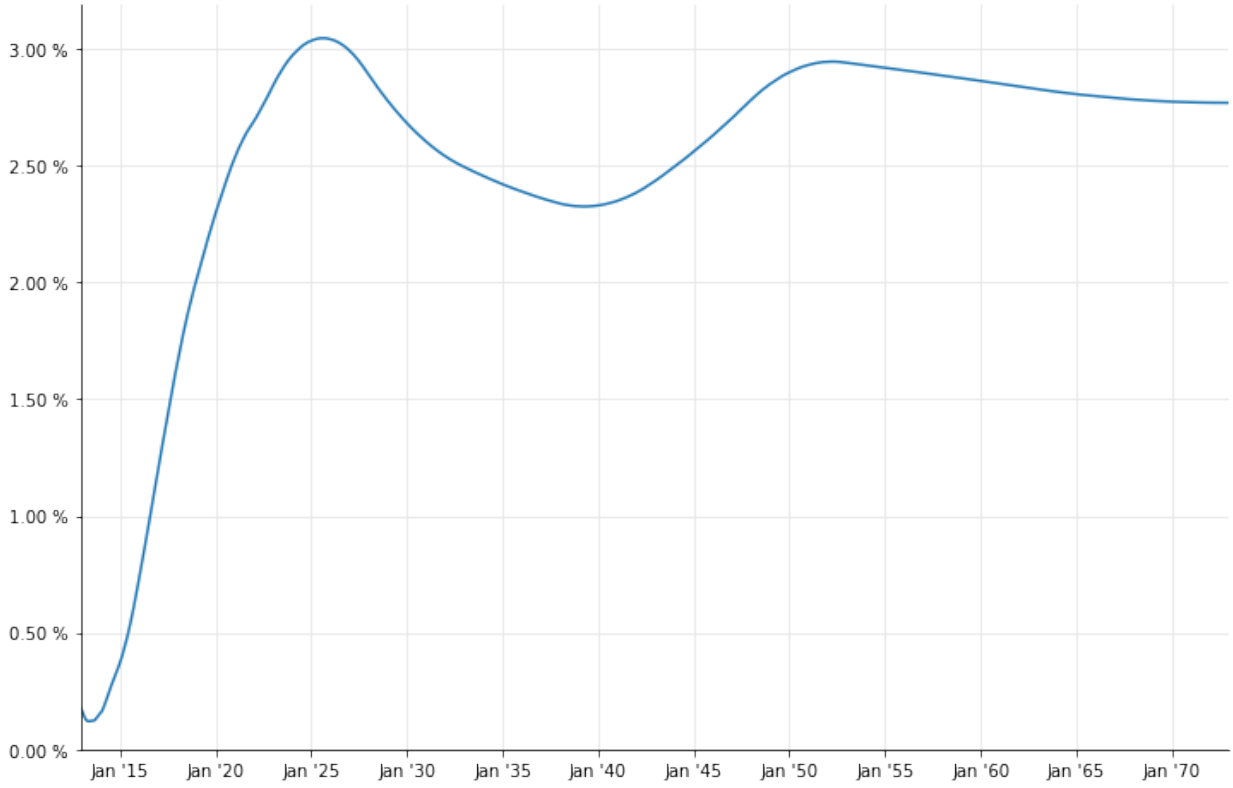
```
                                                helpers+synth_helpers,
                                                Actual365Fixed(),
                                                jumps, jump_dates)
        euribor3m_curve.enableExtrapolation()
```

We can now reproduce figure 30 in the paper. The end-of-year jump can be seen clearly in the first plot.

```
In [51]: spot = euribor3m_curve.referenceDate()
         dates = [ spot+Period(i,Weeks) for i in range(0, 2*52+1) ]
         rates = [ euribor3m_curve.forwardRate(d, euribor3m.maturityDate(d),
                                               Actual360(), Simple).rate()
                   for d in dates ]
         _, ax = utils.plot()
         utils.plot_curve(ax, dates, [(rates, '-')], ymin=0.0, ymax=0.0075, format_rates=True)
```



```
In [52]: dates = [ spot+Period(i,Months) for i in range(0, 60*12+1) ]
         rates = [ euribor3m_curve.forwardRate(d, euribor3m.maturityDate(d),
                                               Actual360(), Simple).rate()
                   for d in dates ]
         _, ax = utils.plot()
         utils.plot_curve(ax, dates, [(rates, '-')], ymin=0.0, format_rates=True)
```



1-month Euribor

Last, let's bootstrap the 1-month Euribor curve. Quoted instruments based on this tenor include the 1-month deposit and interest-rate swaps paying a monthly fixed rate against 1-month Euribor with maturities up to 1 year; their rates are listed in figures 4 and 11.

```
In [53]: euribor1m = Euribor1M()
         helpers = [ DepositRateHelper(QuoteHandle(SimpleQuote(0.110/100)),
                                       Period(1,Months), 2,
                                       TARGET(), Following, False, Actual360()) ]
         helpers += [
             SwapRateHelper(QuoteHandle(SimpleQuote(rate/100)),
                            Period(tenor, Months), TARGET(),
                            Monthly, Unadjusted, Thirty360(Thirty360.BondBasis),
                            euribor1m, QuoteHandle(), Period(0, Days),
                            discount_curve)
             for rate, tenor in [(0.106, 2), (0.096, 3), (0.085, 4), (0.079, 5),
                                 (0.075, 6), (0.071, 7), (0.069, 8), (0.066, 9),
                                 (0.065, 10), (0.064, 11), (0.063, 12)] ]
```

For longer maturities, we can combine the swaps against 6-months Euribor with the 1-month vs 6-months basis swaps shown in figure 15.

```
In [54]: helpers += [
             SwapRateHelper(QuoteHandle(SimpleQuote((rate-basis)/100)),
                            Period(tenor, Years), TARGET(),
                            Annual, Unadjusted, Thirty360(Thirty360.BondBasis),
                            euribor1m, QuoteHandle(), Period(0, Days),
                            discount_curve)
             for rate, basis, tenor in [(0.324, 0.226, 2), (0.424, 0.238, 3),
                                        (0.576, 0.246, 4), (0.762, 0.250, 5),
                                        (0.954, 0.250, 6), (1.135, 0.248, 7),
                                        (1.303, 0.245, 8), (1.452, 0.241, 9),
                                        (1.584, 0.237, 10), (1.703, 0.233, 11),
                                        (1.809, 0.228, 12), (2.037, 0.211, 15),
                                        (2.187, 0.189, 20), (2.234, 0.175, 25),
                                        (2.256, 0.163, 30)] ]
```

As before, we can use synthetic deposits for maturities below the 1-month tenor…

```
In [55]: synth_helpers = [
             DepositRateHelper(QuoteHandle(SimpleQuote(rate/100)),
                               Period(*tenor), 2,
                               TARGET(), Following, False, Actual360())
             for rate, tenor in [(0.0661, (1,Days)), (0.098, (1,Weeks)),
                                 (0.0993, (2,Weeks)), (0.1105, (3,Weeks))] ]
```

…and we'll extend the 30-years basis spread flatly to combine it with longer-maturity swaps against 6-months Euribor.

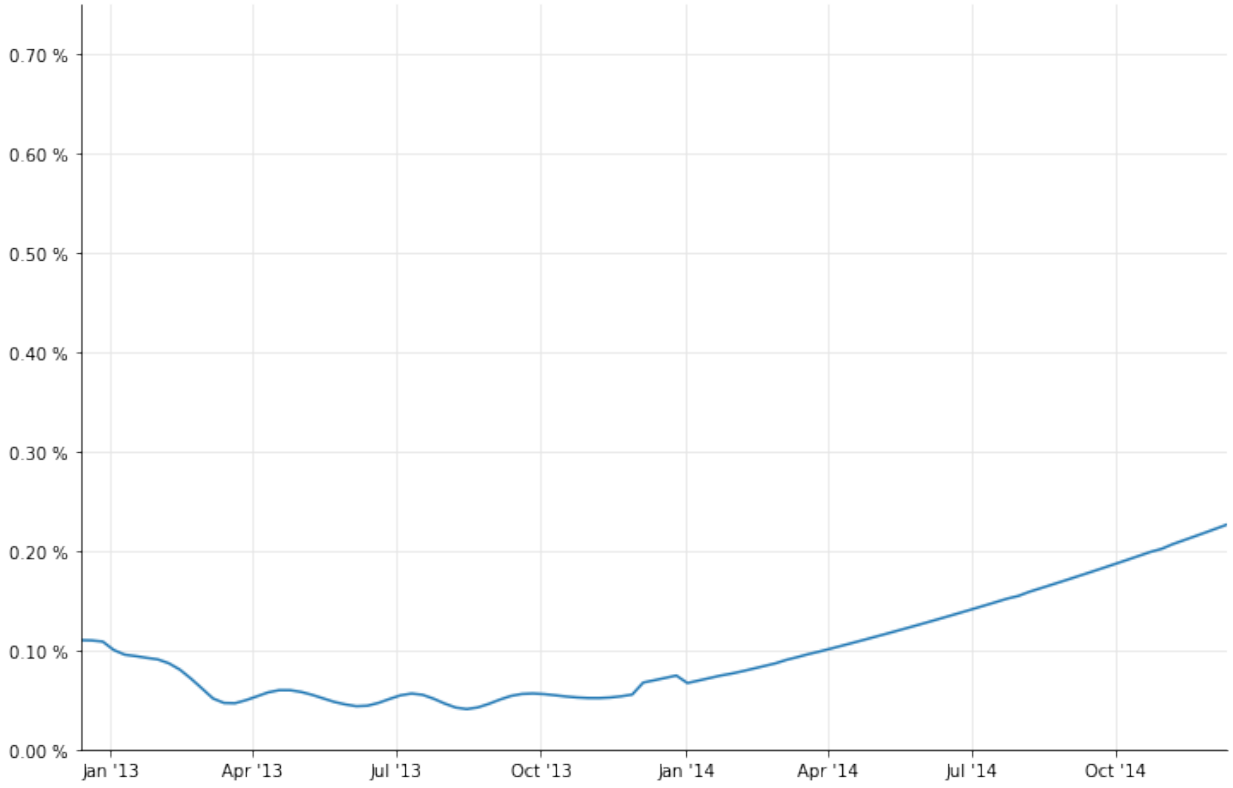
```
In [56]: last_basis = 0.163
         synth_helpers += [
             SwapRateHelper(QuoteHandle(SimpleQuote((rate-last_basis)/100)),
                            Period(tenor, Years), TARGET(),
                            Annual, Unadjusted, Thirty360(Thirty360.BondBasis),
                            euribor1m, QuoteHandle(), Period(0, Days),
                            discount_curve)
             for rate, tenor in [(2.295, 35), (2.348, 40),
                                 (2.421, 50), (2.463, 60)] ]
```

This curve, too, shows a jump at the end of the year. The paper claims that it can be determined and corrected by interpolating the quoted swaps with maturities from 1 to 12 months, but I haven't reproduced the calculation yet. For the time being, I'll just use the value reported in the paper and calculate the corresponding discount factor.

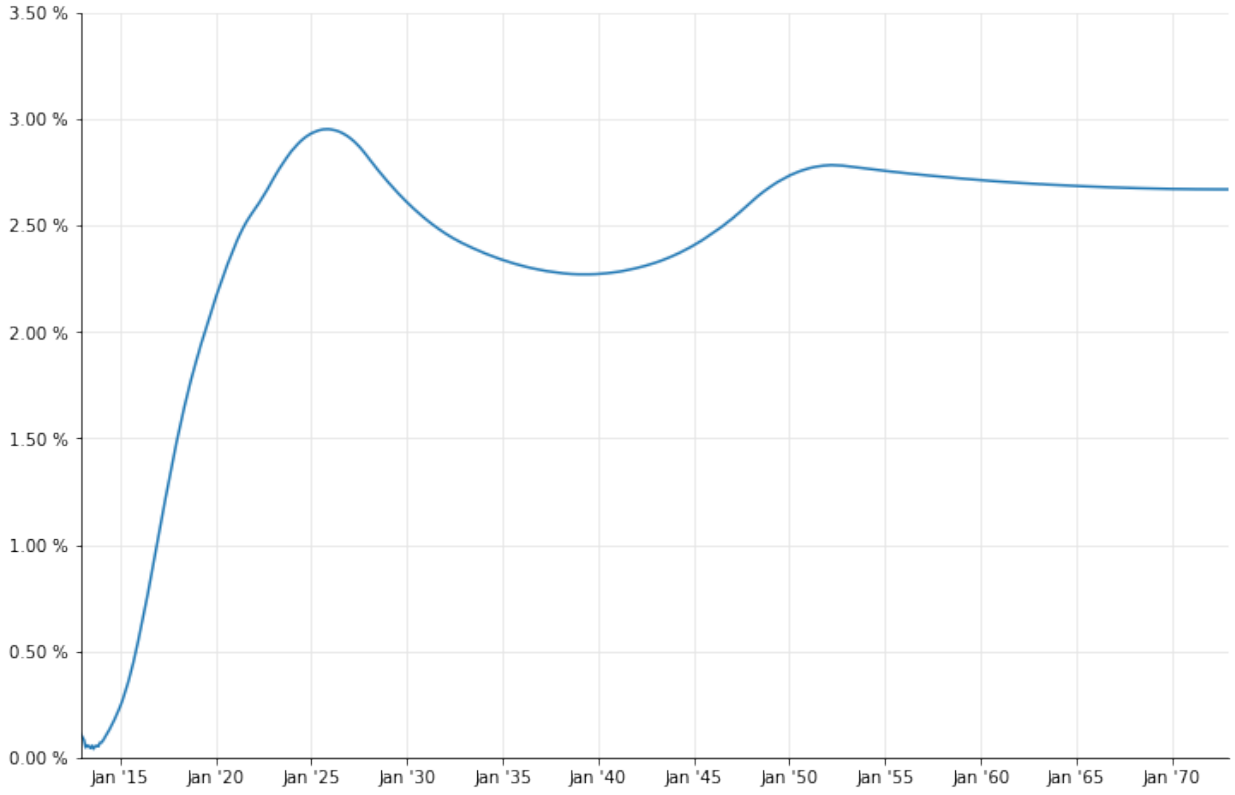
```
In [57]: J = 0.0016
         t_j = euribor1m.dayCounter().yearFraction(Date(31,December,2012),
                                                   Date(2,January,2013))
         B = 1.0/(1.0+J*t_j)
         jumps = [QuoteHandle(SimpleQuote(B))]
         jump_dates = [Date(31,December,2013)]

In [58]: euribor1m_curve = PiecewiseLogCubicDiscount(2, TARGET(),
                                                     helpers+synth_helpers,
                                                     Actual365Fixed(),
                                                     jumps, jump_dates)
         euribor1m_curve.enableExtrapolation()
```

This last curve gives us figure 28 in the paper, down to the oscillations during the first year.

```
In [59]: spot = euribor1m_curve.referenceDate()
         dates = [ spot+Period(i,Weeks) for i in range(0, 2*52+1) ]
         rates = [ euribor1m_curve.forwardRate(d, euribor1m.maturityDate(d),
                                               Actual360(), Simple).rate()
                   for d in dates ]
         _, ax = utils.plot()
         utils.plot_curve(ax, dates, [(rates, '-')], ymin=0.0, ymax=0.0075, format_rates=True)
```




```
In [60]: dates = [ spot+Period(i,Months) for i in range(0, 60*12+1) ]
         rates = [ euribor1m_curve.forwardRate(d, euribor1m.maturityDate(d),
                                               Actual360(), Simple).rate()
                   for d in dates ]
         _, ax = utils.plot()
         utils.plot_curve(ax, dates, [(rates, '-')], ymin=0.0, ymax=0.035, format_rates=True)
```



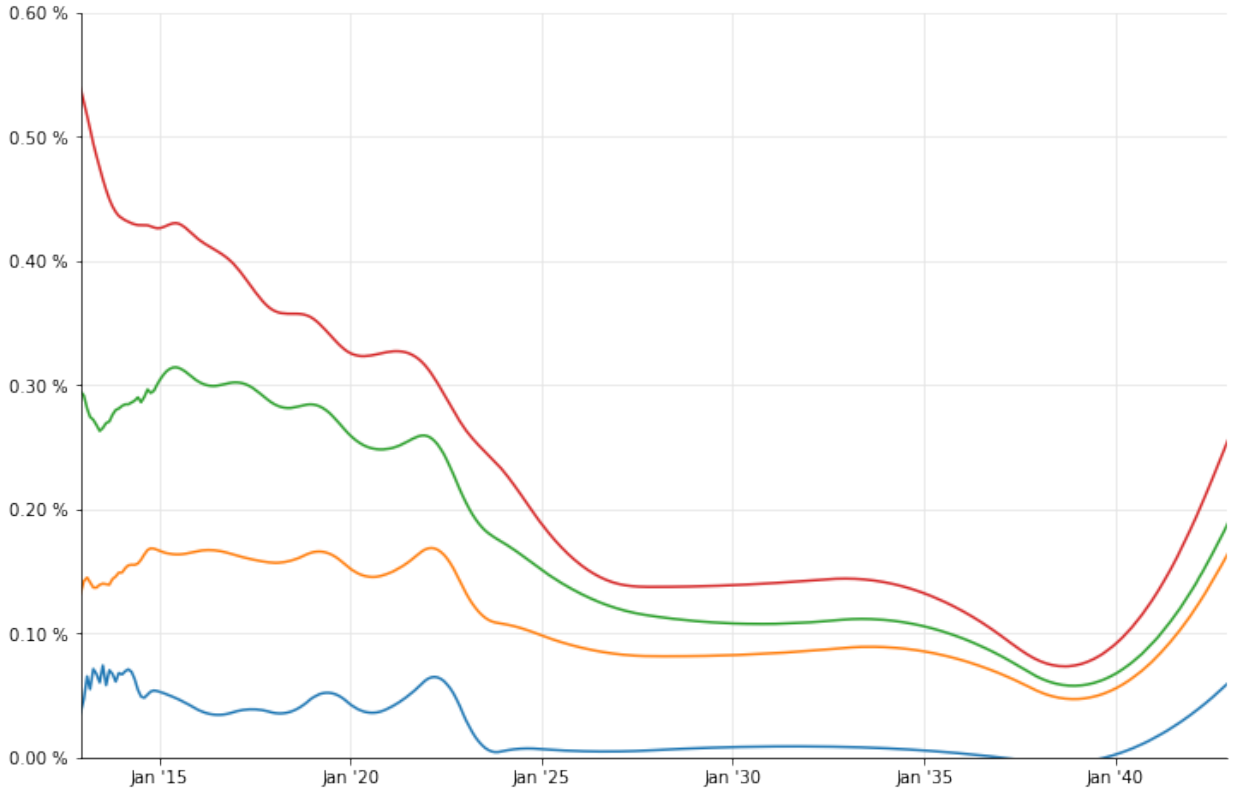
Basis curves

Finally, like the authors of the paper, we summarize the results by calculating the difference between the FRA rates calculated on the corresponding Euribor curve and those calculated on the ON curve. This lets us reproduce the top panel of figure 35.

```
In [61]: dates = [ spot+Period(i,Months) for i in range(0, 12*30+1) ]
```

```
def basis(curve, tenor):
    results = []
    for d in dates:
        d2 = TARGET().advance(d, Period(*tenor), ModifiedFollowing)
        FRA1 = curve.forwardRate(d, d2, Actual360(), Simple).rate()
        FRA2 = eonia_curve.forwardRate(d, d2, Actual360(), Simple).rate()
        results.append(FRA1-FRA2)
    return results

basis_1m = basis(euribor1m_curve, (1,Months))
basis_3m = basis(euribor3m_curve, (3,Months))
basis_6m = basis(euribor6m_curve, (6,Months))
basis_12m = basis(euribor12m_curve, (12,Months))
```

```
_, ax = utils.plot()
utils.plot_curve(ax, dates, [(basis_1m, '-'), (basis_3m, '-'),
                             (basis_6m, '-'), (basis_12m, '-')],
                 ymin=0, ymax=0.006, format_rates=True)
```



10. 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]: from QuantLib import *
        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 = [Period(6,Months), Period(12, Months)]
        depo_rates = [5.25, 5.5]

        # Bond rates
        bond_maturities = [Period(6*i, 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
1Y	5.50
1Y6M	5.75
2Y	6.00
2Y6M	6.25
3Y	6.50
3Y6M	6.75
4Y	6.80
4Y6M	7.00
5Y	7.10

Maturities	Curve
5Y6M	7.15
6Y	7.20
6Y6M	7.30
7Y	7.35
7Y6M	7.40
8Y	7.50
8Y6M	7.60
9Y	7.60
9Y6M	7.70
10Y	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 = Date(15, 1, 2015)
        Settings.instance().evaluationDate = calc_date

        calendar = UnitedStates()
        business_convention = Unadjusted
        day_count = Thirty360()
        end_of_month = True
        settlement_days = 0
        face_amount = 100
        coupon_frequency = Period(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 = [DepositRateHelper(QuoteHandle(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 = Schedule(calc_date,
                                termination_date,
                                coupon_frequency,
                                calendar,
                                business_convention,
                                business_convention,
                                DateGeneration.Backward,
                                end_of_month)

            bond_helper = FixedRateBondHelper(QuoteHandle(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=UnitedStates(), 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, Period(month, Months))
                compounding = Compounded
                freq = 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 = 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.

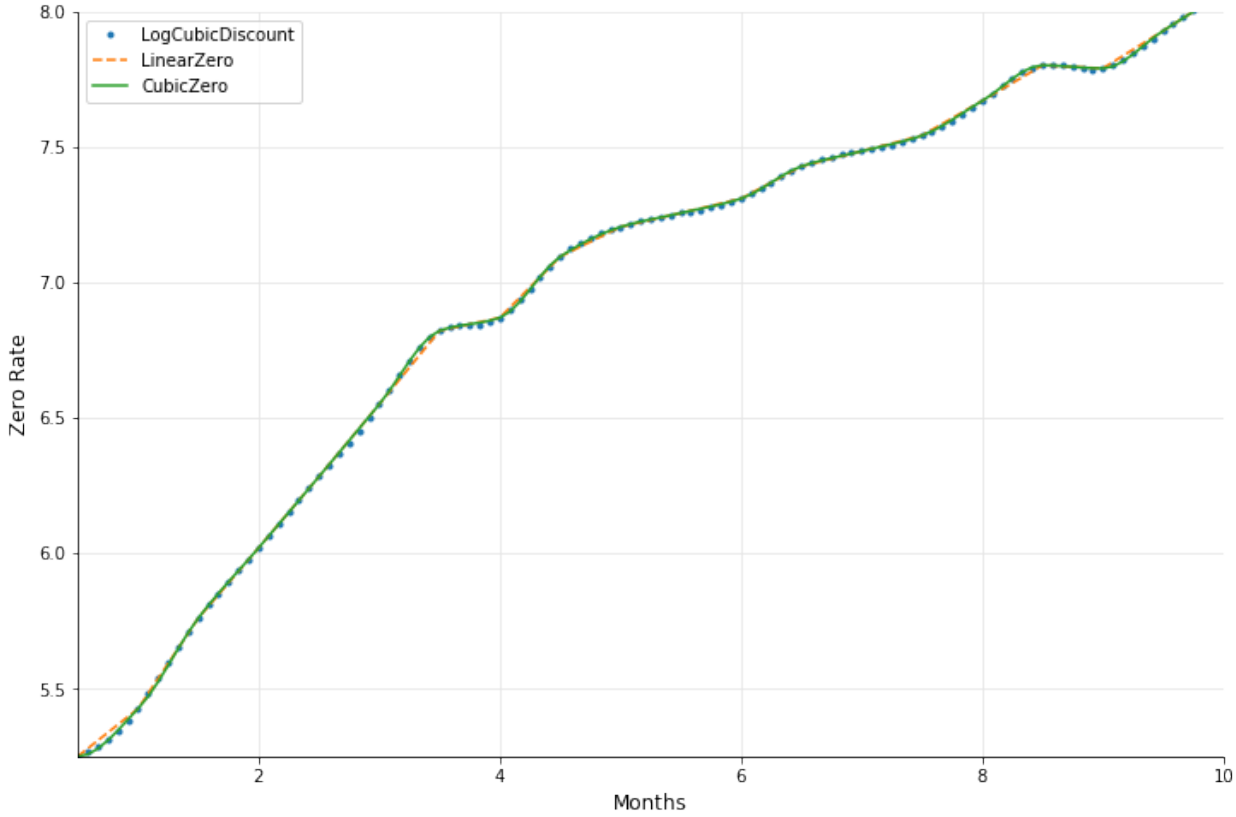
```
In [10]: yc_linearzero = PiecewiseLinearZero(
                 calc_date,rate_helpers,day_count)
         yc_cubiczero = PiecewiseCubicZero(
                 calc_date,rate_helpers,day_count)

         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.

11. Dangerous day-count conventions

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

```
In [1]: from QuantLib import *

In [2]: today = Date(22,1,2018)
        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 = Thirty360(Thirty360.USA)

In [5]: d1 = d1 = Date(1,January,2018)
        d2 = Date(15, January, 2018)
        d3 = Date(31, 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 = Date(1,January,2018)
        d2 = Date(30, January, 2018)
        d3 = Date(31, 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:

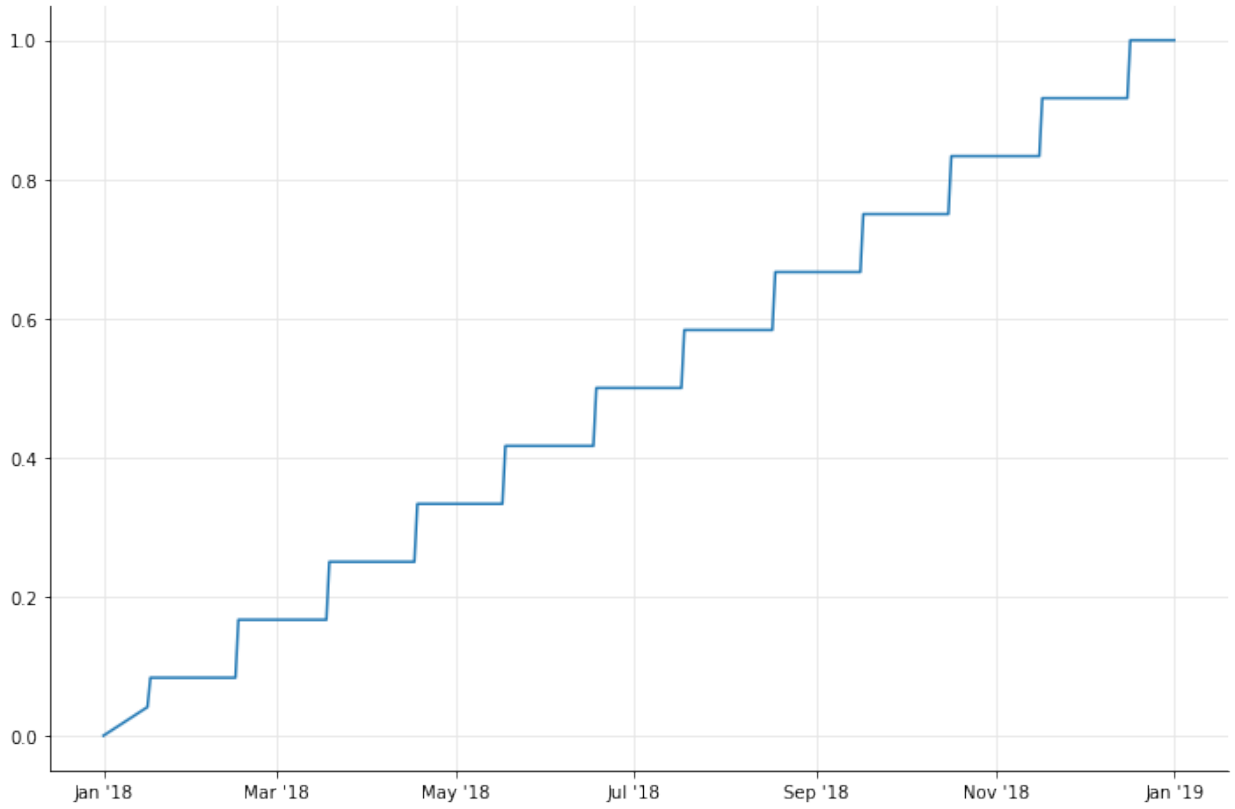
```
In [9]: d1 = Date(30, January, 2018)
        d2 = Date(31, January, 2018)

        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 = Date(1, January, 2018)
         d2 = Date(15, January, 2018)

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

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

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

Out[11]: 0.03867403314917127
         0.038356164383561646
```

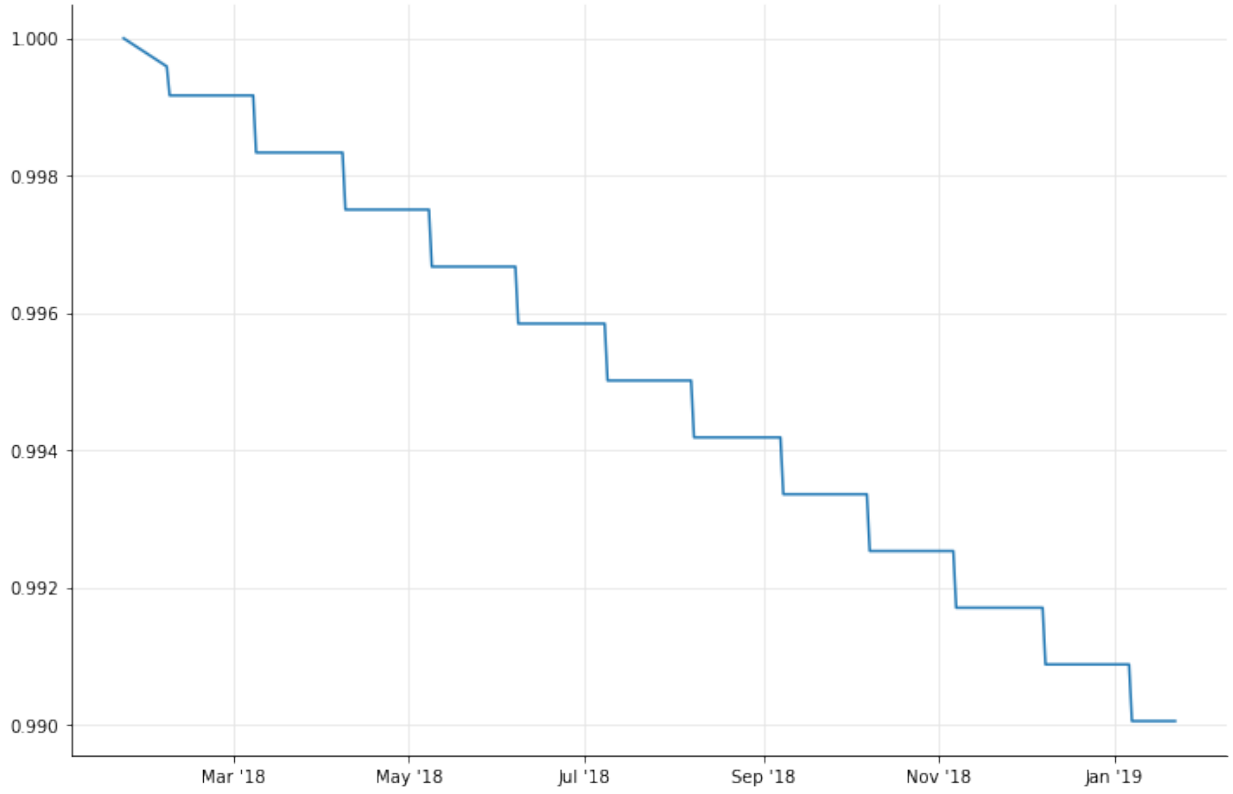
…and sometimes, in spectacularly wrong results. Here is what happens if we plot the year fraction since January 1st, 2018 as a function of the date over that same year.

```
In [12]: d1 = Date(1, 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 = FlatForward(today, 0.01, ActualActual(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.

12. Implied term structures

(Based on a question[1] by *Stack Exchange* user Lisa Ann. Thanks!)

```
In [1]: from QuantLib import *
```

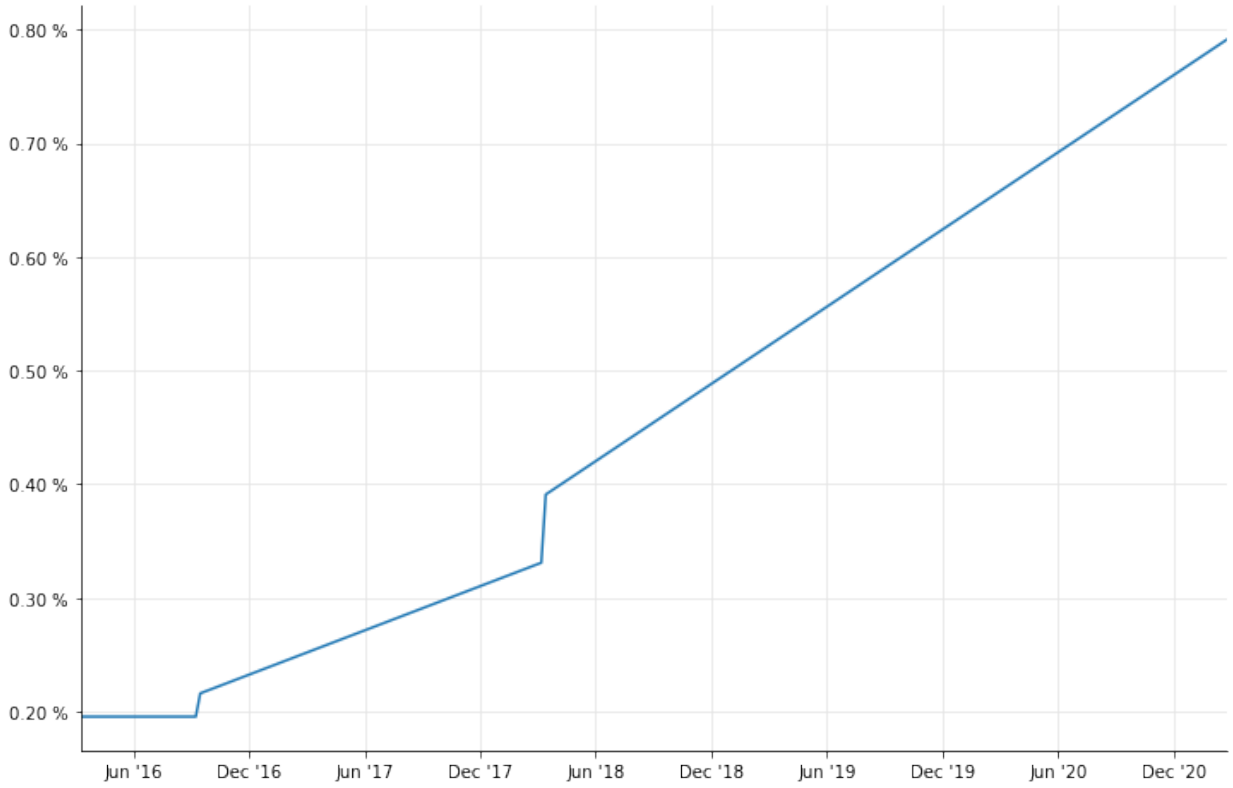
```
In [2]: %matplotlib inline
        import pandas as pd
        import utils
        from utils import to_datetime, format_rate

        from matplotlib.dates import MonthLocator, DateFormatter
        from matplotlib.ticker import FuncFormatter
        def plot_curve(*curves):
            fig, ax = utils.plot()
            dates = [ today+Period(i,Weeks) for i in range(0, 52*5) ]
            for (c, style) in curves:
                valid_dates = [ d for d in dates if d >= c.referenceDate() ]
                rates = [ c.forwardRate(d, d+1, Actual360(), Simple).rate()
                          for d in valid_dates ]
                ax.plot_date([ to_datetime(d) for d in valid_dates ], rates, style)
            ax.set_xlim(to_datetime(min(dates)),to_datetime(max(dates)))
            ax.xaxis.set_major_locator(MonthLocator(bymonth=[6,12]))
            ax.xaxis.set_major_formatter(DateFormatter("%b '%y"))
            ax.xaxis.grid(True, 'major')
            ax.xaxis.grid(False, 'minor')
            ax.yaxis.set_major_formatter(FuncFormatter(lambda r,pos: format_rate(r)))
```

The statement of the case

Let's say we have an interest-rate curve. For the sake of example, I'll take a simple one bootstrapped on a few swap rates.

[1] http://quant.stackexchange.com/questions/9589/

```
In [3]: today = Date(9,March,2016)
        Settings.instance().evaluationDate = today

In [4]: helpers = [ SwapRateHelper(QuoteHandle(SimpleQuote(rate/100.0)),
                                   Period(*tenor), TARGET(),
                                   Annual, Unadjusted,
                                   Thirty360(),
                                   Euribor6M())
                    for tenor, rate in [((6,Months), 0.201),
                                        ((2,Years), 0.258),
                                        ((5,Years), 0.464),
                                        ((10,Years), 1.151),
                                        ((15,Years), 1.588)] ]
        curve = PiecewiseLinearZero(0, TARGET(), helpers, Actual360())

In [5]: plot_curve((curve, '-'))
```



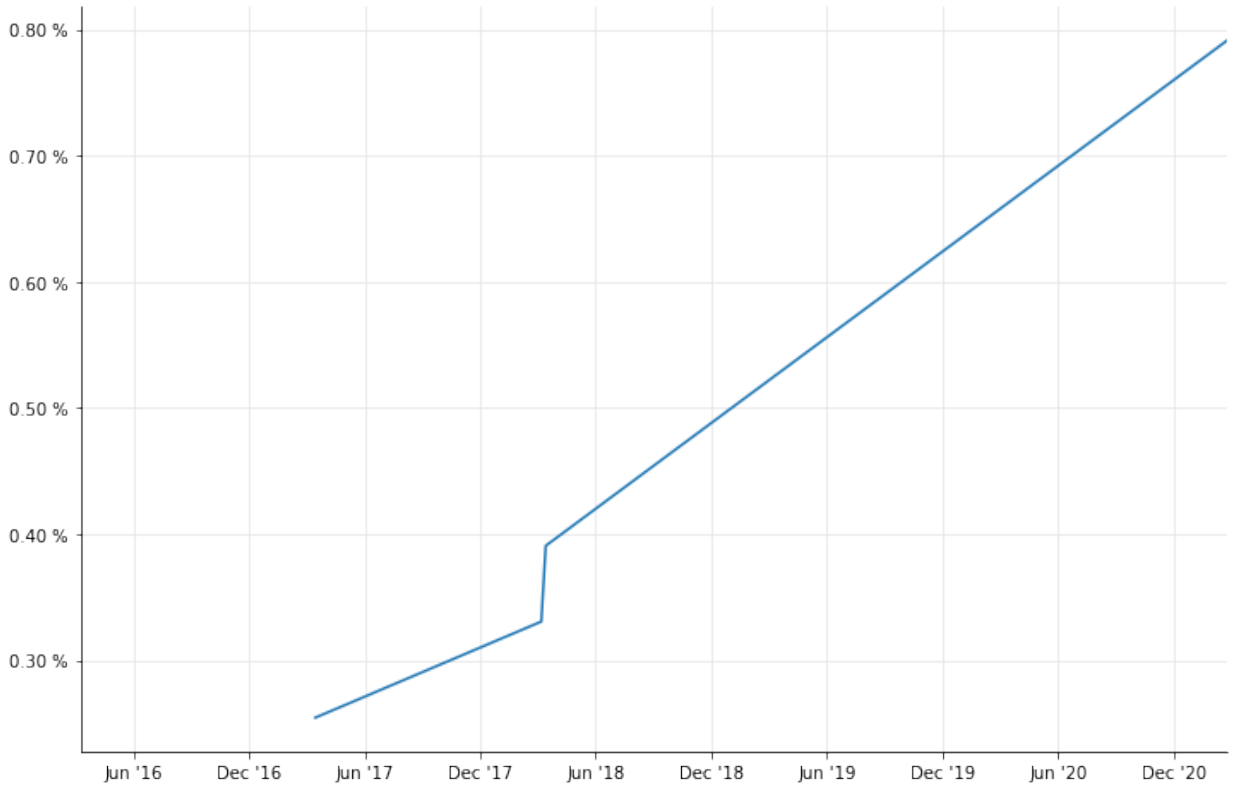
I'm using linear interpolation on the zero rates, which isn't great for actual use. However, the resulting jumps in the forward rates will be useful as visual points of reference; note, for instance, the jump around March 2018.

The curve also implies an interest-rate curve in 1 year; meaning that, for instance, it can give us the forward rate between 1 and to 2 years, that we expect to be the 1-year spot rate one year from now,

or the forward rate between 1 year and 18 months, which will be the 6-months spot rate in one year. The implied curve can be built as an instance of the `ImpliedTermStructure` class:

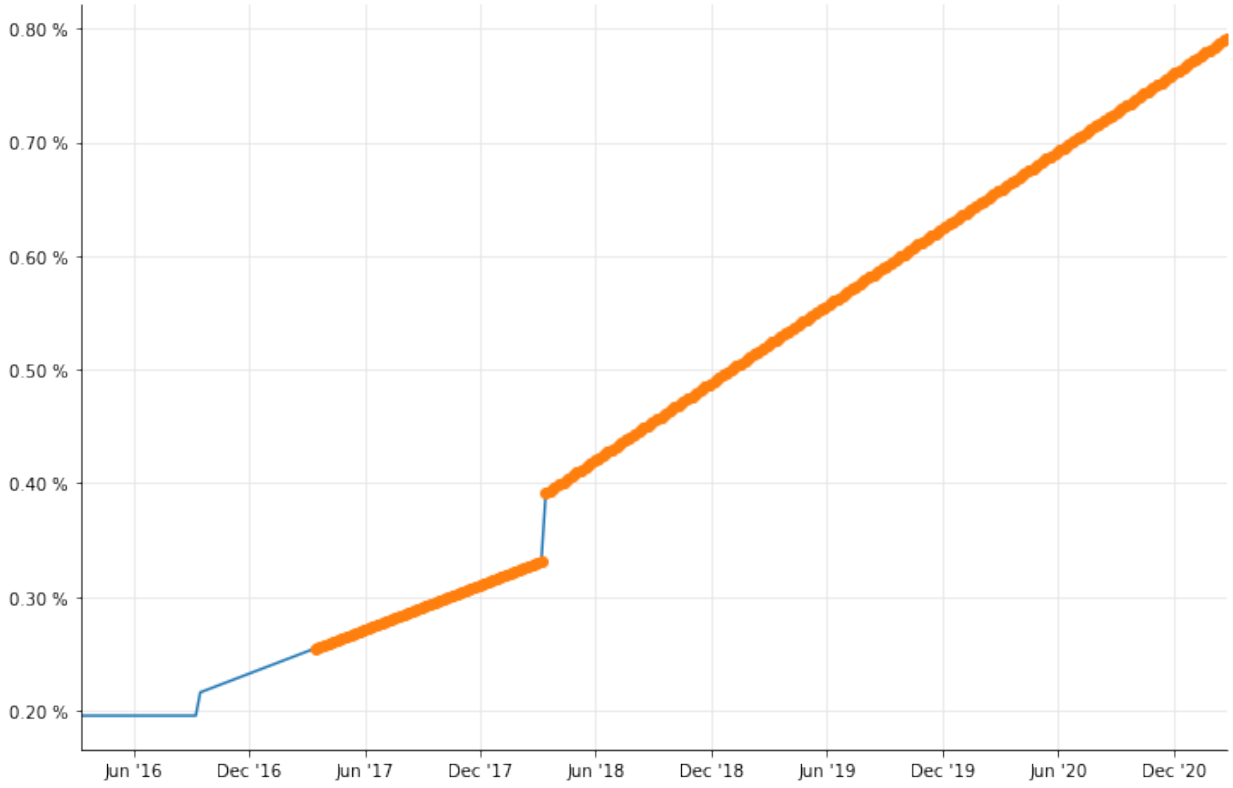
```
In [6]: future_reference = today + Period(1,Years)
        implied_curve = ImpliedTermStructure(YieldTermStructureHandle(curve),
                                             future_reference)
```

```
In [7]: plot_curve((implied_curve, '-'))
```



In the common range, the two curves are the same...

```
In [8]: plot_curve((curve, '-'), (implied_curve, 'o'))
```



...although, of course, a spot rate for the implied curve corresponds to a forward rate for the original curve.

```
In [9]: dates = [ future_reference + Period(i, Years) for i in range(6) ]
        rates_1 = [ curve.forwardRate(future_reference, d,
                                      Actual360(), Continuous).rate()
                    for d in dates ]
        rates_2 = [ implied_curve.zeroRate(d, Actual360(), Continuous).rate()
                    for d in dates ]
        pd.DataFrame(list(zip(dates,
                              [ format_rate(r) for r in rates_1 ],
                              [ format_rate(r) for r in rates_2 ])),
                     columns=('Maturity', 'Original forward rate',
                              'Implied zero rate'), index=['']*6)
```

Out[9]:

Maturity	Original forward rate	Implied zero rate
March 9th, 2017	0.25 %	0.25 %
March 9th, 2018	0.29 %	0.29 %
March 9th, 2019	0.37 %	0.37 %
March 9th, 2020	0.45 %	0.45 %
March 9th, 2021	0.52 %	0.52 %
March 9th, 2022	0.67 %	0.67 %

Now, Lisa Ann's idea was to forecast a bond price as of one year from now based on the implied curve. In the library framework, that means setting the evaluation date to 1 year from today and using the implied curve to discount the bond cash flows. However, after changing the evaluation date...

```
In [10]: Settings.instance().evaluationDate = future_reference
```

...the implied curve had changed.

```
In [11]: rates_3 = [ implied_curve.zeroRate(d, Actual360(), Continuous).rate()
                     for d in dates ]
         pd.DataFrame(list(zip(dates,
                               [ format_rate(r) for r in rates_2 ],
                               [ format_rate(r) for r in rates_3 ])),
                      columns=('Maturity', 'Before date change',
                               'After date change'), index=['']*6)
```

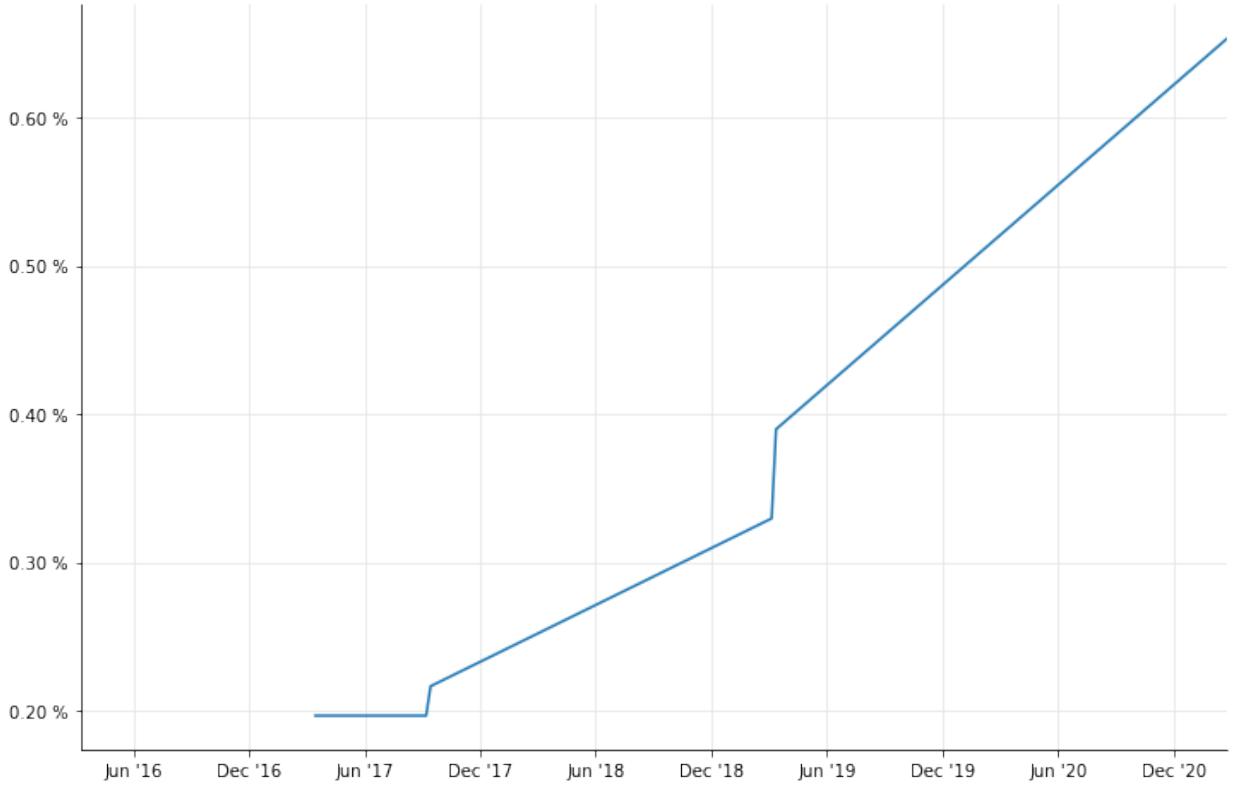
Out[11]:

Maturity	Before date change	After date change
March 9th, 2017	0.25 %	0.20 %
March 9th, 2018	0.29 %	0.22 %
March 9th, 2019	0.37 %	0.25 %
March 9th, 2020	0.45 %	0.32 %
March 9th, 2021	0.52 %	0.39 %
March 9th, 2022	0.67 %	0.46 %

What happened?

Simply put: the reference date of the original curve was specified relative to the evaluation date, and when we moved it the curve moved, too. Let's try it:

```
In [12]: Settings.instance().evaluationDate = future_reference
         plot_curve((curve, '-'))
```



Remember that jump in March 2018? It's in March 2019 now.

Thus, after moving the evaluation date, the original and implied curve are exactly the same; and the spot rates returned by the implied curve are no longer forward rates, but the spot rates returned by the original curve.

```
In [13]: rates_1 = [ curve.zeroRate(d, Actual360(), Continuous).rate()
                     for d in dates ]
         rates_2 = [ implied_curve.zeroRate(d, Actual360(), Continuous).rate()
                     for d in dates ]
         pd.DataFrame(list(zip(dates,
                               [ format_rate(r) for r in rates_1 ],
                               [ format_rate(r) for r in rates_2 ])),
                      columns=('Maturity', 'Original zero rate',
                               'Implied zero rate'), index=['']*6)
```

Out[13]:

Maturity	Original zero rate	Implied zero rate
March 9th, 2017	0.20 %	0.20 %
March 9th, 2018	0.22 %	0.22 %
March 9th, 2019	0.25 %	0.25 %
March 9th, 2020	0.32 %	0.32 %
March 9th, 2021	0.39 %	0.39 %
March 9th, 2022	0.46 %	0.46 %

The solution would be to build the original curve so that it doesn't move when the evaluation date changes; and as you might remember, the way to do that is to specify a reference date explicitly.

Unfortunately, though, doing this to a bootstrapped curve is an open issue; even if we specified the reference date, the underlying swaps would still move (it's a long story). Thus, the actual solution will be a bit of a kludge: we'll make a frozen copy of the original curve that doesn't move when the evaluation date does. The way we do it is to return to the original evaluation date…

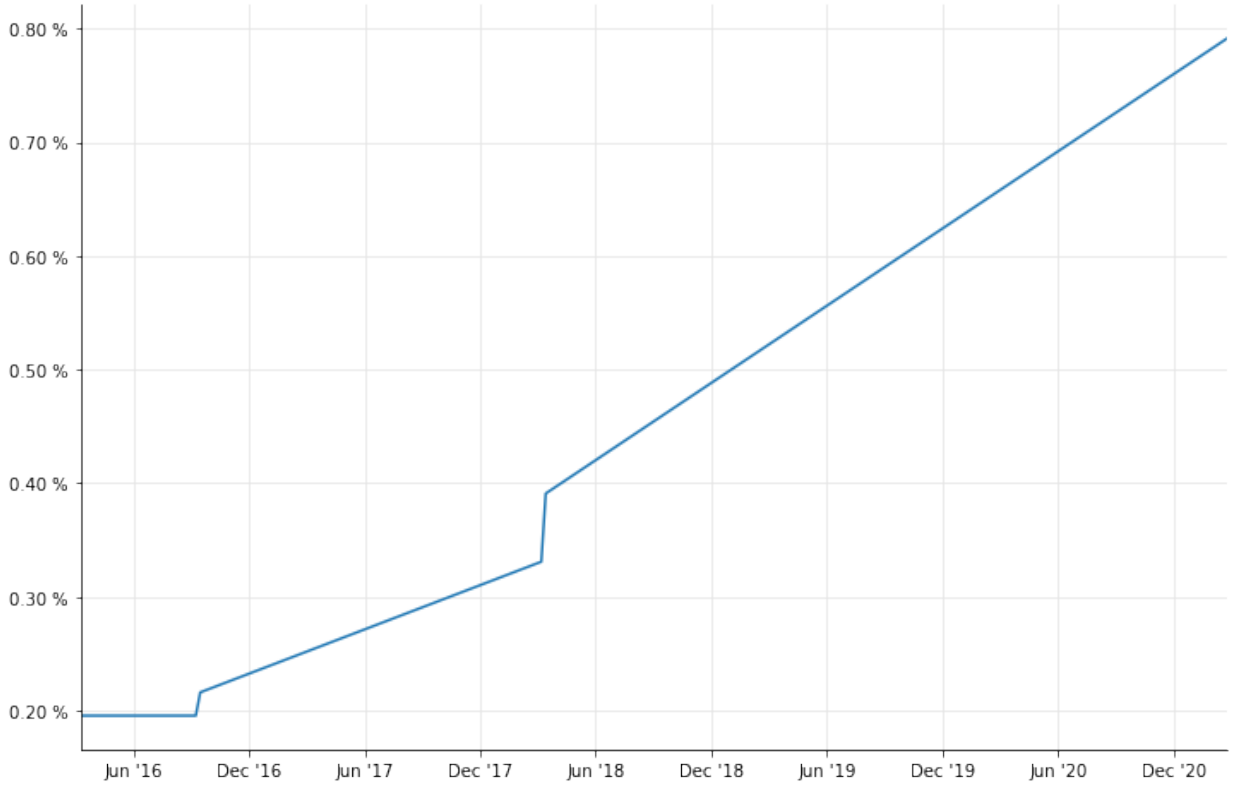
```
In [14]: Settings.instance().evaluationDate = today
```

…extract the bootstrapped rates…

```
In [15]: curve.nodes()

Out[15]: ((Date(9,3,2016), 0.001954693606572509),
          (Date(12,9,2016), 0.001954693606572509),
          (Date(12,3,2018), 0.002536800732553941),
          (Date(11,3,2021), 0.004572804156623578),
          (Date(11,3,2026), 0.011524783611804843),
          (Date(11,3,2031), 0.01615156507336212))
```

…and create a curve with the same rates and a fixed reference date.

```
In [16]: node_dates, node_rates = zip(*curve.nodes())
         frozen_curve = ZeroCurve(node_dates, node_rates, curve.dayCounter())

In [17]: plot_curve((frozen_curve, '-'))
```



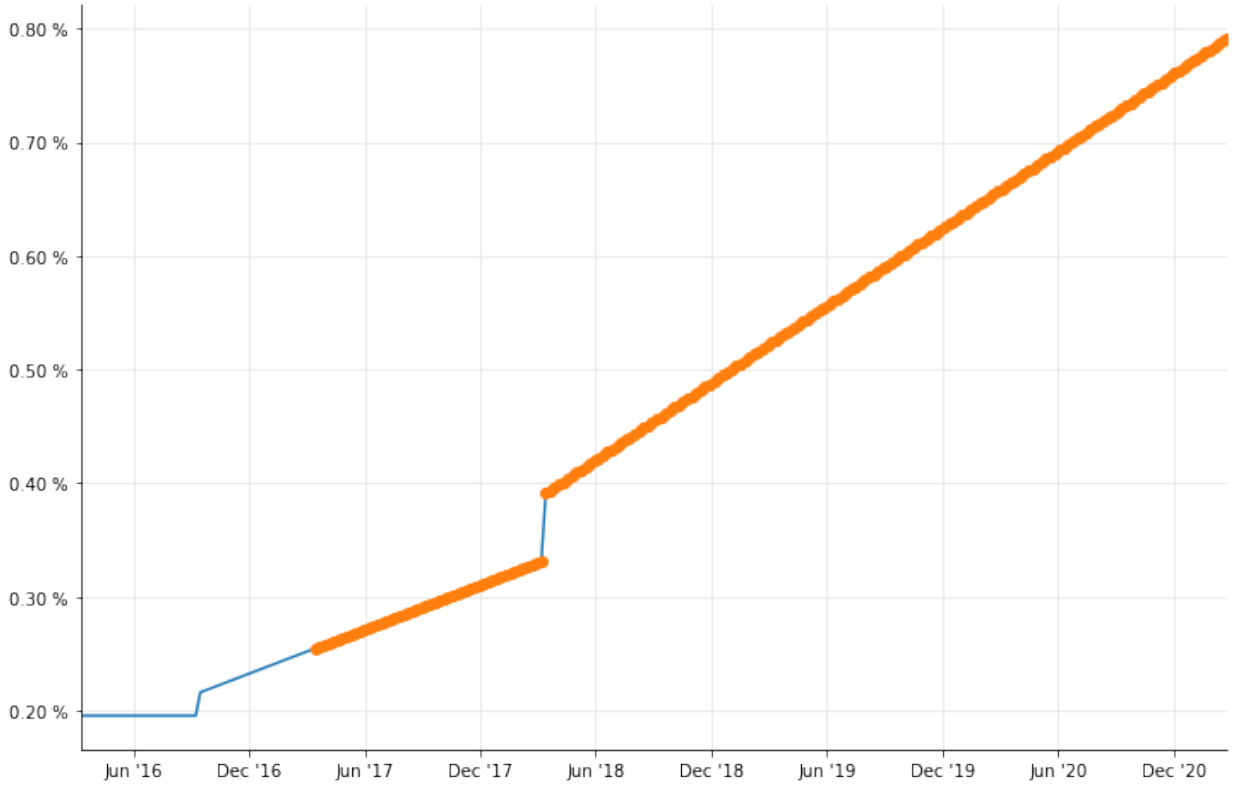
As I said, a bit of a kludge: this curve is a frozen copy, and won't react to changes in the underlying quoted swap rates, so you'd have to recreate it manually if you want to track the market as it moves.

However, now we can build the implied curve based on the frozen one:

```
In [18]: implied_curve = ImpliedTermStructure(YieldTermStructureHandle(frozen_curve),
                                               future_reference)
```

If we move the evaluation date, the frozen curve remains fixed at today's date…

```
In [19]: Settings.instance().evaluationDate = future_reference
         plot_curve((frozen_curve,'-'), (implied_curve,'o'))
```



...and the implied curve returns the correct rates.

```
In [20]: rates_1 = [ frozen_curve.zeroRate(d, Actual360(), Continuous).rate()
                    for d in dates ]
         rates_2 = [ frozen_curve.forwardRate(future_reference, d,
                                              Actual360(), Continuous).rate()
                    for d in dates ]
         rates_3 = [ implied_curve.zeroRate(d, Actual360(), Continuous).rate()
                    for d in dates ]
         pd.DataFrame(list(zip(dates,
                               [ format_rate(r) for r in rates_1 ],
                               [ format_rate(r) for r in rates_2 ],
                               [ format_rate(r) for r in rates_3 ])),
                      columns=('Maturity', 'Original zero rate',
                               'Original forward rate', 'Implied zero rate'),
                      index=['']*6)

Out[20]:
```

Maturity	Original zero rate	Original forward rate	Implied zero rate
March 9th, 2017	0.21 %	0.25 %	0.25 %
March 9th, 2018	0.25 %	0.29 %	0.29 %
March 9th, 2019	0.32 %	0.37 %	0.37 %
March 9th, 2020	0.39 %	0.45 %	0.45 %
March 9th, 2021	0.46 %	0.52 %	0.52 %
March 9th, 2022	0.60 %	0.67 %	0.67 %

13. Interest-rate sensitivities via zero spread

In this notebook, I'll show a couple of different ways to calculate the sensitivity of an instrument price to changes in the interest-rate curve.

```
In [1]: from QuantLib import *
        today = Date(8, March, 2016)
        Settings.instance().evaluationDate = today
```

```
In [2]: %matplotlib inline
        from matplotlib.ticker import FuncFormatter
        import numpy as np
        import utils
        from utils import format_rate

        def plot_curves(*curves):
            fig, ax = utils.plot()
            ax.yaxis.set_major_formatter(FuncFormatter(lambda r,pos: format_rate(r)))
            ax.set_xlim(0,15)
            ax.set_xticks([0,5,10,15])
            times = np.linspace(0.0, 15.0, 400)
            for curve, style in curves:
                rates = [ curve.zeroRate(t, Continuous).rate() for t in times ]
                ax.plot(times, rates, style)
        def plot_curve(curve):
            plot_curves((curve,'-'))
```

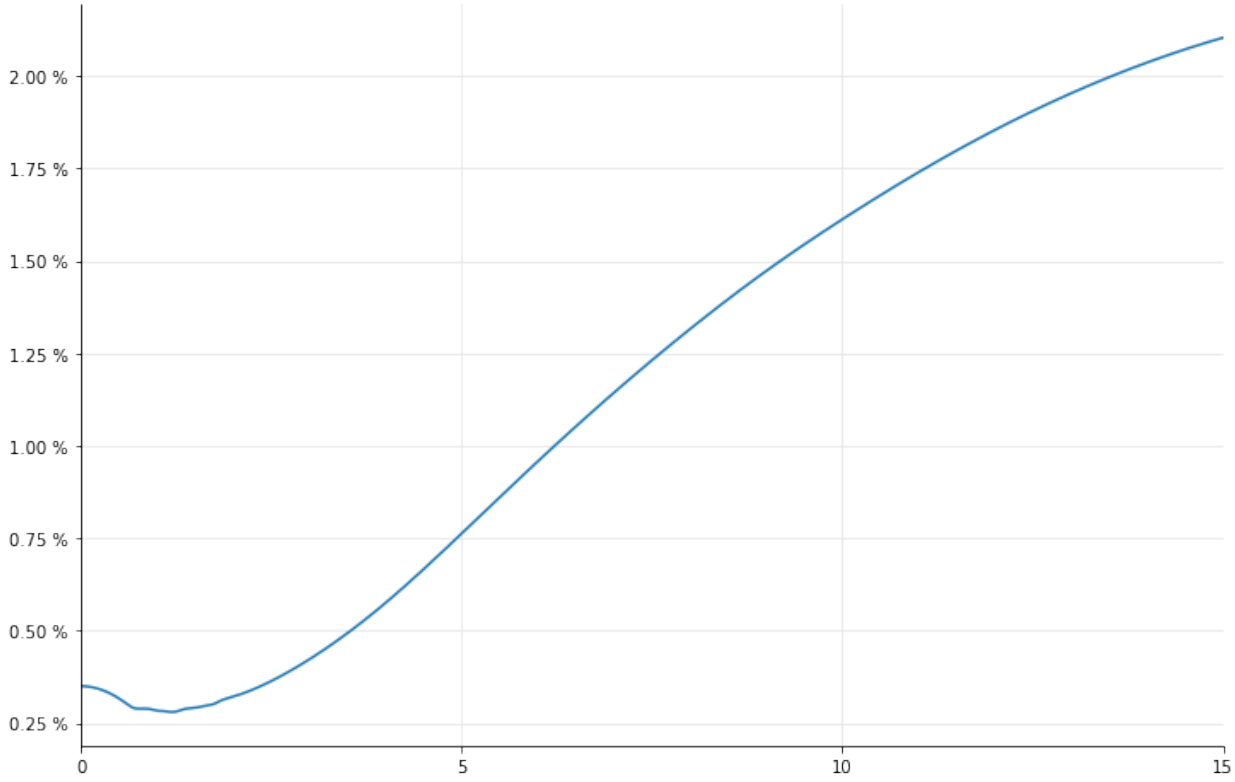
Setup

Let's say we have an interest-rate curve, no matter how it was calculated. As an example, I'll use a curve bootstrapped over the 6-months deposit, a strip of 6-months FRAs and a number of swaps against the 6-months Euribor. All the market inputs are stored in quotes so that their values can be changed.

```
In [3]: quotes = [ SimpleQuote(0.312/100) ]
        helpers = [ DepositRateHelper(QuoteHandle(quotes[0]),
                                      Period(6,Months), 3,
                                      TARGET(), Following, False, Actual360()) ]

        for rate, months_to_start in [(0.293, 1), (0.272, 2), (0.260, 3),
                                      (0.256, 4), (0.252, 5), (0.248, 6),
                                      (0.254, 7), (0.261, 8), (0.267, 9),
                                      (0.279, 10), (0.291, 11), (0.303, 12),
                                      (0.318, 13), (0.335, 14), (0.352, 15),
                                      (0.371, 16), (0.389, 17), (0.409, 18)]:
            quotes.append(SimpleQuote(rate/100))
            helpers.append(FraRateHelper(QuoteHandle(quotes[-1]),
                                         months_to_start, Euribor6M()))

        for rate, tenor in [(0.424, 3), (0.576, 4), (0.762, 5), (0.954, 6),
                            (1.135, 7), (1.303, 8), (1.452, 9), (1.584, 10),
                            (1.809, 12), (2.037, 15), (2.187, 20), (2.234, 25),
                            (2.256, 30), (2.295, 35), (2.348, 40), (2.421, 50),
                            (2.463, 60)]:
            quotes.append(SimpleQuote(rate/100))
            helpers.append(SwapRateHelper(QuoteHandle(quotes[-1]),
                                          Period(tenor, Years), TARGET(),
                                          Annual, Unadjusted, Thirty360(Thirty360.BondBasis),
                                          Euribor6M()))

        rate_curve = PiecewiseLogCubicDiscount(2, TARGET(), helpers, Actual365Fixed())
        curve_handle = RelinkableYieldTermStructureHandle(rate_curve)
```

Here's the curve, plotted over 15 years.

```
In [4]: plot_curve(rate_curve)
```



For illustration purposes, I'll be using the curve to price an interest-rate swap. Let's create a 12-years swap starting in one month, with an annual schedule for the fixed leg and a semiannual schedule for the floating leg. We'll use the curve above to forecast the floating-rate fixings, so we pass it to the `Euribor6M` instance that, in turn, we pass to the swap constructor.

```
In [5]: fixed_schedule = Schedule(Date(8, April, 2016), Date(8, April, 2028),
                                  Period(1, Years), TARGET(), Following, Following,
                                  DateGeneration.Forward, False)
        floating_schedule = Schedule(Date(8, April, 2016), Date(8, April, 2028),
                                     Period(6, Months), TARGET(), Following, Following,
                                     DateGeneration.Forward, False)
        index = Euribor6M(curve_handle)
        swap = VanillaSwap(VanillaSwap.Payer, 10000.0,
                           fixed_schedule, 0.02, Thirty360(),
                           floating_schedule, index, 0.0, Actual360())
```

Of course, we should use a different curve for discounting. But let me just skip that part for brevity, and simply pass the same curve to the engine used by the swap. The points I'm going to make won't suffer for this.

Once we've done this, we can finally get the value of the swap.

```
In [6]: swap.setPricingEngine(DiscountingSwapEngine(curve_handle))
        P0 = swap.NPV()
        print(P0)

Out[6]: -189.83267948709272
```

Now, let's say that this was you pricing a deal. And let's also say that you're interested (as you should) in how the swap price reacts to changes in the underlying rates.

Interest-rate sensitivities

If you're interested in the sensitivities of the price to the input rates, you can have them: shift any input rate by setting a perturbed value to the corresponding quote and recalculate the NPV. For instance, you can bump the 6-months deposit rate by one basis point and get the new price as follows:

```
In [7]: bp = 1.0e-4
        ref = quotes[0].value()
        quotes[0].setValue(ref+1*bp)
        print(swap.NPV())
        quotes[0].setValue(ref)

Out[7]: -190.1069970119836
```

(Also, don't forget to set the value back to the actual quoted rate when you're done).

This can be done for a single rate, as above, or for any number of rates; all of them, for instance...

```
In [8]: for q in quotes:
            q.setValue(q.value()+1*bp)
        print(swap.NPV())
        for q in quotes:
            q.setValue(q.value()-1*bp)

Out[8]: -178.68820577843826
```

...so the above gives you the swap price when all the input rates move 1 basis point upwards; the difference between the new price and the old one will give you the DV01 of the swap. (Note that, depending on how you define it, you might want to shift either the forecast curve, the discount curve, or both.)

Different combinations of changes can also give you different stress scenarios; for instance, ones in which the curve tilts in some direction, or ones in which you only move the short end or the long end of the curve. In doing so, though, you're constrained to use the nodes of the original curve. For instance, in the curve above there are no nodes between 20 and 25 years, thus there are no levers to pull in that interval.

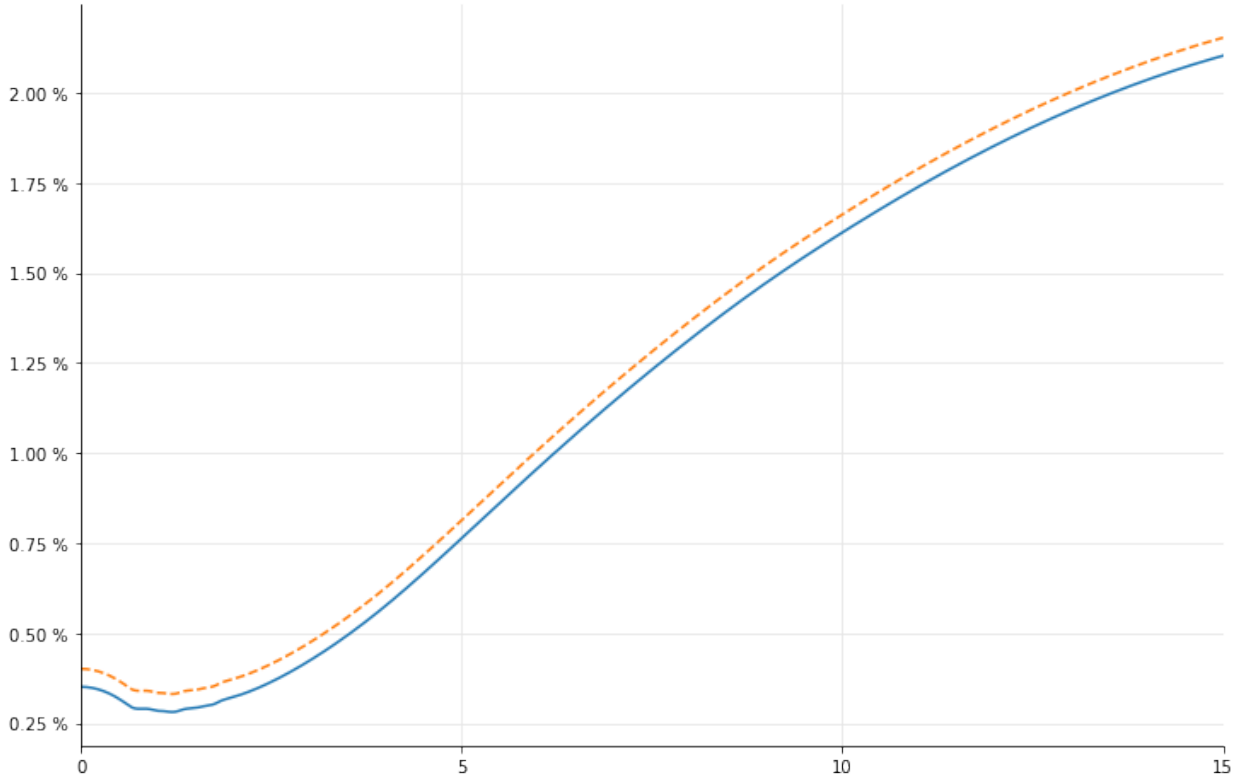
As an alternative, you can take an approach in which you modify the curve as a whole independently of the underlying rates. For instance, to shift all the zero rates upwards, you can keep the original curve as it is and add one basis point to all its zero rates by means of the `ZeroSpreadedTermStructure` class. To use it for pricing our swap, we'll store the original curve in a separate handle, add the spread, and link the new curve to the handle we're using for forecasting. As usual, the swap price will react accordingly.

```
In [9]: base_curve = YieldTermStructureHandle(rate_curve)
        spread = SimpleQuote(1*bp)

        curve_handle.linkTo(ZeroSpreadedTermStructure(base_curve, QuoteHandle(spread)))
        print(swap.NPV())

Out[9]: -178.8676404436867
```

As we could expect, the result is close to what we got by shifting all the input rates (the difference is just one or two digits in the first decimal place) but not quite the same: modifying, say, an input swap rate doesn't have the same effect as applying the same change to the zero rates directly. As usual, I'll trust you to know what you're doing in either case.

To get a more visual idea of what we're doing, we can also increase the spread and plot the resulting curve on top of the original one:

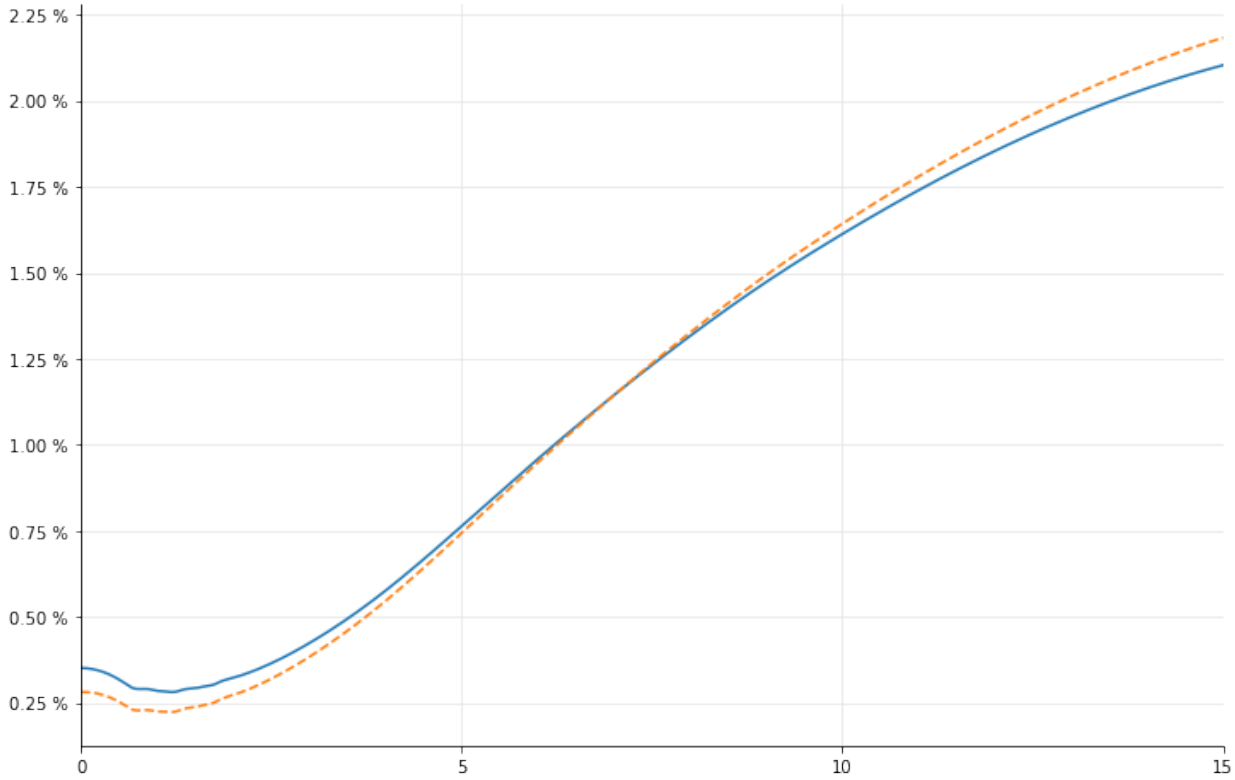
```
In [10]: spread.setValue(5*bp)
         plot_curves((rate_curve,'-'), (curve_handle,'--'))
```



Another class, `SpreadedLinearZeroInterpolatedTermStructure` (for whose name I apologize on behalf of the library developers: I don't know what came over us) allows one to apply a spread which is interpolated linearly between a set of nodes, which of course are independent of the nodes of the underlying curve.

For instance, we can create a scenario in which we tilt the curve by taking equally spaced dates each year between now and 20 years, and define the corresponding spreads as negative in the short end, increasing until they reach zero at the 7-years mark, and more and more positive in the long end. Again, we can plot the resulting curve for comparison with the original one…

```
In [11]: spot = rate_curve.referenceDate()
         dates = [ spot + Period(n,Years) for n in range(21)]
         spreads = [ QuoteHandle(SimpleQuote((n-7)*bp))  for n in range(21) ]

         curve_handle.linkTo(
             SpreadedLinearZeroInterpolatedTermStructure(base_curve, spreads, dates))
         plot_curves((rate_curve,'-'), (curve_handle,'--'))
```



...and again, we can ask the swap for its price under this scenario.

```
In [12]: print(swap.NPV())

Out[12]: -138.69485276964178
```

When using this technique, going back to the actual market quotes simply means linking the curve handle to the original curve:

```
In [13]: curve_handle.linkTo(rate_curve)
         print(swap.NPV())

Out[13]: -189.83267949204492
```

If you want more control of the shift (as asked, for instance, by user6142489 on *Stack Overflow*[1] who wanted to calculate key-rate risks) you might have to increase the number of nodes. If your nodes are one year apart as above, and if you modify, e.g., the node at 7 years, the interpolation scheme will cause the whole range between 6 and 8 years to change, and all coupons paid in that period to be affected. The more nodes you have and the closer they are together, the more localized any change will be.

[1] https://stackoverflow.com/questions/46279785/quantlib-building-key-rate-risks

14. A glitch in forward-rate curves

(Based on a question asked by Boris Chow[1] on the QuantLib mailing list. Thanks!)

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

In [2]: from QuantLib import *

In [3]: today = Date(24, August, 2015)
        Settings.instance().evaluationDate = today
```

The statement of the case

Let's say we have built an interpolated forward-rate curve, by which I mean that it interpolates instantaneous forward rates (for more details, read my other book). We're using a backward-flat interpolation, which corresponds to log-linear discount factors. The dates and forwards are entirely made up; they are just for show.

```
In [4]: dates = [ today ] + [ today + Period(i, Years)
                              for i in [1, 2, 3, 5, 10, 20] ]
        forwards = [ 0.01, 0.03, 0.02, 0.025, 0.035, 0.05, 0.04 ]
        curve = ForwardCurve(dates, forwards, Actual360())
```

We can ask the curve for its nodes, and it will return those we expect—that is, those we passed ourselves…

```
In [5]: DataFrame(list(curve.nodes()),
                  columns = ('date','rate'),
                  index = ['']*len(dates))

Out[5]:
```

[1] https://sourceforge.net/p/quantlib/mailman/message/34286980/

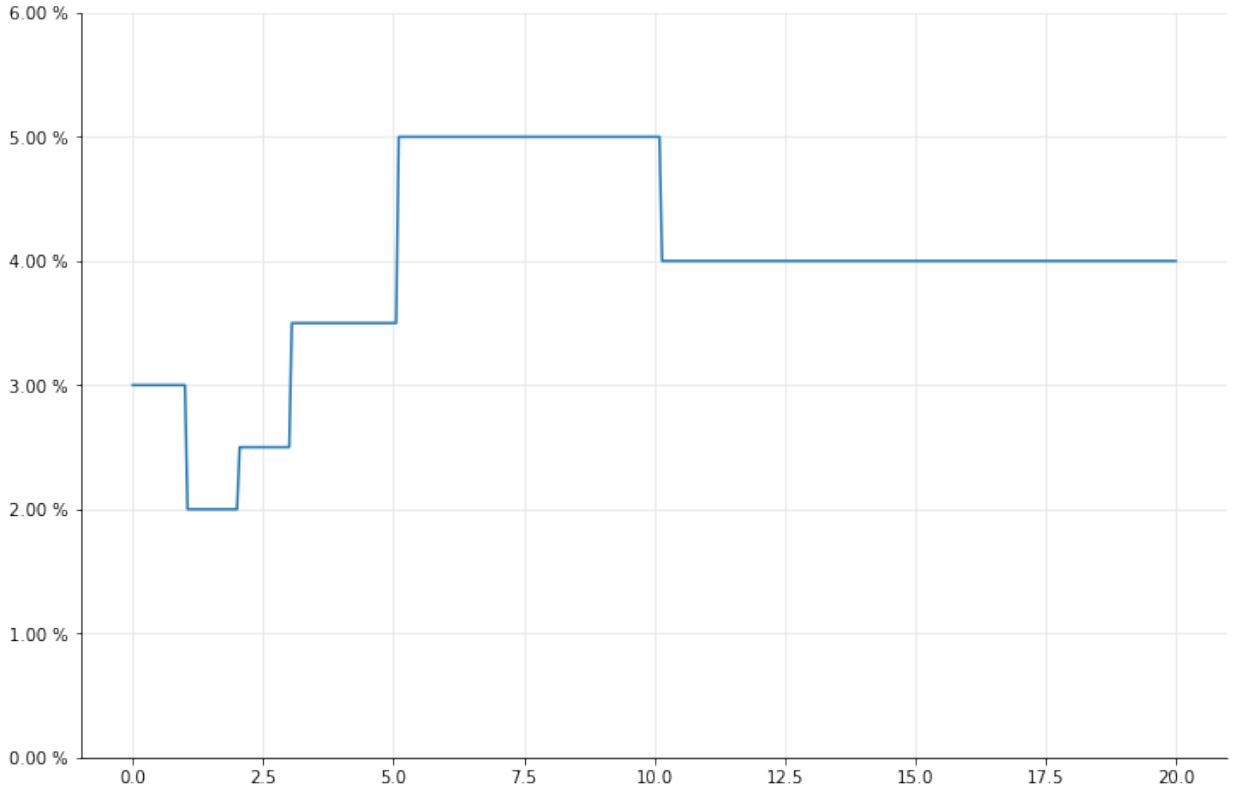
date	rate
August 24th, 2015	0.010
August 24th, 2016	0.030
August 24th, 2017	0.020
August 24th, 2018	0.025
August 24th, 2020	0.035
August 24th, 2025	0.050
August 24th, 2035	0.040

...and if we retrieve the instantaneous forward from a date between the nodes, it's the same as that of the following node, as expected for a backward-flat interpolation.

```
In [6]: d = today + Period(4,Years)
        print(d)
        print(curve.forwardRate(d, d, curve.dayCounter(), Continuous))

Out[6]: August 24th, 2019
        3.500000 % Actual/360 continuous compounding
```

We can even plot the whole thing and get the expected shape.

```
In [7]: sample_times = np.linspace(0.0, 20.0, 401)
        sample_rates = [ curve.forwardRate(t, t, Continuous).rate()
                         for t in sample_times ]

        f, ax = utils.plot()
        ax.set_ylim(0.0,0.06)
        ax.yaxis.set_major_formatter(utils.rate_formatter())
        ax.plot(sample_times, sample_rates);
```



So it seems all is well with the world. What if we retrieve the instantaneous forward rates at the curve nodes, though?

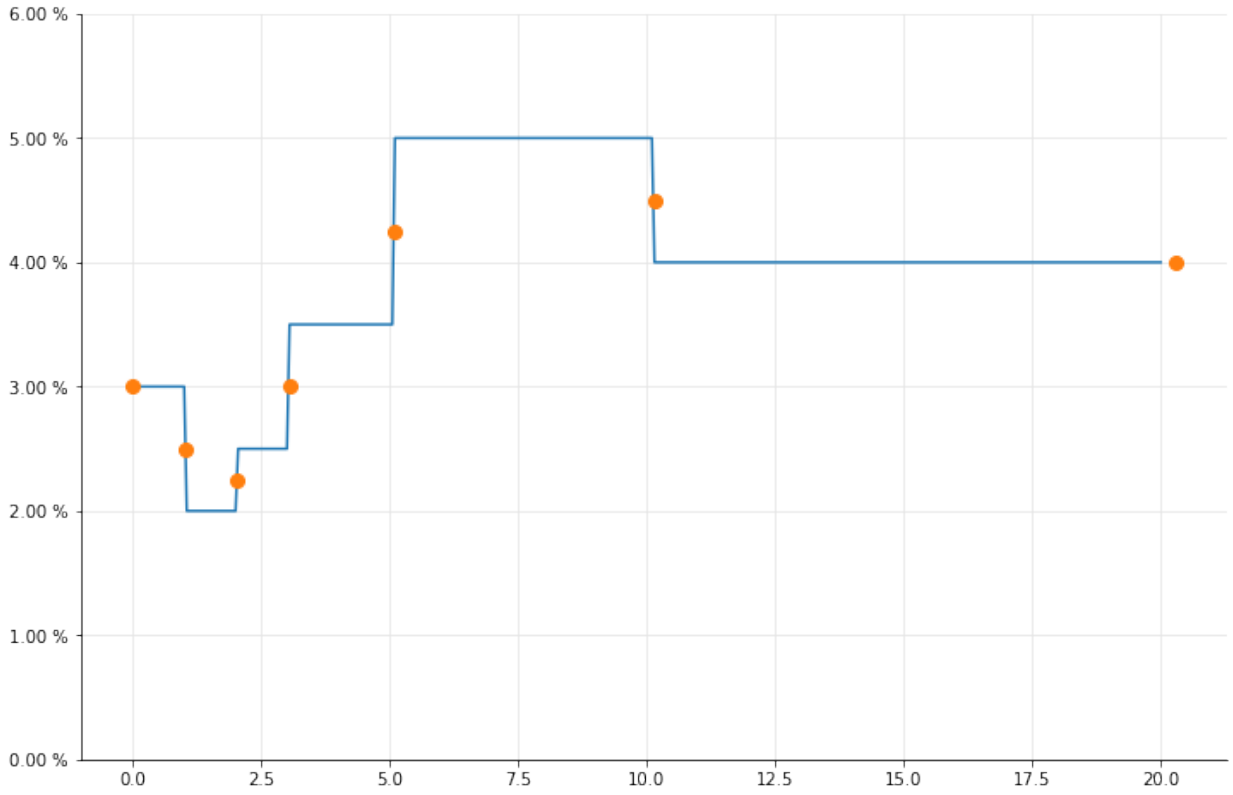
```
In [8]: dates, expected = zip(*curve.nodes())
        rates = [ curve.forwardRate(d, d, curve.dayCounter(), Continuous).rate()
                  for d in dates]
        DataFrame(list(zip(dates, expected, rates)),
                  columns = ('date','expected','retrieved'),
                  index = ['']*len(dates))
```

Out[8]:

date	expected	retrieved
August 24th, 2015	0.010	0.0300
August 24th, 2016	0.030	0.0250
August 24th, 2017	0.020	0.0225
August 24th, 2018	0.025	0.0300
August 24th, 2020	0.035	0.0425
August 24th, 2025	0.050	0.0450
August 24th, 2035	0.040	0.0400

Here are the above points, together with the rest of the curve.

```
In [9]: node_times = [ curve.dayCounter().yearFraction(today, d) for d in dates ]
        ax.plot(node_times, rates, 'o', markersize=8)
        display(f)
```



What's wrong?

It's a combination of two things. First, the particular interpolation we've chosen causes the instantaneous forwards to be discontinuous at the nodes. Second, there's a limitation in the implementation of the base `TermStructure` class: the instantaneous forwards are not taken directly from the interpolation, but retrieved generically from the discount factors as the forward over a small interval around the given time; that is,

$$\tilde{f}(t) = \frac{1}{2\delta t} \log\left(\frac{B(t-\delta t)}{B(t+\delta t)}\right)$$

Again, the details are in my other book.

By writing the discount factors $B(t)$ in terms of the zero rates as $\exp(Z(t) \cdot t)$, and in turn the zero rates in terms of the instantaneous forwards as $Z(t) = \int_0^t f(\tau)d\tau$, the above simplifies (well, for some value of "simplifies") to

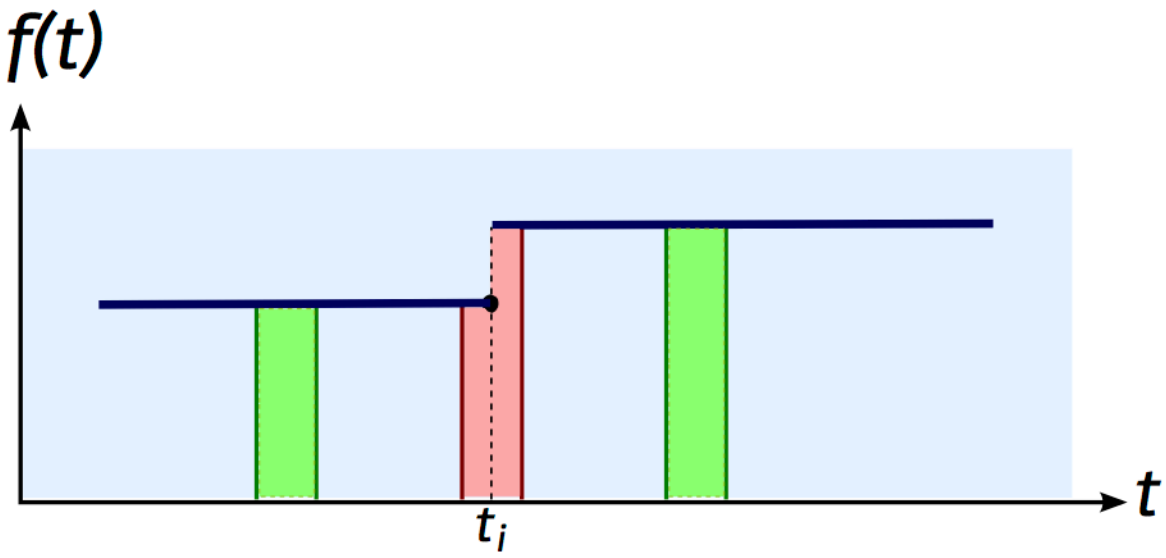
$$\tilde{f}(t) = \frac{1}{2\delta t}\left[\int_0^{t+\delta t} f(\tau)d\tau - \int_0^{t-\delta t} f(\tau)d\tau\right]$$

We can interpret the above expression in two ways; both explain why the values at the nodes are off and why we get the correct values elsewhere.

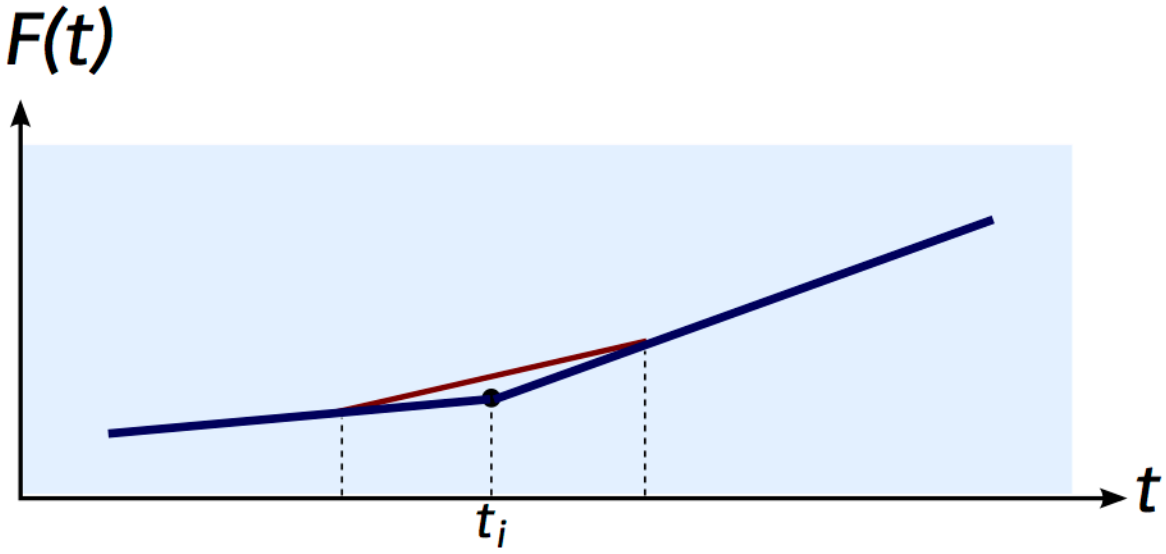
As the difference of two integrals, it equals

$$\frac{1}{2\delta t}\left[\int_{t-\delta t}^{t+\delta t} f(\tau)d\tau\right]$$

that is, the average of $f(\tau)$ between $t-\delta t$ and $t+\delta t$. What this means is clear from the following figure: off the nodes, the result equals the flat value of the forwards; at the nodes, though, it equals the average between the two adjacent levels.



If we consider the integral $\int_0^t f(\tau)d\tau$ as a function $F(t)$ instead, $\tilde{f}(t)$ equals $\frac{F(t+\delta t)-F(t-\delta t)}{2\delta t}$; that is, the numerical derivative of F at t. Again, a figure shows clearly what happens at and off the nodes: the forwards are piecewise flat, their integral is piecewise linear with slopes equal to the forwards, and the derivative at a given node is in between the two joining slopes.



Is the curve wrong, then?

Yes and no. The glitch above is real, but discount factors and discrete rates are retrieved correctly so there's no problem using the curve (unless the value of an instrument depends on instantaneous forwards, but that's unlikely). If the above troubles you, though, what you can do is simply to choose another interpolation that doesn't cause discontinuities.

Interest-rate models

15. Simulating interest rates using Hull White model

The Hull-White Short Rate Model is defined as:

$$dr_t = (\theta(t) - ar_t)dt + \sigma dW_t$$

where a and σ are constants, and $\theta(t)$ is chosen in order to fit the input term structure of interest rates. Here we use QuantLib to show how to simulate the Hull-White model and investigate some of the properties.

We import the libraries and set things up as shown below:

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

The constants that we use for this example is all defined as shown below. Variables `sigma` and `a` are the constants that define the Hull-White model. In the simulation, we discretize the time span of `length` 30 years into 360 intervals (one per month) as defined by the `timestep` variable. For simplicity we will use a constant forward rate term structure as an input. It is straight forward to swap with another term structure here.

```
In [2]: sigma = 0.1
        a = 0.1
        timestep = 360
        length = 30 # in years
        forward_rate = 0.05
        day_count = Thirty360()
        todays_date = Date(15, 1, 2015)

In [3]: Settings.instance().evaluationDate = todays_date

        spot_curve = FlatForward(todays_date,
                                 QuoteHandle(SimpleQuote(forward_rate)),
                                 day_count)
        spot_curve_handle = YieldTermStructureHandle(spot_curve)

In [4]: hw_process = HullWhiteProcess(spot_curve_handle, a, sigma)
```

```
        rng = GaussianRandomSequenceGenerator(
            UniformRandomSequenceGenerator(timestep, UniformRandomGenerator()))
        seq = GaussianPathGenerator(hw_process, length, timestep, rng, False)
```

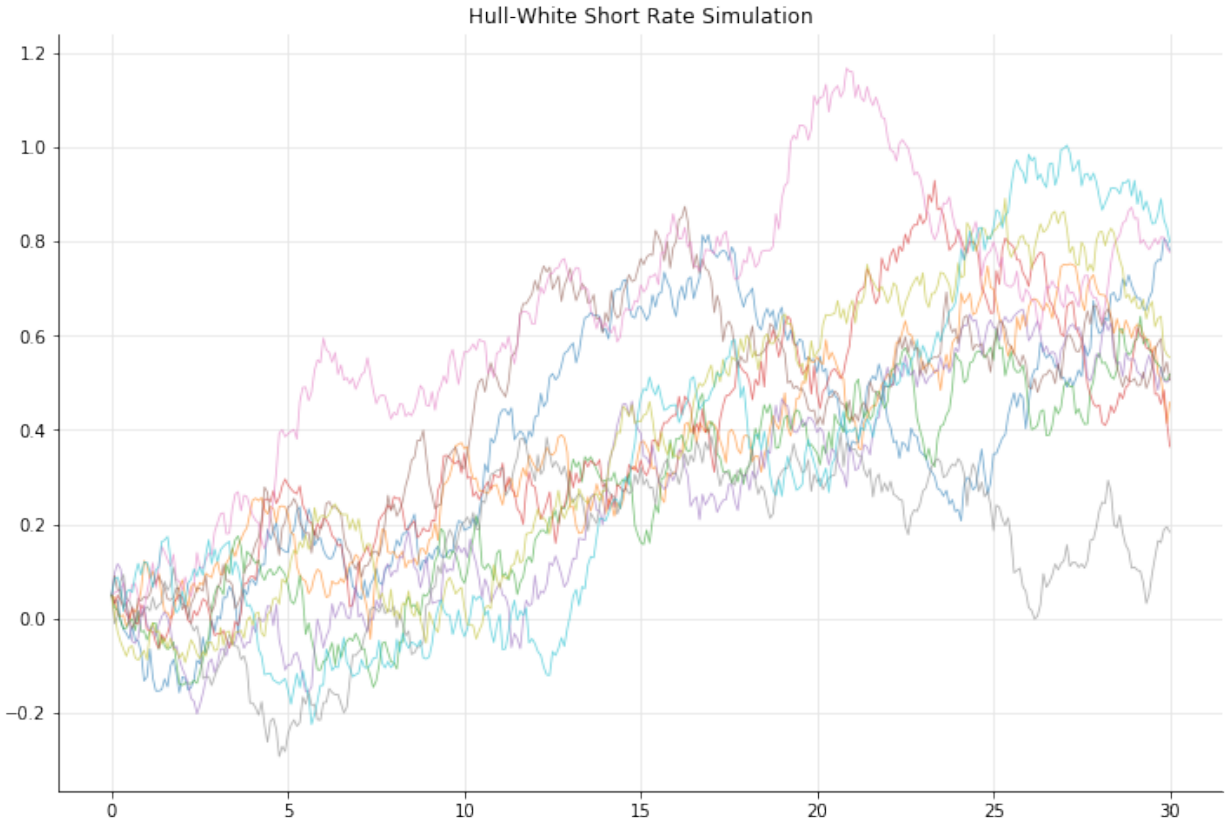
The Hull-White process is constructed by passing the term-structure, `a` and `sigma`. To create the path generator, one has to provide a random sequence generator along with other simulation inputs such as `timestep` and 'length.

A function to generate paths can be written as shown below:

```
In [5]: def generate_paths(num_paths, timestep):
            arr = np.zeros((num_paths, timestep+1))
            for i in range(num_paths):
                sample_path = seq.next()
                path = sample_path.value()
                time = [path.time(j) for j in range(len(path))]
                value = [path[j] for j in range(len(path))]
                arr[i, :] = np.array(value)
            return np.array(time), arr
```

The simulation of the short rates look as shown below:

```
In [6]: num_paths = 10
        time, paths = generate_paths(num_paths, timestep)
        fig, ax = utils.plot()
        for i in range(num_paths):
            ax.plot(time, paths[i, :], lw=0.8, alpha=0.6)
        ax.set_title("Hull-White Short Rate Simulation");
```



The short rate $r(t)$ is given a distribution with the properties:

$$E\{r(t)|F_s\} = r(s)e^{-a(t-s)} + \alpha(t) - \alpha(s)e^{-a(t-s)} \quad (15..1)$$

$$Var\{r(t)|F_s\} = \frac{\sigma^2}{2a}[1 - e^{-2a(t-s)}] \quad (15..2)$$

where

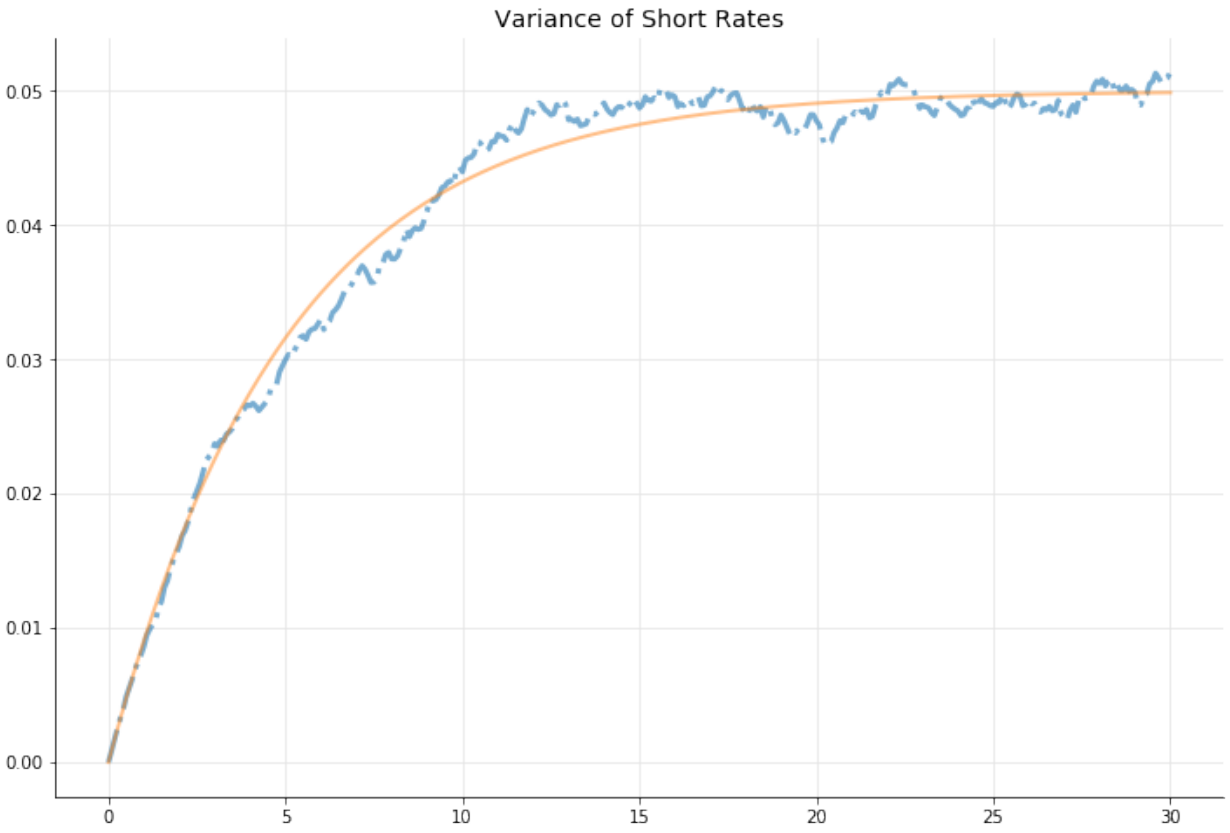
$$\alpha(t) = f^M(0,t) + \frac{\sigma^2}{2a^2}(1 - e^{-at})^2$$

as shown in Brigo & Mercurio's book on Interest Rate Models.

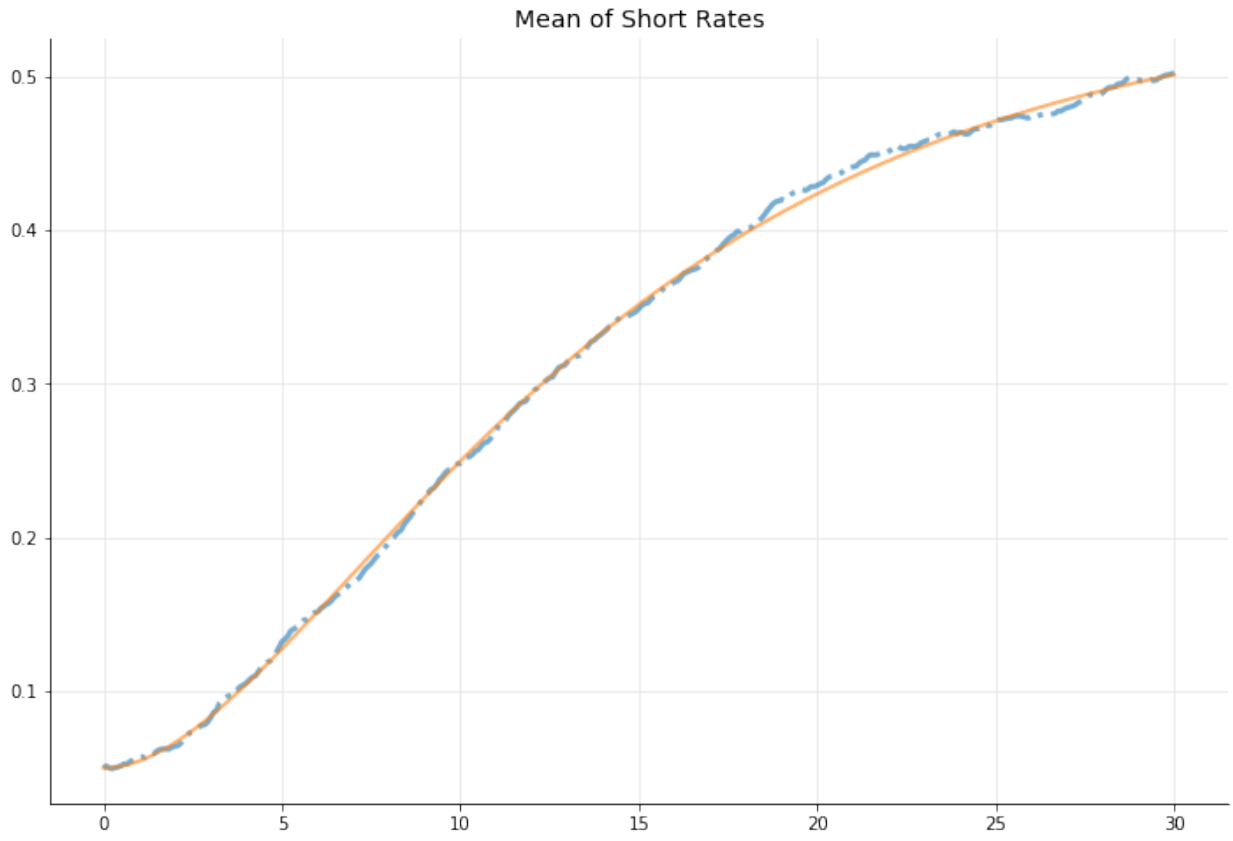
```
In [7]: num_paths = 1000
        time, paths = generate_paths(num_paths, timestep)
```

The mean and variance compared between the simulation (red dotted line) and theory (blue line).

```
In [8]: vol = [np.var(paths[:, i]) for i in range(timestep+1)]
        fig, ax = utils.plot()
        ax.plot(time, vol, "-.", lw=3, alpha=0.6)
        ax.plot(time,sigma*sigma/(2*a)*(1.0-np.exp(-2.0*a*np.array(time))), "-",
                lw=2, alpha=0.5)
        ax.set_title("Variance of Short Rates", size=14);
```




```
In [9]: def alpha(forward, sigma, a, t):
            return forward + 0.5* np.power(sigma/a*(1.0 - np.exp(-a*t)), 2)

        avg = [np.mean(paths[:, i]) for i in range(timestep+1)]
        fig, ax = utils.plot()
        ax.plot(time, avg, "-.", lw=3, alpha=0.6)
        ax.plot(time,alpha(forward_rate, sigma, a, time), "-", lw=2, alpha=0.6)
        ax.set_title("Mean of Short Rates", size=14);
```


Mean of Short Rates
0.5
0.4
0.3
0.2
0.1
0
5
10
15
20
25
30

16. Thoughts on the convergence of Hull-White model Monte Carlo simulations

I had recently written an introductory post on simulating short rates in the Hull-White Model[1]. This question on the QuantLib forum[2] raised some interesting questions on the convergence of the Hull-White model simulations. In this post, I discuss the convergence of Monte Carlo simulations using the Hull-White model.

The Hull-White Short Rate Model is defined as:

$$dr_t = (\theta(t) - ar_t)dt + \sigma dW_t$$

where a and σ are constants, and $\theta(t)$ is chosen in order to fit the input term structure of interest rates. Here we use QuantLib to show how to simulate the Hull-White model and investigate some of the properties.

The variables used in this post are described below: - `timestep` is the number of steps used to discretize the time grid - `hw_process` the object that defines the Hull-White process, - `length` is the time span of the simulation in years - `low_discrepancy` is a boolean variable that is used to chose Sobol low discrepancy random or not - `brownian_bridge` is a boolean that chooses brownian bridge for path generation - `num_paths` is the number of paths in the simulation - `a` is the constant parameter in the Hull-White model - `sigma` is the constant parameter σ in the Hull-White model that describes volatility

```
In [1]: import QuantLib as ql
        import matplotlib.pyplot as plt
        import numpy as np
        from scipy.integrate import simps, cumtrapz, romb
        %matplotlib inline
        import math
        import utils

        todays_date = ql.Date(15, 1, 2015)
        ql.Settings.instance().evaluationDate = todays_date
```

The `get_path_generator` function creates the a path generator. This function takes various inputs such as

[1] http://gouthamanbalaraman.com/blog/hull-white-simulation-quantlib-python.html
[2] http://quantlib.10058.n7.nabble.com/Matching-results-between-HW-tree-and-simulation-models-td16399.html

```
In [2]: def get_path_generator(timestep, hw_process, length,
                               low_discrepancy=False, brownian_bridge=True):
            """
            Returns a path generator
            """
            if low_discrepancy:
                usg = ql.UniformLowDiscrepancySequenceGenerator(timestep)
                rng = ql.GaussianLowDiscrepancySequenceGenerator(usg)
                seq = ql.GaussianSobolPathGenerator(
                        hw_process, length, timestep, rng,brownian_bridge)
            else:
                usg = ql.UniformRandomSequenceGenerator(timestep,
                                                        ql.UniformRandomGenerator())
                rng = ql.GaussianRandomSequenceGenerator(usg)
                seq = ql.GaussianPathGenerator(
                        hw_process, length, timestep, rng, brownian_bridge)
            return seq
```

The `generate_paths` function uses the generic path generator produced by the `get_path_generator` function to return a tuple of the array of the points in the time grid and a matrix of the short rates generated.

```
In [3]: def generate_paths(num_paths, timestep, seq):
            arr = np.zeros((num_paths, timestep+1))
            for i in range(num_paths):
                sample_path = seq.next()
                path = sample_path.value()
                time = [path.time(j) for j in range(len(path))]
                value = [path[j] for j in range(len(path))]
                arr[i, :] = np.array(value)
            return np.array(time), arr
```

The `generate_paths_zero_price` essentially is a wrapper around `generate_path_generator` and `generate_paths` taking all the required raw inputs. This function returns the average of zero prices from all the paths for different points in time. I wrote this out so that I can conveniently change all the required inputs and easily plot the results.

```
In [4]: def generate_paths_zero_price(spot_curve_handle, a, sigma, timestep, length,
                                      num_paths, avg_grid_array, low_discrepancy=False,
                                      brownian_bridge=True):
            """
            This function returns a tuple (T_array, F_array), where T_array is the array
            of points in the time grid, and F_array is the array of the average of zero
            prices observed from the simulation.
            """
            hw_process = ql.HullWhiteProcess(spot_curve_handle, a, sigma)
            seq = get_path_generator(
                    timestep, hw_process, length, low_discrepancy, brownian_bridge
            )
            time, paths = generate_paths(num_paths, timestep, seq)
            avgs = [(time[j],(np.mean([math.exp(-simps(paths[i][0:j], time[0:j]))
                                       for i in range(num_paths)])))
                    for j in avg_grid_array
                    ]
            return zip(*avgs)

        def generate_paths_discount_factors(spot_curve_handle, a, sigma, timestep, length,
                                      num_paths, avg_grid_array, low_discrepancy=False,
                                      brownian_bridge=True):
            """
            This function returns a tuple (T_array, S_matrix), where T_array is the array
            of points in the time grid, and S_matrix is the matrix of the spot rates for
            each path in the different points in the time grid.
            """
            hw_process = ql.HullWhiteProcess(spot_curve_handle, a, sigma)
            seq = get_path_generator(
                    timestep, hw_process, length, low_discrepancy, brownian_bridge
            )
            time, paths = generate_paths(num_paths, timestep, seq)
            arr = np.zeros((num_paths, len(avg_grid_array)))
            for i in range(num_paths):
                arr[i, :] =  [np.exp(-simps(paths[i][0:j], time[0:j]))
                              for j in avg_grid_array ]
            t_array = [time[j] for j in avg_grid_array]
            return t_array, arr

        def V(t,T, a, sigma):
            """ Variance of the integral of short rates, used below"""
            return sigma*sigma/a/a*(T-t + 2.0/a*math.exp(-a*(T-t)) -
                                    1.0/(2.0*a)*math.exp(-2.0*a*(T-t)) - 3.0/(2.0*a) )
```

Factors affecting the convergence

In order to understand the convergence of Monte Carlo for the Hull-White model, let us compare the market discount factor,

$$P^M(t,T) = \exp\left(-\int_t^T f^M(t,u)du\right)$$

with the expectation of the discount factors from the sample of Monte Carlo paths,
$P^{MC}(t,T) = E_t\left\{e^{-\int_t^T r^{MC}(u)du}\right\}$.

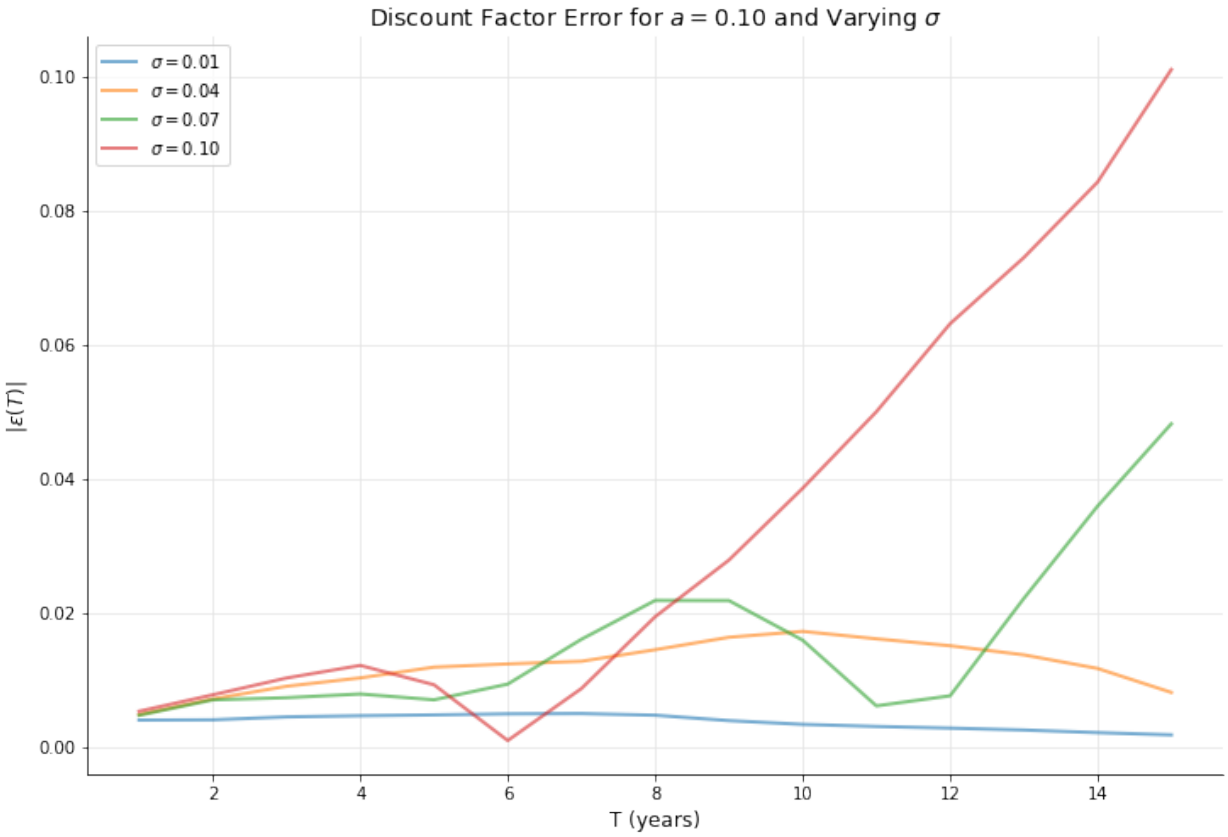
Here $f^M(t,T)$ is the instantaneous forward rate implied by the market, and $r^{MC}(s)$ is the instantaneous short rate from the Monte Carlo simulations. The error in the Monte Carlo simulation can be defined as:

$$\epsilon(T) = P^M(0,T) - P^{MC}(0,T)$$

As a first step, let us look at the plots of $\epsilon(t)$ for different values of a and σ.

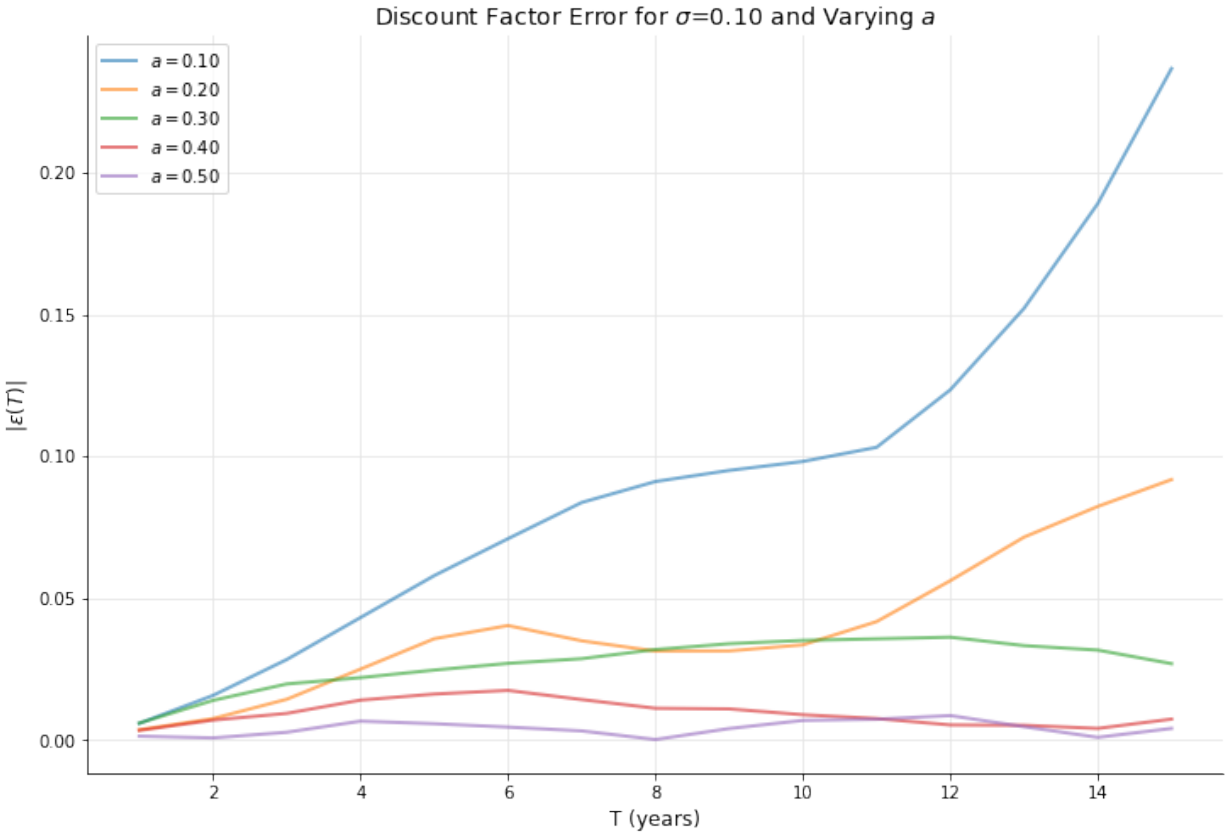
```
In [5]: # Here we vary sigma with fixed a and observe the error epsilon
        # define constants
        num_paths = 500
        sigma_array = np.arange(0.01,0.1,0.03)
        a = 0.1
        timestep = 180
        length = 15 # in years
        forward_rate = 0.05
        day_count = ql.Thirty360()
        avg_grid_array = np.arange(12, timestep+1, 12)

        # generate spot curve
        spot_curve = ql.FlatForward(
            todays_date,
            ql.QuoteHandle(ql.SimpleQuote(forward_rate)), day_count
        )
        spot_curve_handle = ql.YieldTermStructureHandle(spot_curve)

        #initialize plots
        figure, axis = utils.plot()
        plots = []
        zero_price_theory = np.array([spot_curve.discount(j*float(length)/float(timestep))
                                      for j in avg_grid_array])
        for sigma in sigma_array:
            term, zero_price_empirical = generate_paths_zero_price(
```

```
            spot_curve_handle, a, sigma, timestep, length, num_paths,
            avg_grid_array
        )
        plots += axis.plot(
            term, np.abs(zero_price_theory - np.array(zero_price_empirical)),
            lw=2, alpha=0.6,
            label="$\sigma=%.2f$"%sigma
        )

    # plot legend
    labels = [p.get_label() for p in plots]
    legend = axis.legend(plots, labels, loc=0)
    axis.set_xlabel("T (years)", size=12)
    axis.set_ylabel("|$\epsilon(T)$|", size=12)
    axis.set_title("Discount Factor Error for $a=$%0.2f and Varying $\sigma$"%a,
                   size=14);
```



The above plot illustrates that for $\sigma = 0.01$, the Monte Carlo model shows good convergence, and the convergence gradually deteriorates as σ increases and approaches a.

In [6]:
```
# Here we vary a with fixed sigma and observe the error epsilon
#define constants
num_paths = 500
sigma = 0.1
a_array = np.arange(0.1, 0.51, 0.1)
timestep = 180
length = 15 # in years
forward_rate = 0.05
day_count = ql.Thirty360()
avg_grid_array = np.arange(12, timestep+1, 12)

# generate spot curve
spot_curve = ql.FlatForward(
    todays_date,
    ql.QuoteHandle(ql.SimpleQuote(forward_rate)), day_count
)
spot_curve_handle = ql.YieldTermStructureHandle(spot_curve)

#initialize plots
figure, axis = utils.plot()
plots = []
zero_price_theory = np.array([spot_curve.discount(j*float(length)/float(timestep))
                              for j in avg_grid_array])
for a in a_array:
    term, zero_price_empirical = generate_paths_zero_price(
        spot_curve_handle, a, sigma, timestep, length, num_paths,
        avg_grid_array
    )
    plots += axis.plot(
        term,np.abs(zero_price_theory - np.array(zero_price_empirical)),
        lw=2, alpha=0.6,
        label="$a=%.2f$"%a
    )

# plot legend
labels = [p.get_label() for p in plots]
legend =axis.legend(plots,labels, loc=0)
axis.set_xlabel("T (years)", size=12)
axis.set_ylabel("|$\epsilon(T)$|", size=12)
axis.set_title("Discount Factor Error for $\sigma$=%0.2f and Varying $a$"%sigma,
               size=14);
```



The above plot illustrates that for $a = 0.1$ the convergence of Monte Carlo is poor, and it gradually improves as a increases more than σ.

From the plots above, we observe that the convergence is good if the ratio $\sigma/a < 1$, and the convergence deteriorates as the ratio σ/a increases above unity. Now, let us try to formalize this observation from the theoretical footing of the Hull-White model.

Distribution of Discount Factors

The Monte Carlo approach estimates the market discount factor as the expectation of discount factors from each Monte Carlo path. If distribution of discount factors has a standard deviation σ_D, then the error in our estimate of $P^{MC}(t,T)$ on using N paths will be of the order of:

$$\epsilon(t,T) \approx \frac{\sigma_D}{\sqrt{(N)}}.$$

In other words, there are two factors at play in our Monte Carlo estimate, the number of Monte Carlo paths N and the standard deviation of the distribution of discount factors σ. Using more Monte Carlo paths will lead to improved convergence. But at the same time, the σ_D has to be relatively small for us to get a good estimate.

The integral of short rates can be shown to be normally distributed (refer Brigo-Mercurio, second edition page 75), and is given as

$$\int_t^T r(u)du|\mathcal{F}_t \sim \mathcal{N}\left(B(t,T)[r(t)-\alpha(t)] + \ln\frac{P^M(0,t)}{P^M(0,T)} + \frac{1}{2}[V(0,T)-V(0,t)], V(t,T)\right)$$

where,

$$B(t,T) = \frac{1}{a}\left[1-e^{-a(T-t)}\right] \quad (16..1)$$

$$V(t,T) = \frac{\sigma^2}{a^2}\left[T-t+\frac{2}{a}e^{-a(T-t)} - \frac{1}{2a}e^{-2a(T-t)} - \frac{3}{2a}\right] \quad (16..2)$$

Based on this result, the discount factor from the Monte Carlo simulation of short rates

$$P^{MC}(t,T) = \exp\left(-\int_t^T r(u)du|\mathcal{F}_t\right)$$

will have a log-normal distribution with a standard deviation

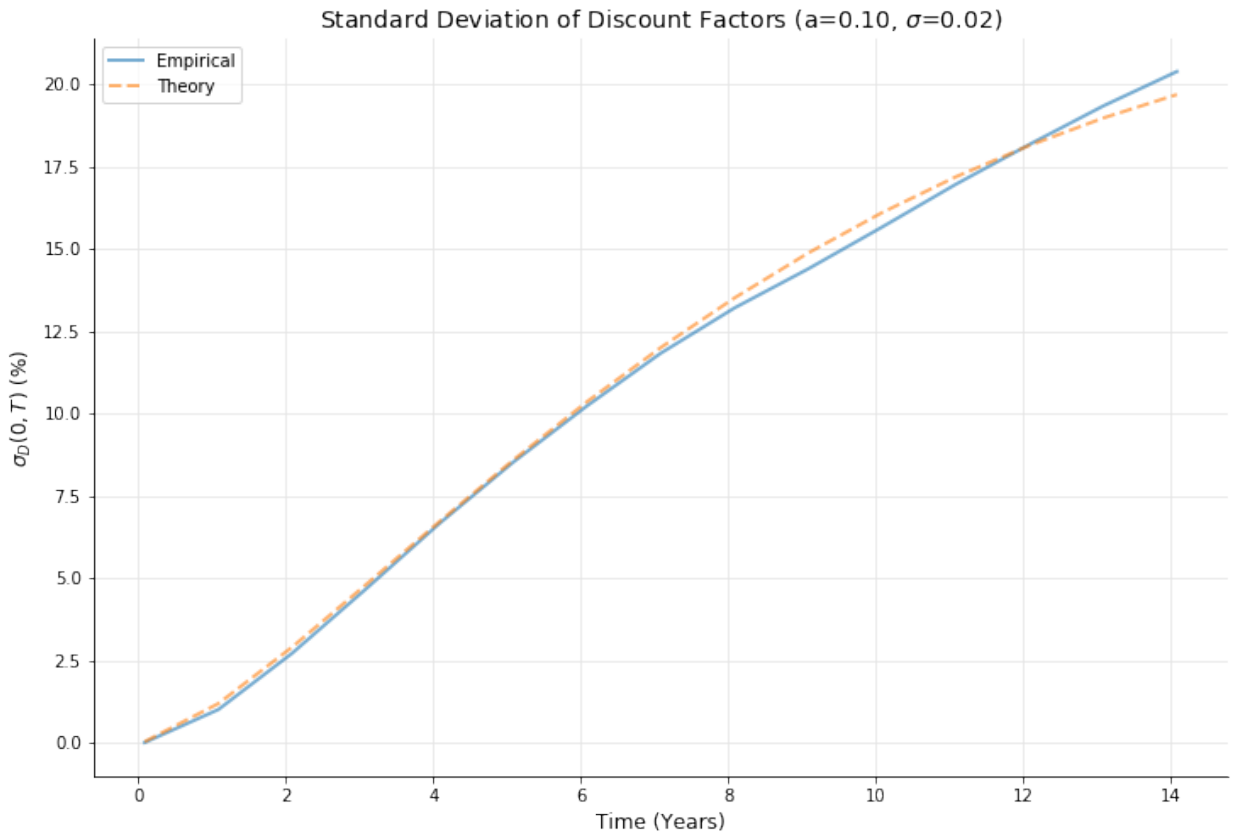
$$\sigma_D(t,T) = P^M(t,T)\sqrt{e^{V(t,T)}-1}$$

This result follows from the fact that if X is a random process with a normal distribution having mean μ and standard deviation σ, then log-normal distribution[3] $Y = e^X$ will satisfy:

[3] http://en.wikipedia.org/wiki/Log-normal_distribution

$$E(Y) = e^{\mu+\sigma^2/2} \tag{16..3}$$
$$Var(Y) = (e^{\sigma^2}-1)E(Y)^2 \tag{16..4}$$

```
In [7]: #define constants
        num_paths = 500
        sigma = 0.02
        a = 0.1
        timestep = 180
        length = 15 # in years
        forward_rate = 0.05
        day_count = ql.Thirty360()
        avg_grid_array = np.arange(1, timestep+1, 12)

        # generate spot curve
        spot_curve = ql.FlatForward(
            todays_date,
            ql.QuoteHandle(ql.SimpleQuote(forward_rate)), day_count
        )
        spot_curve_handle = ql.YieldTermStructureHandle(spot_curve)

        term, discount_factor_matrix = generate_paths_discount_factors(
                spot_curve_handle, a, sigma, timestep, length, num_paths,
                avg_grid_array
            )

        fig, axis = utils.plot()

        vol = [np.var(discount_factor_matrix[:, i]) for i in range(len(term))]
        l1 = axis.plot(term, 100*np.sqrt(vol),"-", lw=2, alpha=0.6,label="Empirical")
        vol_theory = [100*np.sqrt(math.exp(V(0,T,a, sigma))-1.0) *
                      spot_curve_handle.discount(T) for T in term]
        l2 = axis.plot(term, vol_theory,"--", lw=2, alpha=0.6,label="Theory")

        plots = l1+l2
        labels = [p.get_label() for p in plots]
        legend =plt.legend(plots,labels, loc=0)
        axis.set_xlabel("Time (Years)", size=12)
        axis.set_ylabel("$\sigma_D(0,T)$ (%)", size=12)
        axis.set_title("Standard Deviation of Discount Factors "
                       "(a=%0.2f, $\sigma$=%0.2f)"%(a, sigma), size=14);
```



The plot above compares the standard deviation of the discount factors σ_D from the closed form expression with a Monte Carlo estimate. The empirical estimate is in agreement with the theoretical expectation. We can estimate the value of σ_D for the asymptotic limit of $T \to \infty$:

$$\sigma_D(0,T) \approx P^M(0,T) e^{f^M(0,T)/a - \sigma^2/(4a^3)} \sqrt{e^{\sigma^2 T/a^2} - 1}$$

The exponential term, $e^{\sigma^2 T/a^2}$, can become very large when $\sigma^2 T/a^2$ grows above 1. Thus we can expect good convergence when $\sigma^2 T/a^2$ remains small or close to zero for the time T of interest to us.

The above result suggests that if the parameters σ and a are not chosen carefully, (i.e. $\sigma/a > 1$) then the convergence of the simulation would be poor and the results untrustworthy.

17. Short interest rate model calibration

In the earlier chapters, we have discussed simulating Hull-White model. That exercise gave a primer on how to use the model classes. There the model parameters were assumed to be given. However in practice, the model parameters need to calibrated from market data. Typically instruments such as swaptions, caps or floors and their market prices / volatilities are taken as inputs. Then the model parameters are fit in such a way that the model prices for these options are close enough. The goodness of fit depends, apart from the choice of the numerical methods, on the type of model itself. This is because models such as Hull-White 1 factor cannot fit some of the humped volatility term structures observed in the market. Never the less, Hull-White is usually a good starting point to understand calibration process.

Here we will discuss Hull-White model in detail. Then we will also show how the same procedure can be applied to calibrate other short rate models. We will assume the quotes to be at-the-money (ATM) log-normal volatilities. In the later section, we will extend to normal volatilities.

```
In [1]: from QuantLib import *
        from collections import namedtuple
        import math
        from pandas import DataFrame

        # This is for compatibility with QuantLib < 1.15
        try:
            BlackCalibrationHelper
        except:
            BlackCalibrationHelper = CalibrationHelper
```

Hull-White 1-Factor Model

Hull-White model was one of the first practical exogenous models that attempted to fit to the market interest rate term structures. The model is described as:

$$dr_t = (\theta(t) - ar_t)dt + \sigma dW_t$$

where a is the mean reversion constant, σ is the volatility parameter. The parameter $\theta(t)$ is chosen in order to fit the input term structure of interest rates.

What is the "right" value for parameters a and σ? This is the question that we address by calibrating to market instruments.

```
In [2]: today = Date(15, February, 2002);
        settlement= Date(19, February, 2002);
        Settings.instance().evaluationDate = today;
        term_structure = YieldTermStructureHandle(
            FlatForward(settlement,0.04875825,Actual365Fixed())
            )
        index = Euribor1Y(term_structure)
```

In this example we are going to calibrate given the starting tenor, months to maturity, and the swaption volatilities as shown below.

```
In [3]: CalibrationData = namedtuple("CalibrationData",
                                    "start, length, volatility")
        data = [CalibrationData(1, 5, 0.1148),
                CalibrationData(2, 4, 0.1108),
                CalibrationData(3, 3, 0.1070),
                CalibrationData(4, 2, 0.1021),
                CalibrationData(5, 1, 0.1000 )]
```

In order to make the code succinct in the various examples, we will create two functions. Function `create_swaption_helpers` takes all the swaption data, the index such as `Euribor1Y`, the term structure and the pricing engine, and returns a list of `SwaptionHelper` objects. The `calibration_report` evaluates the calibration by comparing the model price and implied volatilities with the Black price and market volatilities.

```
In [4]: def create_swaption_helpers(data, index, term_structure, engine):
            swaptions = []
            fixed_leg_tenor = Period(1, Years)
            fixed_leg_daycounter = Actual360()
            floating_leg_daycounter = Actual360()
            for d in data:
                vol_handle = QuoteHandle(SimpleQuote(d.volatility))
                helper = SwaptionHelper(Period(d.start, Years),
                                        Period(d.length, Years),
                                        vol_handle,
                                        index,
                                        fixed_leg_tenor,
                                        fixed_leg_daycounter,
                                        floating_leg_daycounter,
                                        term_structure
                                        )
                helper.setPricingEngine(engine)
                swaptions.append(helper)
            return swaptions
```

```
        def calibration_report(swaptions, data):
            columns = ["Model Price", "Market Price", "Implied Vol", "Market Vol",
                       "Rel Error Price", "Rel Error Vols"]
            report_data = []
            cum_err = 0.0
            cum_err2 = 0.0
            for i, s in enumerate(swaptions):
                model_price = s.modelValue()
                market_vol = data[i].volatility
                black_price = s.blackPrice(market_vol)
                rel_error = model_price/black_price - 1.0
                implied_vol = s.impliedVolatility(model_price,
                                                  1e-5, 50, 0.0, 0.50)
                rel_error2 = implied_vol/market_vol-1.0
                cum_err += rel_error*rel_error
                cum_err2 += rel_error2*rel_error2

                report_data.append((model_price, black_price, implied_vol,
                                    market_vol, rel_error, rel_error2))
            print("Cumulative Error Price: %7.5f" % math.sqrt(cum_err))
            print("Cumulative Error Vols : %7.5f" % math.sqrt(cum_err2))
            return DataFrame(report_data,columns= columns, index=['']*len(report_data))
```

Calibrating Reversion and Volaitility

Here we use the `JamshidianSwaptionEngine` to value the swaptions as part of calibration. The `JamshidianSwaptionEngine` requires one-factor affine models as input. For other interest rate models, we need a pricing engine that is more suited to those models.

```
In [5]: model = HullWhite(term_structure);
        engine = JamshidianSwaptionEngine(model)
        swaptions = create_swaption_helpers(data, index, term_structure, engine)

        optimization_method = LevenbergMarquardt(1.0e-8,1.0e-8,1.0e-8)
        end_criteria = EndCriteria(10000, 100, 1e-6, 1e-8, 1e-8)
        model.calibrate(swaptions, optimization_method, end_criteria)

        a, sigma = model.params()
        print("a = %6.5f, sigma = %6.5f" % (a, sigma))

Out[5]: a = 0.04642, sigma = 0.00580

In [6]: calibration_report(swaptions, data)

Out[6]: Cumulative Error Price: 0.11583
        Cumulative Error Vols : 0.11614
```

Out[6]:

Model Price	Market Price	Implied Vol	Market Vol	Rel Error Price	Rel Error Vols
0.008775	0.009485	0.106198	0.1148	-0.074854	-0.074928
0.009669	0.010078	0.106292	0.1108	-0.040610	-0.040688
0.008663	0.008716	0.106343	0.1070	-0.006138	-0.006138
0.006490	0.006226	0.106442	0.1021	0.042367	0.042525
0.003542	0.003323	0.106612	0.1000	0.065817	0.066122

Calibrating Volatility With Fixed Reversion

There are times when we need to calibrate with one parameter held fixed. QuantLib allows you to perform calibration with constraints. However, this ability is not exposed in the SWIG wrappers as of version 1.6. I have created a github issue[1] and provided a patch to address this issue. This patch has been merged into QuantLib-SWIG version 1.7. If you are using version lower than 1.7, you will need this patch to execute the following cells. Below, the model is calibrated with a fixed reversion value of 5%.

The following code is similar to the Hull-White calibration, except we initialize the constrained model with given values. In the calibrate method, we provide a list of boolean with constraints `[True, False]`, meaning that the first parameter `a` is constrained where as the second `sigma` is not constrained.

```
In [7]: constrained_model = HullWhite(term_structure, 0.05, 0.001);
        engine = JamshidianSwaptionEngine(constrained_model)
        swaptions = create_swaption_helpers(data, index, term_structure, engine)

        optimization_method = LevenbergMarquardt(1.0e-8,1.0e-8,1.0e-8)
        end_criteria = EndCriteria(10000, 100, 1e-6, 1e-8, 1e-8)
        constrained_model.calibrate(swaptions, optimization_method,
                                    end_criteria, NoConstraint(),
                                    [], [True, False])
        a, sigma = constrained_model.params()
        print("a = %6.5f, sigma = %6.5f" % (a, sigma))

Out[7]: a = 0.05000, sigma = 0.00586

In [8]: calibration_report(swaptions, data)

Out[8]: Cumulative Error Price: 0.11584
        Cumulative Error Vols : 0.11615

Out[8]:
```

[1] https://github.com/lballabio/quantlib-old/issues/336

Model Price	Market Price	Implied Vol	Market Vol	Rel Error Price	Rel Error Vols
0.008776	0.009485	0.106212	0.1148	-0.074738	-0.074812
0.009668	0.010078	0.106284	0.1108	-0.040682	-0.040761
0.008662	0.008716	0.106330	0.1070	-0.006261	-0.006261
0.006490	0.006226	0.106436	0.1021	0.042311	0.042469
0.003542	0.003323	0.106625	0.1000	0.065946	0.066252

Black Karasinski Model

The Black Karasinski model is described as:

$$d\ln(r_t) = (\theta_t - a\ln(r_t))dt + \sigma dW_t$$

Black-Karasinski is not an affine model, and hence we cannot use the `JamshidianSwaptionEngine`. In order to calibrate, we use the `TreeSwaptionEngine` which will work with all short rate models. The calibration procedure is shown below.

```
In [9]: model = BlackKarasinski(term_structure);
        engine = TreeSwaptionEngine(model, 100)
        swaptions = create_swaption_helpers(data, index, term_structure, engine)

        optimization_method = LevenbergMarquardt(1.0e-8,1.0e-8,1.0e-8)
        end_criteria = EndCriteria(10000, 100, 1e-6, 1e-8, 1e-8)
        model.calibrate(swaptions, optimization_method, end_criteria)

        a, sigma =  model.params()
        print("a = %6.5f, sigma = %6.5f" % (a, sigma))

Out[9]: a = 0.03902, sigma = 0.11695

In [10]: calibration_report(swaptions, data)

Out[10]: Cumulative Error Price: 0.12132
         Cumulative Error Vols : 0.12163

Out[10]:
```

Model Price	Market Price	Implied Vol	Market Vol	Rel Error Price	Rel Error Vols
0.008717	0.009485	0.105497	0.1148	-0.080954	-0.081033
0.009670	0.010078	0.106309	0.1108	-0.040453	-0.040531
0.008679	0.008716	0.106540	0.1070	-0.004297	-0.004297
0.006503	0.006226	0.106656	0.1021	0.044457	0.044623
0.003547	0.003323	0.106765	0.1000	0.067333	0.067646

G2++ Model

As a final example, let us look at a calibration example of the 2-factor G2++ model. $dr_t = \varphi(t) + x_t + y_t$ where x_t and y_t are defined by

$$dx_t = -ax_t dt + \sigma dW_t^1 \quad (17..1)$$

$$dy_t = -by_t dt + \eta dW_t^2 \quad (17..2)$$

$$\left\langle dW_t^1 dW_t^2 \right\rangle = \rho dt \quad (17..3)$$

Once again, we use the `TreeSwaptionEngine` to value the swaptions in the calibration step. One can also use `G2SwaptionEngine` and `FdG2SwaptionEngine`. But the calibration times, and accuracy can vary depending on the choice of parameters.

```
In [11]: model = G2(term_structure);
         engine = TreeSwaptionEngine(model, 25)
         # engine = ql.G2SwaptionEngine(model, 10, 400)
         # engine = ql.FdG2SwaptionEngine(model)
         swaptions = create_swaption_helpers(data, index, term_structure, engine)

         optimization_method = LevenbergMarquardt(1.0e-8,1.0e-8,1.0e-8)
         end_criteria = EndCriteria(1000, 100, 1e-6, 1e-8, 1e-8)
         model.calibrate(swaptions, optimization_method, end_criteria)

         a, sigma, b, eta, rho = model.params()
         print("a = %6.5f, sigma = %6.5f, b = %6.5f, eta = %6.5f, rho = %6.5f " % \
               (a, sigma, b, eta, rho))

Out[11]: a = 0.03942, sigma = 0.00473, b = 0.04720, eta = 0.00301, rho = 0.03865

In [12]: calibration_report(swaptions, data)

Out[12]: Cumulative Error Price: 0.12241
         Cumulative Error Vols : 0.12272

Out[12]:
```

Model Price	Market Price	Implied Vol	Market Vol	Rel Error Price	Rel Error Vols
0.008704	0.009485	0.105333	0.1148	-0.082383	-0.082464
0.009672	0.010078	0.106322	0.1108	-0.040334	-0.040412
0.008676	0.008716	0.106510	0.1070	-0.004583	-0.004583
0.006503	0.006226	0.106647	0.1021	0.044365	0.044531
0.003548	0.003323	0.106800	0.1000	0.067681	0.067996

Calibrating to Normal Volatilities

In certain markets in Europe and Japan for instance have had negative interest rates in the recent past for some of the tenors of the yield curve. The lognormal volatility quotes used above are inconsistent with negative rates and it is generally a practice to quote normal volaitilities in this case. The `SwaptionHelperPtr` used above with lognormal volatilities can be modified for normal volatilities by setting the `VolatilityType` parameter.

The full `C++` syntax for the `SwaptionHelper` object is as shown below:

```
SwaptionHelperPtr(const Date& exerciseDate, const Period& length,
                  const Handle<Quote>& volatility,
                  const IborIndexPtr& index,
                  const Period& fixedLegTenor,
                  const DayCounter& fixedLegDayCounter,
                  const DayCounter& floatingLegDayCounter,
                  const Handle<YieldTermStructure>& termStructure,
                  BlackCalibrationHelper::CalibrationErrorType errorType
                            = BlackCalibrationHelper::RelativePriceError,
                  const Real strike = Null<Real>(),
                  const Real nominal = 1.0,
                  const VolatilityType type = ShiftedLognormal,
                  const Real shift = 0.0)
```

In the above examples, we did not pass any of the optional arguments for `errorType`, `strike`, `nominal` and `type` specifying the `VolatilityType`. One can set the optional `type` parameter to change from log-normal volatilities to normal volatilities.

A function to create swaption helpers with `normal` volatilities is shown below:

```
In [13]: def create_swaption_helpers_normal(data, index, term_structure, engine):
             swaptions = []
             fixed_leg_tenor = Period(1, Years)
             fixed_leg_daycounter = Actual360()
             floating_leg_daycounter = Actual360()
             for d in data:
                 vol_handle = QuoteHandle(SimpleQuote(d.volatility))
                 helper= SwaptionHelper(Period(d.start, Years),
                                        Period(d.length, Years),
                                        vol_handle,
                                        index,
                                        fixed_leg_tenor,
                                        fixed_leg_daycounter,
                                        floating_leg_daycounter,
                                        term_structure,
                                        BlackCalibrationHelper.RelativePriceError,
                                        nullDouble(),
```

```
                                    1.0,
                                    Normal
                                    )
            helper.setPricingEngine(engine)
            swaptions.append(helper)
        return swaptions
```

Now, we can call the `create_swaption_helpers_normal` instead to constructions swaptions with normal volatilities and pass it to the calibration routine to determine the model parameters.

```
In [14]: model = HullWhite(term_structure);
         engine = JamshidianSwaptionEngine(model)
         swaptions = create_swaption_helpers_normal(data, index, term_structure, engine)

         optimization_method = LevenbergMarquardt(1.0e-8,1.0e-8,1.0e-8)
         end_criteria = EndCriteria(10000, 100, 1e-6, 1e-8, 1e-8)
         model.calibrate(swaptions, optimization_method, end_criteria)

         a, sigma = model.params()
         print("a = %6.5f, sigma = %6.5f" % (a, sigma))

Out[14]: a = 0.04595, sigma = 0.11868
```

Conclusion

In this chapter, we saw some simple examples of calibrating the interest rate models to the swaption volatilities. We looked at setting up different interest rate models and discussed both lognormal and normal volatilities.

18. Par versus indexed coupons

(Based on a question asked by KK[1] on the QuantLib mailing list. Thanks!)

```
In [1]: from QuantLib import *
        import pandas as pd

        today = Date(7,January,2013)
        Settings.instance().evaluationDate = today
```

The statement of the case

User KK was pricing an interest-rate swap. In the interest of brevity, I'll skip the part where he bootstrapped a LIBOR curve (there are other notebooks showing that in detail) and instantiate the resulting curve directly from the resulting forward rates.

```
In [2]: dates, forwards = zip(*[(Date(7,1,2013), 0.03613672438543303),
                                (Date(8,4,2013), 0.03613672438543303),
                                (Date(8,7,2013), 0.033849133719219514),
                                (Date(7,1,2014), 0.03573931373272106),
                                (Date(7,7,2014), 0.03445303757052511)])
        libor_curve = ForwardCurve(dates, forwards, Actual365Fixed())
```

Here is the floating leg of the swap; we don't need to care about the fixed leg.

```
In [3]: index = GBPLibor(Period(6,Months),
                        YieldTermStructureHandle(libor_curve))

        calendar = index.fixingCalendar()
        nominal = 1000000
        length = 1
        maturity = calendar.advance(today,length,Years)
        adjustment = index.businessDayConvention()

        schedule = Schedule(today, maturity,
                            index.tenor(), calendar,
                            adjustment, adjustment,
                            DateGeneration.Backward, False)

        floating_leg = IborLeg([nominal], schedule,
                               index, index.dayCounter())
```

[1] https://sourceforge.net/p/quantlib/mailman/message/32902476/

Next, KK set out to do some cash-flow analysis. He reproduced the coupon amounts by multiplying the LIBOR fixing, the notional, and the accrual period; the actual code was different, but the calculations are the same I'm doing here:

```
In [4]: df = pd.DataFrame()

        dates = list(schedule)
        df['fixing date'] = dates[:-1]
        df['index fixing'] = [ index.fixing(d) for d in df['fixing date'] ]
        df['start date'] = dates[:-1]
        df['end date'] = dates[1:]
        df['days'] = df['end date'] - df['start date']
        df['accrual period'] = df['days']/365

        df['amount'] = df['index fixing'] * nominal * df['accrual period']

        df
```

Out[4]:

	fixing date	index fixing	start date	end date	days	accrual period	amount
0	January 7th, 2013	0.035300	January 7th, 2013	July 8th, 2013	182	0.49863	17601.6
1	July 8th, 2013	0.036056	July 8th, 2013	January 7th, 2014	183	0.50137	18077.4

Unfortunately, the results for the second coupon don't agree with what the library says:

```
In [5]: df2 = pd.DataFrame({'amount': [ c.amount() for c in floating_leg ],
                            'rate': [ as_coupon(c).rate() for c in floating_leg ]})
        df2
```

Out[5]:

	amount	rate
0	17601.643836	0.035300
1	18080.116395	0.036061

The difference (in the rate, and thus the amount) is small but well above the expected precision for the calculations.

Where's the problem?

Let's go through the calculations again. The second coupon fixes on the expected date, and the forecast for the fixing is the same KK obtained.

```
In [6]: coupon = as_floating_rate_coupon(floating_leg[1])

        print(coupon.fixingDate())
        print(index.fixing(coupon.fixingDate()))

Out[6]: July 8th, 2013
        0.036056087457623655
```

The fixing is also consistent with what we can forecast from the LIBOR curve, given the start and end date of the underlying tenor:

```
In [7]: startDate = index.valueDate(coupon.fixingDate())
        endDate = index.maturityDate(startDate)
        print(startDate)
        print(endDate)

Out[7]: July 8th, 2013
        January 8th, 2014

In [8]: print(libor_curve.forwardRate(startDate, endDate,
                                      coupon.dayCounter(), Simple))

Out[8]: 3.605609 % Actual/365 (Fixed) simple compounding
```

The above is, in fact, the calculation performed in the `index.fixing` method.

Why does the coupon return a different rate, then?

The problem is that, for historical reasons, the coupon is calculated at par; that is, the floating rate is calculated over the duration of the coupon. Due to the constraints of the schedule, the end of the coupon doesn't correspond to the end of the LIBOR tenor...

```
In [9]: couponStart = coupon.accrualStartDate()
        couponEnd = coupon.accrualEndDate()
        print(couponStart)
        print(couponEnd)

Out[9]: July 8th, 2013
        January 7th, 2014
```

...and therefore, the calculated rate is different:

```
In [10]: print(libor_curve.forwardRate(couponStart, couponEnd,
                                       coupon.dayCounter(), Simple))

Out[10]: 3.606143 % Actual/365 (Fixed) simple compounding
```

The coupon amount is consistent with the rate above...

```
In [11]: coupon.rate()

Out[11]: 0.0360614343399347
```

...and so is the amount:

```
In [12]: coupon.rate() * nominal * coupon.accrualPeriod()

Out[12]: 18080.116395090554


In [13]: coupon.amount()

Out[13]: 18080.11639509055
```

Was it a good idea to use par coupons? Hard to say. They are used in textbook examples, which one might want to reproduce.

In any case, I've heard arguments against both calculations. The one against using the forecast index fixing goes that the rate would be accrued over a period which is different from the one over which it was calculated, and thus it will require a small convexity adjustment. Personally, the argument failed to persuade me, since with par coupons we're using the wrong rate over the right period and therefore we're introducing an error anyway; I doubt that it's smaller than the missing convexity adjustment. Moreover, the value of the swap is going to jump as soon as the coupon rate is actually fixed, forcing an adjustment of the P&L.

The good news is that you can choose which one to use: there's a configuration flag for the library that allows you to use the forecast of the fixing. The bad news is that this requires the recompilation of both the C++ library and the Python module. It would be better if the choice could be made at run-time; I describe the details in *Implementing QuantLib*[2], including a few glitches that this might cause. For the time being, you'll have to make do.

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

19. Modeling interest rate swaps using QuantLib

An interest rate swap is a financial derivative instrument in which two parties agree to exchange interest rate cash flows based on a notional amount from a fixed rate to a floating rate or from one floating rate to another floating rate.

Here we will consider an example of a plain vanilla USD swap with 10 million notional and 10 year maturity. Let the fixed leg pay 2.5% coupon semiannually, and the floating leg pay Libor 3m quarterly.

```
In [1]: from QuantLib import *
        calculation_date = Date(20, 10, 2015)
        Settings.instance().evaluationDate = calculation_date
```

Here we construct the yield curve objects. For simplicity, we will use flat curves for discounting and Libor 3M. This will help us focus on the swap construction. Please refer to curve construction example[1] for some details.

```
In [2]: # construct discount curve and libor curve

        risk_free_rate = 0.01
        libor_rate = 0.02
        day_count = Actual365Fixed()

        discount_curve = YieldTermStructureHandle(
            FlatForward(calculation_date, risk_free_rate, day_count)
        )

        libor_curve = YieldTermStructureHandle(
            FlatForward(calculation_date, libor_rate, day_count)
        )
        #libor3M_index = ql.Euribor3M(libor_curve)
        libor3M_index = USDLibor(Period(3, Months), libor_curve)
```

To construct the swap, we have to specify the fixed rate leg and floating rate leg. We construct the fixed rate and floating rate leg schedules below.

[1] http://gouthamanbalaraman.com/blog/quantlib-term-structure-bootstrap-yield-curve.html

```
In [3]: calendar = UnitedStates()
        settle_date = calendar.advance(calculation_date, 5, Days)
        maturity_date = calendar.advance(settle_date, 10, Years)

        fixed_leg_tenor = Period(6, Months)
        fixed_schedule = Schedule(settle_date, maturity_date,
                                  fixed_leg_tenor, calendar,
                                  ModifiedFollowing, ModifiedFollowing,
                                  DateGeneration.Forward, False)

        float_leg_tenor = Period(3, Months)
        float_schedule = Schedule(settle_date, maturity_date,
                                  float_leg_tenor, calendar,
                                  ModifiedFollowing, ModifiedFollowing,
                                  DateGeneration.Forward, False)
```

Below, we construct a `VanillaSwap` object by including the fixed and float leg schedules created above.

```
In [4]: notional = 10000000
        fixed_rate = 0.025
        fixed_leg_daycount = Actual360()
        float_spread = 0.004
        float_leg_daycount = Actual360()

        ir_swap = VanillaSwap(VanillaSwap.Payer, notional, fixed_schedule,
                              fixed_rate, fixed_leg_daycount, float_schedule,
                              libor3M_index, float_spread, float_leg_daycount )
```

We evaluate the swap using a discounting engine.

```
In [5]: swap_engine = DiscountingSwapEngine(discount_curve)
        ir_swap.setPricingEngine(swap_engine)
```

Result Analysis

The cash flows for the fixed and floating leg can be extracted from the `ir_swap` object. The fixed leg cash flows are shown below:

```
In [6]: from pandas import DataFrame
        DataFrame(
            [(cf.date(), cf.amount()) for cf in ir_swap.leg(0)],
            columns=["Date", "Amount"],
            index=range(1, len(ir_swap.leg(0))+1))
```

Out[6]:

	Date	Amount
1	April 27th, 2016	127083.333333
2	October 27th, 2016	127083.333333
3	April 27th, 2017	126388.888889
4	October 27th, 2017	127083.333333
5	April 27th, 2018	126388.888889
6	October 29th, 2018	128472.222222
7	April 29th, 2019	126388.888889
8	October 28th, 2019	126388.888889
9	April 27th, 2020	126388.888889
10	October 27th, 2020	127083.333333
11	April 27th, 2021	126388.888889
12	October 27th, 2021	127083.333333
13	April 27th, 2022	126388.888889
14	October 27th, 2022	127083.333333
15	April 27th, 2023	126388.888889
16	October 27th, 2023	127083.333333
17	April 29th, 2024	128472.222222
18	October 28th, 2024	126388.888889
19	April 28th, 2025	126388.888889
20	October 27th, 2025	126388.888889

The floating leg cash flows are shown below:

```
In [7]: from pandas import DataFrame
        DataFrame(
            [(cf.date(), cf.amount()) for cf in ir_swap.leg(1)],
            columns=["Date", "Amount"],
            index=range(1, len(ir_swap.leg(1))+1))
```

Out[7]:

	Date	Amount
1	January 27th, 2016	60760.458147
2	April 27th, 2016	60098.647700
3	July 27th, 2016	60098.647700
4	October 27th, 2016	60760.458147
5	January 27th, 2017	60760.458147
6	April 27th, 2017	59436.867427
7	July 27th, 2017	60098.647700
8	October 27th, 2017	60760.458147
9	January 29th, 2018	62084.169572
10	April 27th, 2018	58113.397399
11	July 27th, 2018	60098.647700
12	October 29th, 2018	62084.169572
13	January 28th, 2019	60098.647700
14	April 29th, 2019	60098.647700
15	July 29th, 2019	60098.647700
16	October 28th, 2019	60098.647700
17	January 27th, 2020	60098.647700
18	April 27th, 2020	60098.647700
19	July 27th, 2020	60098.647700
20	October 27th, 2020	60760.458147
21	January 27th, 2021	60760.458147
22	April 27th, 2021	59436.867427
23	July 27th, 2021	60098.647700
24	October 27th, 2021	60760.458147
25	January 27th, 2022	60760.458147
26	April 27th, 2022	59436.867427
27	July 27th, 2022	60098.647700
28	October 27th, 2022	60760.458147
29	January 27th, 2023	60760.458147
30	April 27th, 2023	59436.867427
31	July 27th, 2023	60098.647700
32	October 27th, 2023	60760.458147
33	January 29th, 2024	62084.169572
34	April 29th, 2024	60098.647700
35	July 29th, 2024	60098.647700
36	October 28th, 2024	60098.647700
37	January 27th, 2025	60098.647700
38	April 28th, 2025	60098.647700
39	July 28th, 2025	60098.647700
40	October 27th, 2025	60098.647700

Some other analytics such as the fair value, fair spread etc can be extracted as shown below.

```
In [8]: print("%-20s: %20.3f" % ("Net Present Value", ir_swap.NPV()))
        print("%-20s: %20.3f" % ("Fair Spread", ir_swap.fairSpread()))
        print("%-20s: %20.3f" % ("Fair Rate", ir_swap.fairRate()))
        print("%-20s: %20.3f" % ("Fixed Leg BPS", ir_swap.fixedLegBPS()))
        print("%-20s: %20.3f" % ("Floating Leg BPS", ir_swap.floatingLegBPS()))

Out[8]: Net Present Value   :          -115054.034
        Fair Spread         :                0.005
        Fair Rate           :                0.024
        Fixed Leg BPS       :            -9629.981
        Floating Leg BPS    :             9642.042
```

Conclusion

Here we saw a simple example on how to construct a swap and value them. We evaluated the fixed and floating legs and then valued the `VanillaSwap` using the `DiscountingSwapEngine`.

20. Caps and floors

In this post, I will walk you through a simple example of valuing caps. I want to talk about two specific cases:

1. Value caps given a constant volatility
2. Value caps given a cap volatility surface

Caps, as you might know, can be valued as a sum of caplets. The value of each caplet is determined by the Black formula. In practice, each caplet would have a different volatility. Meaning, a caplet that is in the near term can have a different volatility profile compared to the caplet that is far away in tenor. Similarly caplet volatilities differ with the strike as well.

```
In [1]: from QuantLib import *

In [2]: calc_date = Date(14, 6, 2016)
        Settings.instance().evaluationDate = calc_date
```

Constant Volatility

Let us start by constructing different components required in valuing the caps. The components that we would need are:

1. interest rate term structure for discounting
2. interest rate term structure for the floating leg
3. construction of the cap
4. the pricing engine to value caps using the Black formula

For simplicity, we will construct only one interest rate term structure here, and assume that the discounting and the floating leg is referenced by the same. Below the term structure of interest rates is constructed from a set of zero rates.

```
In [3]: dates = [Date(14,6,2016), Date(14,9,2016),
                 Date(14,12,2016), Date(14,6,2017),
                 Date(14,6,2019), Date(14,6,2021),
                 Date(15,6,2026), Date(16,6,2031),
                 Date(16,6,2036), Date(14,6,2046)
                 ]
        yields = [0.000000, 0.006616, 0.007049, 0.007795,
                  0.009599, 0.011203, 0.015068, 0.017583,
                  0.018998, 0.020080]

        day_count = ActualActual()
        calendar = UnitedStates()
        interpolation = Linear()
        compounding = Compounded
        compounding_frequency = Annual

        term_structure = ZeroCurve(dates, yields, day_count,
                                   calendar, interpolation,
                                   compounding, compounding_frequency)
        ts_handle = YieldTermStructureHandle(term_structure)
```

As a next step, lets construct the cap itself. In order to do that, we start by constructing the `Schedule` object to project the cash-flow dates.

```
In [4]: start_date = Date(14, 6, 2016)
        end_date = Date(14, 6 , 2026)
        period = Period(3, Months)
        calendar = UnitedStates()
        buss_convention = ModifiedFollowing
        rule = DateGeneration.Forward
        end_of_month = False

        schedule = Schedule(start_date, end_date, period,
                            calendar, buss_convention,
                            buss_convention, rule, end_of_month)
```

Now that we have the schedule, we construct the `USDLibor` index. Below, you can see that I use `addFixing` method to provide a fixing date for June 10, 2016. According the schedule constructed, the start date of the cap is June 14, 2016, and there is a 2 business day settlement lag (meaning June 10 reference date) embedded in the `USDLibor` definition. So in order to set the rate for the accrual period, the rate is obtained from the fixing data provided. For all future dates, the libor rates are automatically inferred using the forward rates provided by the given interest rate term structure.

```
In [5]: ibor_index = USDLibor(Period(3, Months), ts_handle)
        ibor_index.addFixing(Date(10,6,2016), 0.0065560)

        ibor_leg = IborLeg([1000000], schedule, ibor_index)
```

Now that we have all the required pieces, the `Cap` can be constructed by passing the `ibor_leg` and the `strike` information. Constructing a floor is done through the `Floor` class. The `BlackCapFloorEngine` can be used to price the cap with constant volatility as shown below.

```
In [6]: strike = 0.02
        cap = Cap(ibor_leg, [strike])

        vols = QuoteHandle(SimpleQuote(0.547295))
        engine = BlackCapFloorEngine(ts_handle, vols)

        cap.setPricingEngine(engine)
        print(cap.NPV())

Out[6]: 54369.85806286924
```

Using Volatility Surfaces

In the above exercise, we used a constant volatility value. In practice, one needs to strip the market quoted cap/floor volatilities to infer the volatility of each caplet. QuantLib provides excellent tools in order to do that. Let us assume the following dummy data represents the volatility surface quoted by the market. I have the various `strikes`, `expiries`, and the volatility quotes in percentage format. I take the raw data and create a `Matrix` in order to construct the volatility surface.

```
In [7]: strikes = [0.01,0.015, 0.02]
        temp = list(range(1, 11)) + [12]
        expiries = [Period(i, Years) for i in temp]
        vols = Matrix(len(expiries), len(strikes))
        data = [[47.27, 55.47, 64.07, 70.14, 72.13, 69.41, 72.15, 67.28, 66.08, 68.64, 65.83],
           [46.65,54.15,61.47,65.53,66.28,62.83,64.42,60.05,58.71,60.35,55.91],
           [46.6,52.65,59.32,62.05,62.0,58.09,59.03,55.0,53.59,54.74,49.54]
           ]

        for i in range(vols.rows()):
            for j in range(vols.columns()):
                vols[i][j] = data[j][i]/100.0
```

The `CapFloorTermVolSurface` offers a way to store the cap/floor volatilities. These are however `CapFloor` volatilities, and not the volatilities of the individual options.

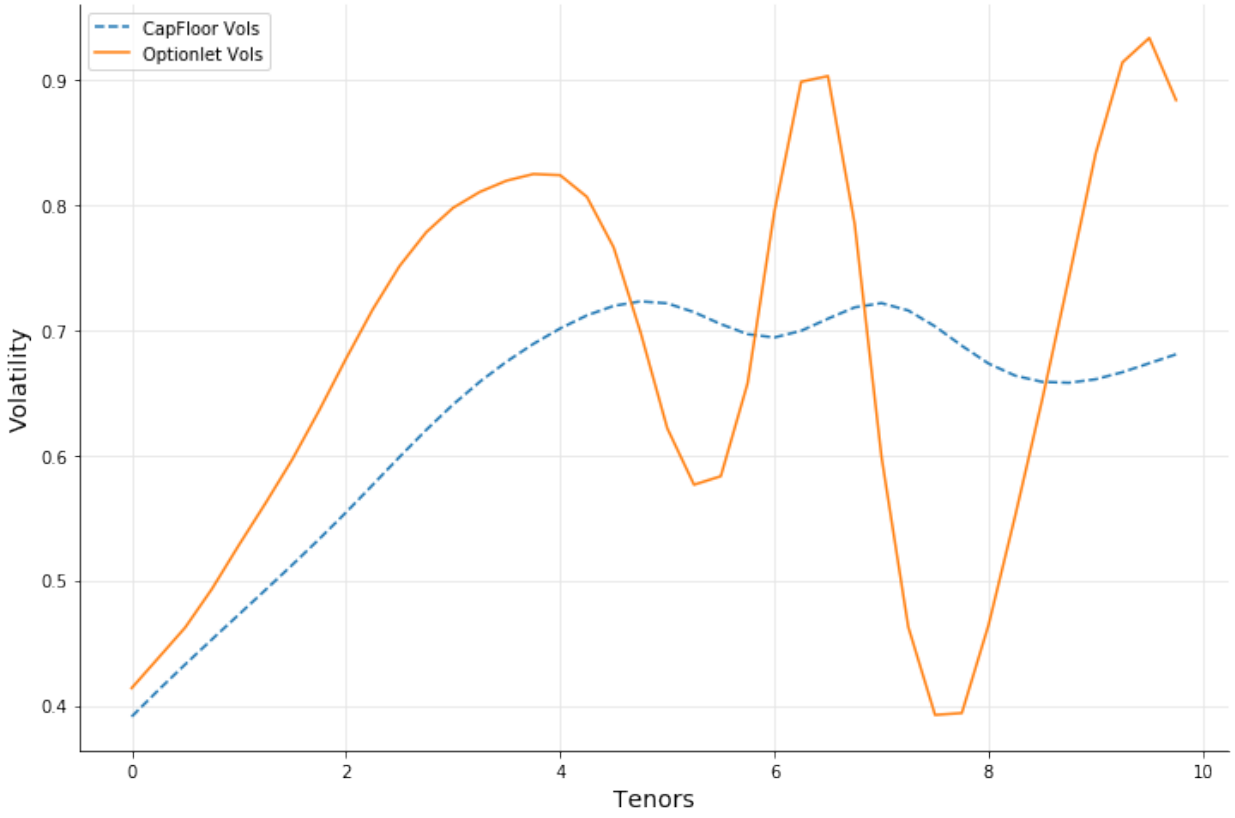
```
In [8]: calendar = UnitedStates()
        bdc = ModifiedFollowing
        daycount = Actual365Fixed()
        settlement_days = 2
        capfloor_vol = CapFloorTermVolSurface(
            settlement_days, calendar, bdc,
            expiries, strikes, vols, daycount
        )
```

The `OptionletStripper1` class lets you to strip the individual caplet/floorlet volatilities from the cap/floor volatilities. We have to 'jump' some hoops here to make it useful for pricing. The `OptionletStripper1` class cannot be consumed directly by a pricing engine. The `StrippedOptionletAdapter` takes the stripped optionlet volatilities, and creates a term structure of optionlet volatilities. We then wrap that into a handle using `OptionletVolatilityStructureHandle`.

```
In [9]: optionlet_surf = OptionletStripper1(capfloor_vol, ibor_index, nullDouble(),
                                            1e-6, 100, ts_handle)
        ovs_handle = OptionletVolatilityStructureHandle(
            StrippedOptionletAdapter(optionlet_surf)
        )
```

Below, we visualize the cap/floor volatility surface, and the optionlet volatility surface for a fixed strike.

```
In [10]: import utils
         import numpy as np
         %matplotlib inline

In [11]: tenors = np.arange(0,10,0.25)
         strike = 0.01
         capfloor_vols = [capfloor_vol.volatility(t, strike) for t in tenors]
         opionlet_vols = [ovs_handle.volatility(t, strike) for t in tenors]

         fig, ax = utils.plot()
         ax.plot(tenors, capfloor_vols, "--", label="CapFloor Vols")
         ax.plot(tenors, opionlet_vols,"-", label="Optionlet Vols")
         ax.legend(loc='best')
         ax.set_xlabel("Tenors", size=14)
         ax.set_ylabel("Volatility", size=14);
```



The `BlackCapFloorEngine` can accept the optionlet volatility surface in order to price the caps or floors.

```
In [12]: engine2 = BlackCapFloorEngine(ts_handle, ovs_handle)
         cap.setPricingEngine(engine2)
         print(cap.NPV())

Out[12]: 54384.928314950135
```

One can infer the implied volatility for the cap at its NPV, and it should be in agreement with what is quote by the surface.

```
In [13]: cap.impliedVolatility(cap.NPV(), ts_handle, 0.4)

Out[13]: 0.5474438930928851
```

The QuantLib C++ class allow for one to view the projected cash flows in terms of individual caplets.

Equity models

21. Valuing European option using the Heston model

Heston model can be used to value options by modeling the underlying asset such as the stock of a company. The one major feature of the Heston model is that it inocrporates a stochastic volatility term.

$$dS_t = \mu S_t dt + \sqrt{V_t} S_t dW_t^1 \quad (21..1)$$
$$dV_t = \kappa(\theta - V_t) + \sigma\sqrt{V_t} dW_t^2 \quad (21..2)$$

Here :

- S_t is the asset's value at time t
- μ is the expected growth rate of the log normal stock value
- V_t is the variance of the asset S_t
- W_t^1 is the stochastic process governing the S_t process
- θ is the value of mean reversion for the variance V_t
- κ is the strength of mean reversion
- σ is the volatility of volatility
- W_t^2 is the stochastic process governing the V_t process
- The correlation between W_t^1 and W_t^2 is ρ

In contrast, the Black-Scholes-Merton process assumes that the volatility is constant.

```
In [1]: from QuantLib import *
        import numpy as np
        import math
```

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%. We assume a short term risk free rate of 0.1%. Lets value this option as of 8th May, 2015.

```
In [2]: # option parameters
        strike_price = 110.0
        payoff = PlainVanillaPayoff(Option.Call, strike_price)

        # option data
        maturity_date = 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 = Option.Call

        risk_free_rate = 0.001
        day_count = Actual365Fixed()
        calendar = UnitedStates()

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

Using the above inputs, we construct the European option as shown below.

```
In [3]: # construct the European Option
        payoff = PlainVanillaPayoff(option_type, strike_price)
        exercise = EuropeanExercise(maturity_date)
        european_option = VanillaOption(payoff, exercise)
```

In order to price the option using the Heston model, we first create the Heston process. In order to create the Heston process, we use the parameter values: mean reversion strength `kappa = 0.1`, the spot variance `v0 = volatility*volatility = 0.04`, the mean reversion variance `theta=v0`, volatility of volatility `sigma = 0.1` and the correlation between the asset price and its variance is `rho = -0.75`.

```
In [4]: # construct the Heston process

        v0 = volatility*volatility  # spot variance
        kappa = 0.1
        theta = v0
        sigma = 0.1
        rho = -0.75

        spot_handle = QuoteHandle(
            SimpleQuote(spot_price)
        )
        flat_ts = YieldTermStructureHandle(
            FlatForward(calculation_date, risk_free_rate, day_count)
        )
```

```
        dividend_yield = YieldTermStructureHandle(
            FlatForward(calculation_date, dividend_rate, day_count)
        )
        heston_process = HestonProcess(flat_ts, dividend_yield,
                                       spot_handle, v0, kappa,
                                       theta, sigma, rho)
```

On valuing the option using the Heston model, we get the net present value as:

```
In [5]: engine = AnalyticHestonEngine(HestonModel(heston_process),0.01, 1000)
        european_option.setPricingEngine(engine)
        h_price = european_option.NPV()
        print("The Heston model price is {0}".format(h_price))

Out[5]: The Heston model price is 6.533855481449102
```

Performing the same calculation using the Black-Scholes-Merton process, we get:

```
In [6]: flat_vol_ts = BlackVolTermStructureHandle(
            BlackConstantVol(calculation_date, calendar,
                             volatility, day_count)
        )
        bsm_process = BlackScholesMertonProcess(spot_handle, dividend_yield,
                                                flat_ts, flat_vol_ts)
        european_option.setPricingEngine(AnalyticEuropeanEngine(bsm_process))
        bs_price = european_option.NPV()
        print("The Black-Scholes-Merton model price is {0}".format(bs_price))

Out[6]: The Black-Scholes-Merton model price is 6.749271812460607
```

The difference in the price between the two models is `bs_price - h_price`, or about 0.215. This difference is due to the stochastic modeling of the volatility as a CIR-process.

22. Volatility smile and Heston model calibration

European options on an equity underlying such as an index (S&P 500) or a stock (AMZN) trade for different combinations of strikes and maturities. It turns out that the Black-Scholes implied volatility for these options with different maturities and strikes is not the same. The fact that the implied volatility varies with strike is often referred in the market as having a *smile*.

```
In [1]: from QuantLib import *
        import math
```

First let us define some of the basic data conventions such as the day_count, calendar etc.

```
In [2]: day_count = Actual365Fixed()
        calendar = UnitedStates()

        calculation_date = Date(6, 11, 2015)

        spot = 659.37
        Settings.instance().evaluationDate = calculation_date

        dividend_yield = QuoteHandle(SimpleQuote(0.0))
        risk_free_rate = 0.01
        dividend_rate = 0.0
        flat_ts = YieldTermStructureHandle(
            FlatForward(calculation_date, risk_free_rate, day_count))
        dividend_ts = YieldTermStructureHandle(
            FlatForward(calculation_date, dividend_rate, day_count))
```

Following is a sample matrix of volatility quote by expiry and strike. The volatilities are log-normal volatilities and can be interpolated to construct the implied volatility surface.

```
In [3]: expiration_dates = [Date(6,12,2015), Date(6,1,2016), Date(6,2,2016),
                            Date(6,3,2016), Date(6,4,2016), Date(6,5,2016),
                            Date(6,6,2016), Date(6,7,2016), Date(6,8,2016),
                            Date(6,9,2016), Date(6,10,2016), Date(6,11,2016),
                            Date(6,12,2016), Date(6,1,2017), Date(6,2,2017),
                            Date(6,3,2017), Date(6,4,2017), Date(6,5,2017),
                            Date(6,6,2017), Date(6,7,2017), Date(6,8,2017),
                            Date(6,9,2017), Date(6,10,2017), Date(6,11,2017)]
        strikes = [527.50, 560.46, 593.43, 626.40, 659.37, 692.34, 725.31, 758.28]
        data = [
        [0.37819, 0.34177, 0.30394, 0.27832, 0.26453, 0.25916, 0.25941, 0.26127],
        [0.3445, 0.31769, 0.2933, 0.27614, 0.26575, 0.25729, 0.25228, 0.25202],
        [0.37419, 0.35372, 0.33729, 0.32492, 0.31601, 0.30883, 0.30036, 0.29568],
        [0.37498, 0.35847, 0.34475, 0.33399, 0.32715, 0.31943, 0.31098, 0.30506],
        [0.35941, 0.34516, 0.33296, 0.32275, 0.31867, 0.30969, 0.30239, 0.29631],
        [0.35521, 0.34242, 0.33154, 0.3219, 0.31948, 0.31096, 0.30424, 0.2984],
        [0.35442, 0.34267, 0.33288, 0.32374, 0.32245, 0.31474, 0.30838, 0.30283],
        [0.35384, 0.34286, 0.33386, 0.32507, 0.3246, 0.31745, 0.31135, 0.306],
        [0.35338, 0.343, 0.33464, 0.32614, 0.3263, 0.31961, 0.31371, 0.30852],
        [0.35301, 0.34312, 0.33526, 0.32698, 0.32766, 0.32132, 0.31558, 0.31052],
        [0.35272, 0.34322, 0.33574, 0.32765, 0.32873, 0.32267, 0.31705, 0.31209],
        [0.35246, 0.3433, 0.33617, 0.32822, 0.32965, 0.32383, 0.31831, 0.31344],
        [0.35226, 0.34336, 0.33651, 0.32869, 0.3304, 0.32477, 0.31934, 0.31453],
        [0.35207, 0.34342, 0.33681, 0.32911, 0.33106, 0.32561, 0.32025, 0.3155],
        [0.35171, 0.34327, 0.33679, 0.32931, 0.3319, 0.32665, 0.32139, 0.31675],
        [0.35128, 0.343, 0.33658, 0.32937, 0.33276, 0.32769, 0.32255, 0.31802],
        [0.35086, 0.34274, 0.33637, 0.32943, 0.3336, 0.32872, 0.32368, 0.31927],
        [0.35049, 0.34252, 0.33618, 0.32948, 0.33432, 0.32959, 0.32465, 0.32034],
        [0.35016, 0.34231, 0.33602, 0.32953, 0.33498, 0.3304, 0.32554, 0.32132],
        [0.34986, 0.34213, 0.33587, 0.32957, 0.33556, 0.3311, 0.32631, 0.32217],
        [0.34959, 0.34196, 0.33573, 0.32961, 0.3361, 0.33176, 0.32704, 0.32296],
        [0.34934, 0.34181, 0.33561, 0.32964, 0.33658, 0.33235, 0.32769, 0.32368],
        [0.34912, 0.34167, 0.3355, 0.32967, 0.33701, 0.33288, 0.32827, 0.32432],
        [0.34891, 0.34154, 0.33539, 0.3297, 0.33742, 0.33337, 0.32881, 0.32492]]
```

Implied Volatility Surface

Each row in `data` is a different expiration time, and each column corresponds to various strikes as given in `strikes`. We load all this data into the QuantLib `Matrix` object. This can then be used seamlessly in the various surface construction routines. The variable `implied_vols` holds the above data in a `Matrix` format. One unusual bit of info that one needs to pay attention to is the ordering of the rows and columns in the `Matrix` object. The implied volatilities in the QuantLib context needs to have strikes along the row dimension and expiries in the column dimension. This is transpose of the way the data was constructed above. All of this detail is taken care by swapping the `i` and `j` variables below. Pay attention to the line:

```
implied_vols[i][j] = data[j][i]
```

in the cell below.

```
In [4]: implied_vols = Matrix(len(strikes), len(expiration_dates))
        for i in range(implied_vols.rows()):
            for j in range(implied_vols.columns()):
                implied_vols[i][j] = data[j][i]
```

Now the Black volatility surface can be constructed using the `BlackVarianceSurface` method.

```
In [5]: black_var_surface = BlackVarianceSurface(
            calculation_date, calendar,
            expiration_dates, strikes,
            implied_vols, day_count)
```

The volatilities for any given strike and expiry pair can be easily obtained using `black_var_surface` shown below.

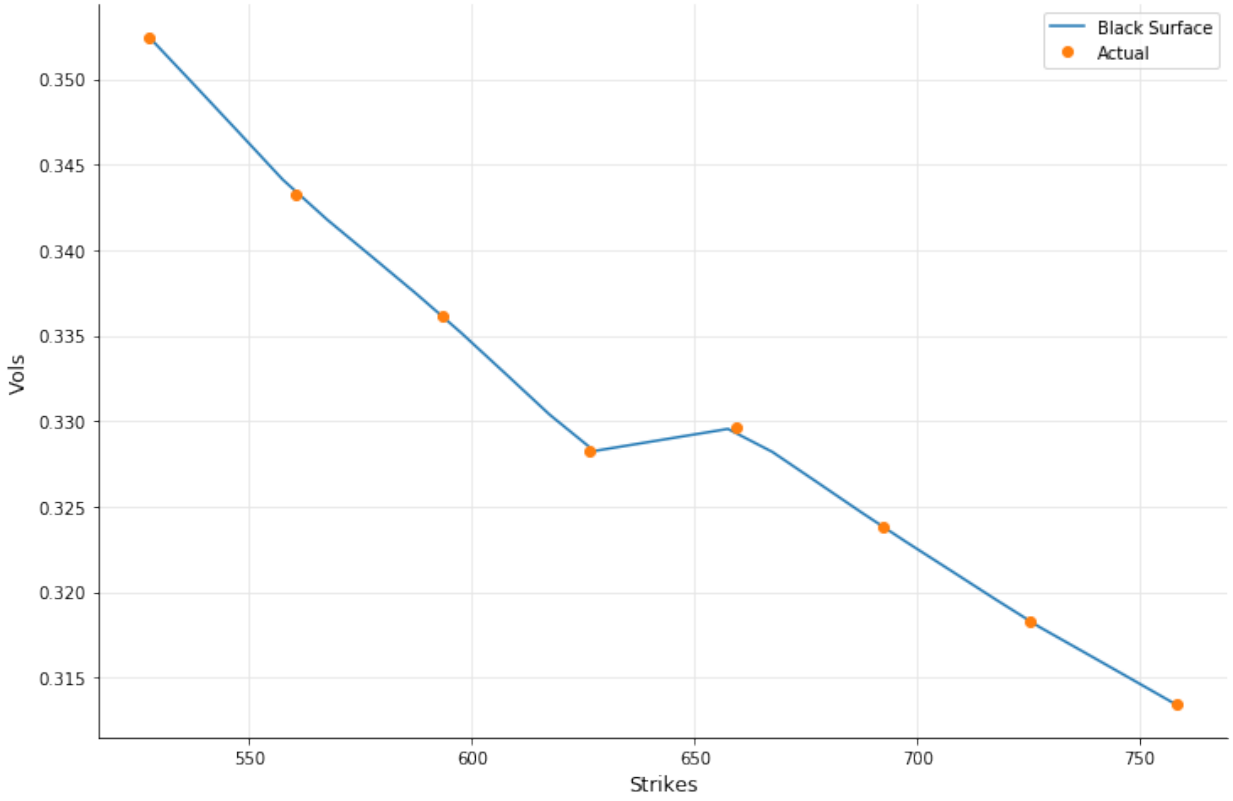
```
In [6]: strike = 600.0
        expiry = 1.2 # years
        black_var_surface.blackVol(expiry, strike)

Out[6]: 0.3352982638587421
```

Visualization

```
In [7]: import numpy as np
        import utils
        %matplotlib inline
        from mpl_toolkits.mplot3d import Axes3D
        import matplotlib.pyplot as plt
        from matplotlib import cm
```

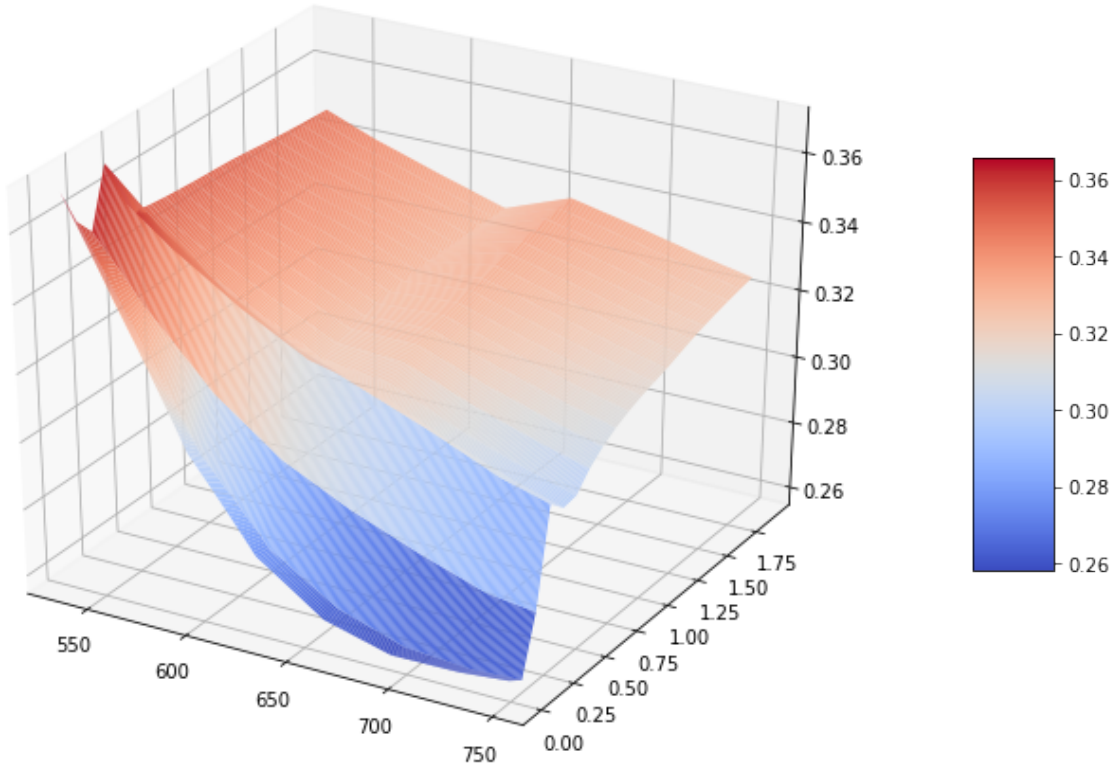
Given an expiry, we can visualize the volatility as a function of the strike.

```
In [8]: strikes_grid = np.arange(strikes[0], strikes[-1],10)
        expiry = 1.0 # years
        implied_vols = [black_var_surface.blackVol(expiry, s)
                        for s in strikes_grid] # can interpolate here
        actual_data = data[11] # cherry picked the data for given expiry

        fig, ax = utils.plot()
        ax.plot(strikes_grid, implied_vols, label="Black Surface")
        ax.plot(strikes, actual_data, "o", label="Actual")
        ax.set_xlabel("Strikes", size=12)
        ax.set_ylabel("Vols", size=12)
        ax.legend(loc="upper right");
```



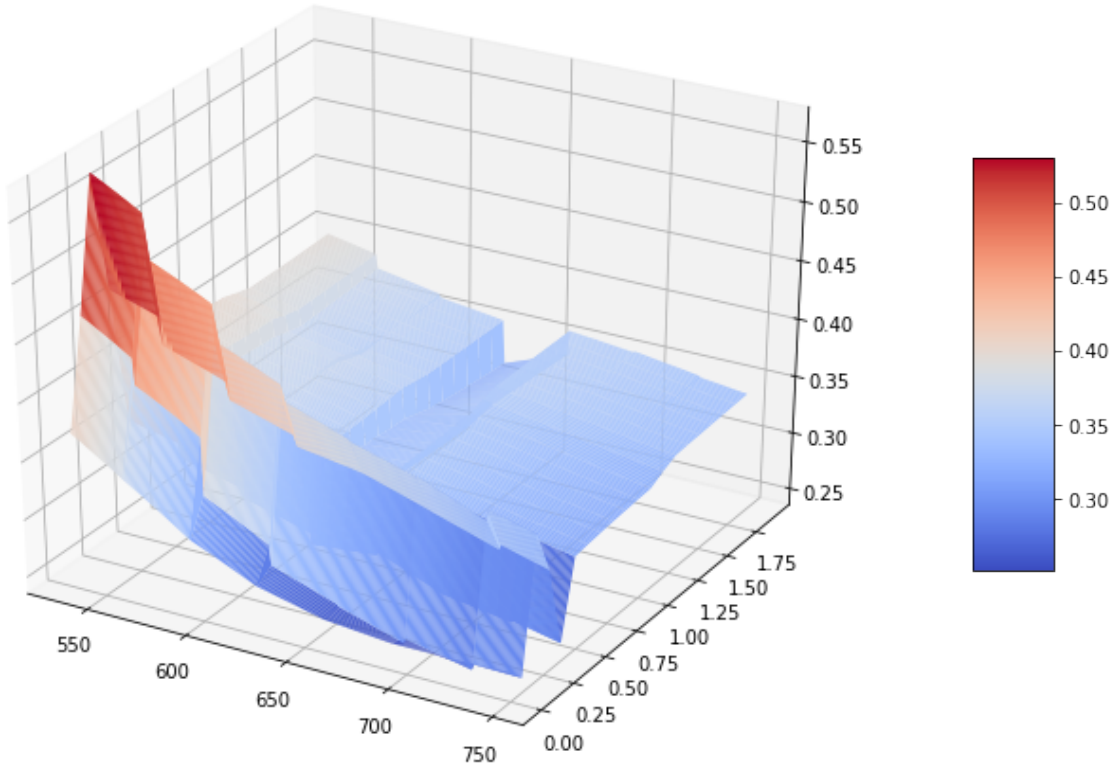
The whole volatility surface can also be visualised as shown below.

```
In [9]: plot_years = np.arange(0, 2, 0.1)
        plot_strikes = np.arange(535, 750, 1)
        fig = plt.figure(figsize=utils.default_plot_size)
        ax = fig.gca(projection='3d')
        X, Y = np.meshgrid(plot_strikes, plot_years)
        Z = np.array([black_var_surface.blackVol(float(y), float(x))
                      for xr, yr in zip(X, Y)
                          for x, y in zip(xr,yr) ]
                     ).reshape(len(X), len(X[0]))

        surf = ax.plot_surface(X,Y,Z, rstride=1, cstride=1, cmap=cm.coolwarm,
                               linewidth=0.1)
        fig.colorbar(surf, shrink=0.5, aspect=5);
```



One can also construct a local volatility surface (*a la* Dupire) using the `LocalVolSurface`. There are some issues with this as shown below.

```
In [10]: local_vol_surface = LocalVolSurface(
             BlackVolTermStructureHandle(black_var_surface),
             flat_ts,
             dividend_ts,
             spot)

In [11]: plot_years = np.arange(0, 2, 0.1)
         plot_strikes = np.arange(535, 750, 1)
         fig = plt.figure(figsize=utils.default_plot_size)
         ax = fig.gca(projection='3d')
         X, Y = np.meshgrid(plot_strikes, plot_years)
         Z = np.array([local_vol_surface.localVol(float(y), float(x))
                       for xr, yr in zip(X, Y)
                           for x, y in zip(xr,yr) ]
                      ).reshape(len(X), len(X[0]))

         surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=cm.coolwarm,
                                linewidth=0.1)
         fig.colorbar(surf, shrink=0.5, aspect=5);
```



The correct pricing of local volatility surface requires an arbitrage free implied volatility surface. If the input implied volatility surface is not arbitrage free, this can lead to negative transition probabilities and/or negative local volatilities and can give rise to mispricing. Refer to Fengler's arbitrage-free smoothing [1] which QuantLib currently lacks.

When you use an arbitrary smoothing, you will notice that the local volatility surface leads to undesired negative volatilities. This can lead to errors as shown below.

```
In [12]: black_var_surface.setInterpolation("bicubic")
         local_vol_surface = LocalVolSurface(
             BlackVolTermStructureHandle(black_var_surface),
             flat_ts,
             dividend_ts,
             spot)
         plot_years = np.arange(0, 2, 0.15)
         plot_strikes = np.arange(535, 750, 10)
         X, Y = np.meshgrid(plot_strikes, plot_years)
         Z = np.array([local_vol_surface.localVol(float(y), float(x))
                       for xr, yr in zip(X, Y)
                           for x, y in zip(xr,yr) ]
                      ).reshape(len(X), len(X[0]))
```

```
Out[12]: ---------------------------------------------------------------------------
         RuntimeError                              Traceback (most recent call last)
         <ipython-input-12-938d20361f8f> in <module>()
               9 X, Y = np.meshgrid(plot_strikes, plot_years)
              10 Z = np.array([local_vol_surface.localVol(float(y), float(x))
         ---> 11                  for xr, yr in zip(X, Y)
              12                      for x, y in zip(xr,yr) ]
              13                  ).reshape(len(X), len(X[0]))

         <ipython-input-12-938d20361f8f> in <listcomp>(.0)
              10 Z = np.array([local_vol_surface.localVol(float(y), float(x))
              11                  for xr, yr in zip(X, Y)
         ---> 12                      for x, y in zip(xr,yr) ]
              13                  ).reshape(len(X), len(X[0]))

         /usr/local/lib/python3.6/dist-packages/QuantLib/QuantLib.py in localVol(self, *args)
            8255
            8256     def localVol(self, *args):
         -> 8257         return _QuantLib.LocalVolTermStructure_localVol(self, *args)
            8258
            8259     def enableExtrapolation(self):

         RuntimeError: negative local vol^2 at strike 655 and time 0.75; the black vol surface\
 is not smooth enough
```

Heston Model Calibration

Heston model is defined by the following stochastic differential equations.

$$
\begin{aligned}
dS(t,S) &= \mu S dt + \sqrt{v} S dW_1 && (22..1)\\
dv(t,S) &= \kappa(\theta - v)dt + \sigma\sqrt{v}dW_2 && (22..2)\\
dW_1 dW_2 &= \rho dt && (22..3)
\end{aligned}
$$

Here the asset is modeled as a stochastic process that depends on volatility v which is a mean reverting stochastic process with a constant volatility of volatility σ. The two stochastic processes have a correlation ρ.

Let us look at how we can calibrate the Heston model to some market quotes. As an example, let's say we are interested in trading options with 1 year maturity. So we will calibrate the Heston model to fit to market volatility quotes with one year maturity. Before we do that, we need to construct the pricing engine that the calibration routines would need. In order to do that, we start by constructing the Heston model with some dummy starting parameters as shown below.

```
In [13]: # dummy parameters
         v0 = 0.01; kappa = 0.2; theta = 0.02; rho = -0.75; sigma = 0.5;

         process = HestonProcess(flat_ts, dividend_ts,
                                 QuoteHandle(SimpleQuote(spot)),
                                 v0, kappa, theta, sigma, rho)
         model = HestonModel(process)
         engine = AnalyticHestonEngine(model)
```

Now that we have the Heston model and a pricing engine, let us pick the quotes with all strikes and 1 year maturity in order to calibrate the Heston model. We build the Heston model helper which will be fed into the calibration routines.

```
In [14]: heston_helpers = []
         black_var_surface.setInterpolation("bicubic")
         one_year_idx = 11 # 12th row in data is for 1 year expiry
         date = expiration_dates[one_year_idx]
         for j, s in enumerate(strikes):
             t = (date - calculation_date )
             p = Period(t, Days)
             sigma = data[one_year_idx][j]
             #sigma = black_var_surface.blackVol(t/365.25, s)
             helper = HestonModelHelper(p, calendar, spot, s,
                                        QuoteHandle(SimpleQuote(sigma)),
                                        flat_ts, dividend_ts)
             helper.setPricingEngine(engine)
             heston_helpers.append(helper)

In [15]: lm = LevenbergMarquardt(1e-8, 1e-8, 1e-8)
         model.calibrate(heston_helpers, lm,
                         EndCriteria(500, 50, 1.0e-8,1.0e-8, 1.0e-8))
         theta, kappa, sigma, rho, v0 = model.params()

In [16]: print("theta = %f, kappa = %f, sigma = %f, rho = %f, v0 = %f" \
             % (theta, kappa, sigma, rho, v0))

Out[16]: theta = 0.132329, kappa = 10.980901, sigma = 4.018093, rho = -0.351560, v0 = 0.065672
```

Let us look at the quality of calibration by pricing the options used in the calibration using the model and lets get an estimate of the relative error.

```
In [17]: avg = 0.0

         summary = []
         for i, opt in enumerate(heston_helpers):
             err = (opt.modelValue()/opt.marketValue() - 1.0)
             summary.append((
                 strikes[i], opt.marketValue(),
                 opt.modelValue(),
                 100.0*(opt.modelValue()/opt.marketValue() - 1.0)))
             avg += abs(err)
         avg = avg*100.0/len(heston_helpers)

In [18]: import pandas as pd

         print("Average Abs Error (%%) : %5.3f" % (avg))

         pd.DataFrame(
             summary,
             columns=["Strikes", "Market value", "Model value", "Relative error (%)"],
             index=['']*len(summary))

Out[18]: Average Abs Error (%) : 0.633

Out[18]:
```

Strikes	Market value	Model value	Relative error (%)
527.50	44.678931	44.465565	-0.477555
560.46	55.052769	55.232883	0.327167
593.43	67.371524	67.665921	0.436975
626.40	80.934108	81.828303	1.104843
659.37	98.889639	97.751713	-1.150703
692.34	93.297706	92.551981	-0.799296
725.31	79.649513	79.448640	-0.252195
758.28	67.621458	67.972305	0.518840

References

[1] Mathias R. Fengler, *Arbitrage Free Smoothing of Implied Volatility Surface*[1]

[1] https://core.ac.uk/download/files/153/6978470.pdf

23. Heston model parameter calibration in QuantLib Python & SciPy

I have discussed parameter calibration using the QuantLib solvers in the earlier chapter. In this chapter I want to show how you can use QuantLib Python and the Python SciPy library to do parameter calibration. This exercise opens us to use more robust solvers available in the Python open source libraries. QuantLib's strength is all financial models. SciPy's strength is all the solvers and numerical methods. So here, I will show you how you can make the best of both worlds. We will start as usual by importing the modules.

```
In [1]: from QuantLib import *
        from math import pow, sqrt
        import numpy as np
        from scipy.optimize import root
```

Let's construct some of the basic dependencies such as the yield and dividend term structures.

```
In [2]: day_count = Actual365Fixed()
        calendar = UnitedStates()
        calculation_date = Date(6, 11, 2015)

        spot = 659.37
        Settings.instance().evaluationDate = calculation_date

        risk_free_rate = 0.01
        dividend_rate = 0.0
        yield_ts = YieldTermStructureHandle(
            FlatForward(calculation_date, risk_free_rate, day_count))
        dividend_ts = YieldTermStructureHandle(
            FlatForward(calculation_date, dividend_rate, day_count))
```

Following is a sample grid of volatilities for different expiration and strikes.

```
In [3]: expiration_dates = [Date(6,12,2015), Date(6,1,2016), Date(6,2,2016),
                            Date(6,3,2016), Date(6,4,2016), Date(6,5,2016),
                            Date(6,6,2016), Date(6,7,2016), Date(6,8,2016),
                            Date(6,9,2016), Date(6,10,2016), Date(6,11,2016),
                            Date(6,12,2016), Date(6,1,2017), Date(6,2,2017),
                            Date(6,3,2017), Date(6,4,2017), Date(6,5,2017),
                            Date(6,6,2017), Date(6,7,2017), Date(6,8,2017),
                            Date(6,9,2017), Date(6,10,2017), Date(6,11,2017)]
        strikes = [527.50, 560.46, 593.43, 626.40, 659.37, 692.34, 725.31, 758.28]
        data = [
        [0.37819, 0.34177, 0.30394, 0.27832, 0.26453, 0.25916, 0.25941, 0.26127],
        [0.3445, 0.31769, 0.2933, 0.27614, 0.26575, 0.25729, 0.25228, 0.25202],
        [0.37419, 0.35372, 0.33729, 0.32492, 0.31601, 0.30883, 0.30036, 0.29568],
        [0.37498, 0.35847, 0.34475, 0.33399, 0.32715, 0.31943, 0.31098, 0.30506],
        [0.35941, 0.34516, 0.33296, 0.32275, 0.31867, 0.30969, 0.30239, 0.29631],
        [0.35521, 0.34242, 0.33154, 0.3219, 0.31948, 0.31096, 0.30424, 0.2984],
        [0.35442, 0.34267, 0.33288, 0.32374, 0.32245, 0.31474, 0.30838, 0.30283],
        [0.35384, 0.34286, 0.33386, 0.32507, 0.3246, 0.31745, 0.31135, 0.306],
        [0.35338, 0.343, 0.33464, 0.32614, 0.3263, 0.31961, 0.31371, 0.30852],
        [0.35301, 0.34312, 0.33526, 0.32698, 0.32766, 0.32132, 0.31558, 0.31052],
        [0.35272, 0.34322, 0.33574, 0.32765, 0.32873, 0.32267, 0.31705, 0.31209],
        [0.35246, 0.3433, 0.33617, 0.32822, 0.32965, 0.32383, 0.31831, 0.31344],
        [0.35226, 0.34336, 0.33651, 0.32869, 0.3304, 0.32477, 0.31934, 0.31453],
        [0.35207, 0.34342, 0.33681, 0.32911, 0.33106, 0.32561, 0.32025, 0.3155],
        [0.35171, 0.34327, 0.33679, 0.32931, 0.3319, 0.32665, 0.32139, 0.31675],
        [0.35128, 0.343, 0.33658, 0.32937, 0.33276, 0.32769, 0.32255, 0.31802],
        [0.35086, 0.34274, 0.33637, 0.32943, 0.3336, 0.32872, 0.32368, 0.31927],
        [0.35049, 0.34252, 0.33618, 0.32948, 0.33432, 0.32959, 0.32465, 0.32034],
        [0.35016, 0.34231, 0.33602, 0.32953, 0.33498, 0.3304, 0.32554, 0.32132],
        [0.34986, 0.34213, 0.33587, 0.32957, 0.33556, 0.3311, 0.32631, 0.32217],
        [0.34959, 0.34196, 0.33573, 0.32961, 0.3361, 0.33176, 0.32704, 0.32296],
        [0.34934, 0.34181, 0.33561, 0.32964, 0.33658, 0.33235, 0.32769, 0.32368],
        [0.34912, 0.34167, 0.3355, 0.32967, 0.33701, 0.33288, 0.32827, 0.32432],
        [0.34891, 0.34154, 0.33539, 0.3297, 0.33742, 0.33337, 0.32881, 0.32492]]
```

I have abstracted some of the repetitive methods into python functions. The function `setup_helpers` will construct the Heston model helpers and returns an array of these objects. The `cost_function_generator` is a method to set the cost function and will be used by the SciPy modules. The `calibration_report` lets us evaluate the quality of the fit. The `setup_model` method initializes the `HestonModel` and the `AnalyticHestonEngine` prior to calibration.

```
In [4]: def setup_helpers(engine, expiration_dates, strikes,
                          data, ref_date, spot, yield_ts,
                          dividend_ts):
            heston_helpers = []
            grid_data = []
            for i, date in enumerate(expiration_dates):
                for j, s in enumerate(strikes):
                    t = (date - ref_date )
                    p = Period(t, Days)
                    vols = data[i][j]
                    helper = HestonModelHelper(
                        p, calendar, spot, s,
                        QuoteHandle(SimpleQuote(vols)),
                        yield_ts, dividend_ts)
                    helper.setPricingEngine(engine)
                    heston_helpers.append(helper)
                    grid_data.append((date, s))
            return heston_helpers, grid_data

        def cost_function_generator(model, helpers,norm=False):
            def cost_function(params):
                params_ = Array(list(params))
                model.setParams(params_)
                error = [h.calibrationError() for h in helpers]
                if norm:
                    return np.sqrt(np.sum(np.abs(error)))
                else:
                    return error
            return cost_function

        def calibration_report(helpers, grid_data, detailed=False):
            avg = 0.0
            if detailed:
                print("%15s %25s %15s %15s %20s" % (
                    "Strikes", "Expiry", "Market Value",
                    "Model Value", "Relative Error (%)"))
                print("="*100)
            for i, opt in enumerate(helpers):
                err = (opt.modelValue()/opt.marketValue() - 1.0)
                date,strike = grid_data[i]
                if detailed:
                    print("%15.2f %25s %14.5f %15.5f %20.7f " % (
                        strike, str(date), opt.marketValue(),
                        opt.modelValue(),
                        100.0*(opt.modelValue()/opt.marketValue() - 1.0)))
                avg += abs(err)
            avg = avg*100.0/len(helpers)
```

```
            if detailed: print("-"*100)
            summary = "Average Abs Error (%%) : %5.9f" % (avg)
            print(summary)
            return avg

        def setup_model(_yield_ts, _dividend_ts, _spot,
                        init_condition=(0.02,0.2,0.5,0.1,0.01)):
            theta, kappa, sigma, rho, v0 = init_condition
            process = HestonProcess(_yield_ts, _dividend_ts,
                                    QuoteHandle(SimpleQuote(_spot)),
                                    v0, kappa, theta, sigma, rho)
            model = HestonModel(process)
            engine = AnalyticHestonEngine(model)
            return model, engine
        summary= []
```

Comparing Different Calibration Methods

Solvers such as Levenberg-Marquardt find local minima and are very sensitive to the initial conditions. Depending on the starting conditions for your solver, you could end up with a good set of parameters with good convergence or not so good set of parameters. We will look at two initial conditions for different solvers and see how the local minima solvers perform. We will compare this with differential evolution that looks for global minima.

We will setup the Heston model with two different initial conditions: - theta, kappa, sigma, rho, v0 = (0.02, 0.2, 0.5, 0.1, 0.01) - theta, kappa, sigma, rho, v0 = (0.07, 0.5, 0.1, 0.1, 0.1)

Local Solvers

Using QuantLib Levenberg-Marquardt Solver

As a first step, let's look at QuantLib's Levenberg-Marquardt solver. The initial condition considered is `theta, kappa, sigma, rho, v0 = (0.02,0.2,0.5,0.1,0.01)`

```
In [5]: model1, engine1 = setup_model(
            yield_ts, dividend_ts, spot,
            init_condition=(0.02,0.2,0.5,0.1,0.01))
        heston_helpers1, grid_data1 = setup_helpers(
            engine1, expiration_dates, strikes, data,
            calculation_date, spot, yield_ts, dividend_ts
        )
        initial_condition = list(model1.params())

In [6]: %%time
        lm = LevenbergMarquardt(1e-8, 1e-8, 1e-8)
```

```
        model1.calibrate(heston_helpers1, lm,
                         EndCriteria(500, 300, 1.0e-8,1.0e-8, 1.0e-8))
        theta, kappa, sigma, rho, v0 = model1.params()
        print("theta = %f, kappa = %f, sigma = %f, rho = %f, v0 = %f" % \
            (theta, kappa, sigma, rho, v0))
        error = calibration_report(heston_helpers1, grid_data1)
        summary.append(["QL LM1", error] + list(model1.params()))

Out[6]: theta = 0.125748, kappa = 7.915000, sigma = 1.887854, rho = -0.364942, v0 = 0.055397
        Average Abs Error (%) : 3.015268051
        CPU times: user 2.8 s, sys: 0 ns, total: 2.8 s
        Wall time: 2.8 s
```

Methods like Levenberg-Marquardt solve for local minima and do not search for global minima. The solver is very sensitive to the initial conditions. Let's set a different set of initial conditions, and see what happens below. The initial condition considered is `theta, kappa, sigma, rho, v0 = (0.07,0.5,0.1,0.1,0.1)`

```
In [7]: model1, engine1 = setup_model(
            yield_ts, dividend_ts, spot,
            init_condition=(0.07,0.5,0.1,0.1,0.1))
        heston_helpers1, grid_data1 = setup_helpers(
            engine1, expiration_dates, strikes, data,
            calculation_date, spot, yield_ts, dividend_ts
        )
        initial_condition = list(model1.params())

In [8]: %%time
        lm = LevenbergMarquardt(1e-8, 1e-8, 1e-8)
        model1.calibrate(heston_helpers1, lm,
                         EndCriteria(500, 300, 1.0e-8,1.0e-8, 1.0e-8))
        theta, kappa, sigma, rho, v0 = model1.params()
        print("theta = %f, kappa = %f, sigma = %f, rho = %f, v0 = %f" % \
            (theta, kappa, sigma, rho, v0))
        error = calibration_report(heston_helpers1, grid_data1)
        summary.append(["QL LM2", error] + list(model1.params()))

Out[8]: theta = 0.084523, kappa = 0.000000, sigma = 0.132289, rho = -0.514278, v0 = 0.099928
        Average Abs Error (%) : 11.007433024
        CPU times: user 2.91 s, sys: 0 ns, total: 2.91 s
        Wall time: 2.89 s
```

We see that the solver produces a 11% average of absolute error. This is not particularly great.

Using SciPy Levenberg-Marquardt Solver

Here we are going to try the same exercise but using SciPy. SciPy has far more optimization, minimization and root finding algorithms that are very robust. So by leveraging it, we can take

advantage of this rich set of options at hand.

```
In [9]: model2, engine2 = setup_model(
            yield_ts, dividend_ts, spot,
            init_condition=(0.02,0.2,0.5,0.1,0.01))
        heston_helpers2, grid_data2 = setup_helpers(
            engine2, expiration_dates, strikes, data,
            calculation_date, spot, yield_ts, dividend_ts
        )
        initial_condition = list(model2.params())

In [10]: %%time
         cost_function = cost_function_generator(model2, heston_helpers2)
         sol = root(cost_function, initial_condition, method='lm')
         theta, kappa, sigma, rho, v0 = model2.params()
         print("theta = %f, kappa = %f, sigma = %f, rho = %f, v0 = %f" % \
             (theta, kappa, sigma, rho, v0))
         error = calibration_report(heston_helpers2, grid_data2)
         summary.append(["SciPy LM1", error] + list(model2.params()))

Out[10]: theta = 0.125747, kappa = 7.915687, sigma = 1.887934, rho = -0.364944, v0 = 0.055394
         Average Abs Error (%) : 3.015252651
         CPU times: user 3.24 s, sys: 20 ms, total: 3.26 s
         Wall time: 3.54 s
```

The solution for this particular case seems to be fairly robust. Both solvers (QuantLib and SciPy) seem to have landed on more or less the same solution for this particular initial condition. Let's see how SciPy does for the second initial condition considered above - `theta, kappa, sigma, rho, v0 = (0.07,0.5,0.1,0.1,0.1)`

```
In [11]: model2, engine2 = setup_model(
             yield_ts, dividend_ts, spot,
             init_condition=(0.07,0.5,0.1,0.1,0.1))
         heston_helpers2, grid_data2 = setup_helpers(
             engine2, expiration_dates, strikes, data,
             calculation_date, spot, yield_ts, dividend_ts
         )
         initial_condition = list(model2.params())

In [12]: %%time
         cost_function = cost_function_generator(model2, heston_helpers2)
         sol = root(cost_function, initial_condition, method='lm')
         theta, kappa, sigma, rho, v0 = model2.params()
         print("theta = %f, kappa = %f, sigma = %f, rho = %f, v0 = %f" % \
             (theta, kappa, sigma, rho, v0))
         error = calibration_report(heston_helpers2, grid_data2)
```

```
          summary.append(["SciPy LM2", error] + list(model2.params()))

Out[12]: theta = 0.048184, kappa = -0.548903, sigma = 0.197958, rho = -0.999547, v0 = 0.090571
          Average Abs Error (%) : 7.019499509
          CPU times: user 20.8 s, sys: 20 ms, total: 20.8 s
          Wall time: 21.4 s
```

For this particular case, SciPy solver has performed significantly better. It would be inappropriate to make loud claims about SciPy's superiority based on one observation. Perhaps this calls for a more detailed study for later.

Using Least Squares Method

If you want to use a simpler approach like least squares, you can do that with SciPy. Here is how you would use it.

```
In [13]: from scipy.optimize import least_squares

In [14]: model3, engine3 = setup_model(
             yield_ts, dividend_ts, spot,
             init_condition=(0.02,0.2,0.5,0.1,0.01))
         heston_helpers3, grid_data3 = setup_helpers(
             engine3, expiration_dates, strikes, data,
             calculation_date, spot, yield_ts, dividend_ts
         )
         initial_condition = list(model3.params())

In [15]: %%time
         cost_function = cost_function_generator(model3, heston_helpers3)
         sol = least_squares(cost_function, initial_condition)
         theta, kappa, sigma, rho, v0 = model3.params()
         print("theta = %f, kappa = %f, sigma = %f, rho = %f, v0 = %f" % \
             (theta, kappa, sigma, rho, v0))
         error = calibration_report(heston_helpers3, grid_data3)
         summary.append(["SciPy LS1", error] + list(model3.params()))

Out[15]: theta = 0.125747, kappa = 7.915814, sigma = 1.887949, rho = -0.364944, v0 = 0.055394
          Average Abs Error (%) : 3.015251175
          CPU times: user 4.59 s, sys: 0 ns, total: 4.59 s
          Wall time: 4.63 s
```

With the second initial condition:

```
In [16]: model3, engine3 = setup_model(
             yield_ts, dividend_ts, spot,
             init_condition=(0.07,0.5,0.1,0.1,0.1))
         heston_helpers3, grid_data3 = setup_helpers(
             engine3, expiration_dates, strikes, data,
             calculation_date, spot, yield_ts, dividend_ts
         )
         initial_condition = list(model3.params())

In [17]: %%time
         cost_function = cost_function_generator(model3, heston_helpers3)
         sol = least_squares(cost_function, initial_condition)
         theta, kappa, sigma, rho, v0 = model3.params()
         print("theta = %f, kappa = %f, sigma = %f, rho = %f, v0 = %f" % \
             (theta, kappa, sigma, rho, v0))
         error = calibration_report(heston_helpers3, grid_data3)
         summary.append(["SciPy LS2", error] + list(model3.params()))

Out[17]: theta = 3.136774, kappa = 0.000005, sigma = -0.000245, rho = -0.000010, v0 = 1.597904
         Average Abs Error (%) : 5.096414042
         CPU times: user 27.9 s, sys: 80 ms, total: 28 s
         Wall time: 29.7 s
```

Global Solvers

Using Differential Evolution

The above methods are more suited to finding local minima. One method that makes an attempt at searching for global minima is the differential evolution. Both QuantLib and SciPy have implementations of this method. SciPy however has a lot more bells and whistles to tune and calibrate the methodology. Let's take a look at SciPy's `differential_evolution` methodology.

```
In [18]: from scipy.optimize import differential_evolution

In [19]: model4, engine4 = setup_model(yield_ts, dividend_ts, spot)
         heston_helpers4, grid_data4 = setup_helpers(
             engine4, expiration_dates, strikes, data,
             calculation_date, spot, yield_ts, dividend_ts
         )
         initial_condition = list(model4.params())
         bounds = [(0,1),(0.01,15), (0.01,1.), (-1,1), (0,1.0) ]

In [20]: %%time
         cost_function = cost_function_generator(
             model4, heston_helpers4, norm=True)
         sol = differential_evolution(cost_function, bounds, maxiter=100)
```

```
        theta, kappa, sigma, rho, v0 = model4.params()
        print("theta = %f, kappa = %f, sigma = %f, rho = %f, v0 = %f" % \
            (theta, kappa, sigma, rho, v0))
        error = calibration_report(heston_helpers4, grid_data4)
        summary.append(["SciPy DE1", error] + list(model4.params()))
```

```
Out[20]: theta = 0.123607, kappa = 4.718829, sigma = 0.897901, rho = -0.595593, v0 = 0.079324
         Average Abs Error (%) : 2.864749340
         CPU times: user 1min 28s, sys: 370 ms, total: 1min 28s
         Wall time: 1min 30s
```

In the above example, I am setting the variable `maxiter` in order to limit the time taken. In production scenarios, you may want to try a larger number or not provide any value and default to 1000. This can help search a larger area of the parameter space.

```
In [21]: model4, engine4 = setup_model(yield_ts, dividend_ts, spot)
         heston_helpers4, grid_data4 = setup_helpers(
             engine4, expiration_dates, strikes, data,
             calculation_date, spot, yield_ts, dividend_ts
         )
         initial_condition = list(model4.params())
         bounds = [(0,1),(0.01,15), (0.01,1.), (-1,1), (0,1.0) ]
```

```
In [22]: %%time
         cost_function = cost_function_generator(
             model4, heston_helpers4, norm=True)
         sol = differential_evolution(cost_function, bounds, maxiter=100)
         theta, kappa, sigma, rho, v0 = model4.params()
         print("theta = %f, kappa = %f, sigma = %f, rho = %f, v0 = %f" % \
             (theta, kappa, sigma, rho, v0))
         error = calibration_report(heston_helpers4, grid_data4)
         summary.append(["SciPy DE2", error] + list(model4.params()))
```

```
Out[22]: theta = 0.121953, kappa = 4.963114, sigma = 0.813166, rho = -0.650423, v0 = 0.079030
         Average Abs Error (%) : 2.882908440
         CPU times: user 1min 37s, sys: 380 ms, total: 1min 38s
         Wall time: 1min 42s
```

Basin Hopping Algorithm

Here we will use the Basin Hopping (annealing-like) method to solve for the parameters. A couple things to make note here. The Basin Hopping method works best when used with a minimizer. Here I played with various minimizers and finally decided to use something that supports bounds checking. Without bounds checking, I often ended with `nan` and did not have a meaningful solution in the end.

I have chosen bounds based on a very basic reasoning. One needs careful reasoning to use appropriate bounds for the problem at hand.

```
In [23]: from scipy.optimize import basinhopping

In [24]: class MyBounds(object):
             def __init__(self, xmin=[0.,0.01,0.01,-1,0], xmax=[1,15,1,1,1.0] ):
                 self.xmax = np.array(xmax)
                 self.xmin = np.array(xmin)
             def __call__(self, **kwargs):
                 x = kwargs["x_new"]
                 tmax = bool(np.all(x <= self.xmax))
                 tmin = bool(np.all(x >= self.xmin))
                 return tmax and tmin
         bounds = [(0,1),(0.01,15), (0.01,1.), (-1,1), (0,1.0) ]

In [25]: model5, engine5 = setup_model(
             yield_ts, dividend_ts, spot,
             init_condition=(0.02,0.2,0.5,0.1,0.01))
         heston_helpers5, grid_data5 = setup_helpers(
             engine5, expiration_dates, strikes, data,
             calculation_date, spot, yield_ts, dividend_ts
         )
         initial_condition = list(model5.params())

In [26]: %%time
         mybound = MyBounds()
         minimizer_kwargs = {"method": "L-BFGS-B", "bounds": bounds }
         cost_function = cost_function_generator(
             model5, heston_helpers5, norm=True)
         sol = basinhopping(cost_function, initial_condition, niter=5,
                            minimizer_kwargs=minimizer_kwargs,
                            stepsize=0.005,
                            accept_test=mybound,
                            interval=10)
         theta, kappa, sigma, rho, v0 = model5.params()
         print("theta = %f, kappa = %f, sigma = %f, rho = %f, v0 = %f" % \
             (theta, kappa, sigma, rho, v0))
         error = calibration_report(heston_helpers5, grid_data5)
         summary.append(["SciPy BH1", error] + list(model5.params()))

Out[26]: theta = 0.123462, kappa = 5.069027, sigma = 0.998750, rho = -0.564407, v0 = 0.079309
         Average Abs Error (%) : 2.850721046
         CPU times: user 1min 54s, sys: 170 ms, total: 1min 54s
         Wall time: 1min 54s

In [27]: model5, engine5 = setup_model(
```

```
            yield_ts, dividend_ts, spot,
            init_condition=(0.07,0.5,0.1,0.1,0.1))
        heston_helpers5, grid_data5 = setup_helpers(
            engine5, expiration_dates, strikes, data,
            calculation_date, spot, yield_ts, dividend_ts
        )
        initial_condition = list(model5.params())
```

```
In [28]: %%time
         mybound = MyBounds()
         minimizer_kwargs = {"method": "L-BFGS-B", "bounds": bounds}
         cost_function = cost_function_generator(
             model5, heston_helpers5, norm=True)
         sol = basinhopping(cost_function, initial_condition, niter=5,
                            minimizer_kwargs=minimizer_kwargs,
                            stepsize=0.005,
                            accept_test=mybound,
                            interval=10)
         theta, kappa, sigma, rho, v0 = model5.params()
         print("theta = %f, kappa = %f, sigma = %f, rho = %f, v0 = %f" % \
             (theta, kappa, sigma, rho, v0))
         error = calibration_report(heston_helpers5, grid_data5)
         summary.append(["SciPy BH2", error] + list(model5.params()))
```

```
Out[28]: theta = 0.123144, kappa = 5.171826, sigma = 0.999454, rho = -0.565149, v0 = 0.079094
         Average Abs Error (%) : 2.850452127
         CPU times: user 1min 43s, sys: 70 ms, total: 1min 44s
         Wall time: 1min 44s
```

Summary

Here is a summary of all the results with the calibration error overall, and the respective parameters. All the local minima methods give parameters that are very different based on the initial condition that we start with. This is different in contrary with the global minimization methods that all end up in more or less the same proximity of each other.

The global solvers such as Differential Evolution and Basin Hopping are capable of finding the global minima and it is sometimes a question of computation resources. Here, I have lower "iterations" set for these routines for faster solving. Even with such a short threshold, we get fairly good solution set. I think it is premature to compare the effectiveness of different global solvers just based on the results here. The scipy optimize[1] package has detailed documentation with various tuning parameters. I haven't exploited the nuances much, and is left as an exercise for the reader.

[1] http://docs.scipy.org/doc/scipy/reference/optimize.html

```
In [29]: from pandas import DataFrame
         DataFrame(
             summary,
             columns=["Name", "Avg Abs Error","Theta", "Kappa", "Sigma", "Rho", "V0"],
             index=['']*len(summary))
```

Out[29]:

Name	Avg Abs Error	Theta	Kappa	Sigma	Rho	V0
QL LM1	3.015268	0.125748	7.915000e+00	1.887854	-0.364942	0.055397
QL LM2	11.007433	0.084523	1.625740e-08	0.132289	-0.514278	0.099928
SciPy LM1	3.015253	0.125747	7.915687e+00	1.887934	-0.364944	0.055394
SciPy LM2	7.019500	0.048184	-5.489029e-01	0.197958	-0.999547	0.090571
SciPy LS1	3.015251	0.125747	7.915814e+00	1.887949	-0.364944	0.055394
SciPy LS2	5.096414	3.136774	4.896844e-06	-0.000245	-0.000010	1.597904
SciPy DE1	2.864749	0.123607	4.718829e+00	0.897901	-0.595593	0.079324
SciPy DE2	2.882908	0.121953	4.963114e+00	0.813166	-0.650423	0.079030
SciPy BH1	2.850721	0.123462	5.069027e+00	0.998750	-0.564407	0.079309
SciPy BH2	2.850452	0.123144	5.171826e+00	0.999454	-0.565149	0.079094

24. 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]: from QuantLib import *
        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 = 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 = Option.Call

        risk_free_rate = 0.001
        day_count = Actual365Fixed()
        calendar = UnitedStates()

        calculation_date = Date(8, 5, 2015)
        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 = PlainVanillaPayoff(option_type, strike_price)
        exercise = EuropeanExercise(maturity_date)
        european_option = VanillaOption(payoff, exercise)
```

The Black-Scholes-Merton process is constructed here.

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

Lets compute the theoretical price using the `AnalyticEuropeanEngine`.

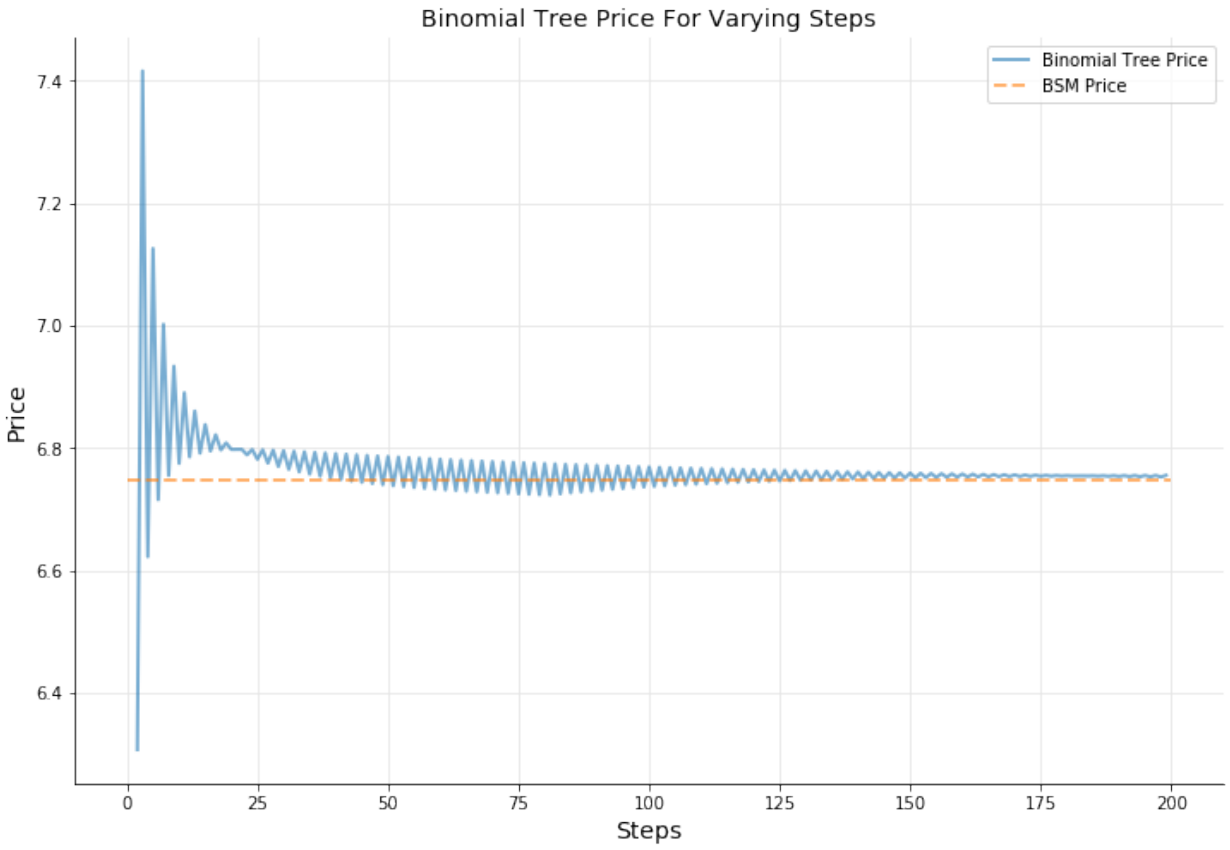
```
In [5]: european_option.setPricingEngine(AnalyticEuropeanEngine(bsm_process))
        bs_price = european_option.NPV()
        print("The theoretical price is %lf" % bs_price)

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 = 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 steps]
```

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

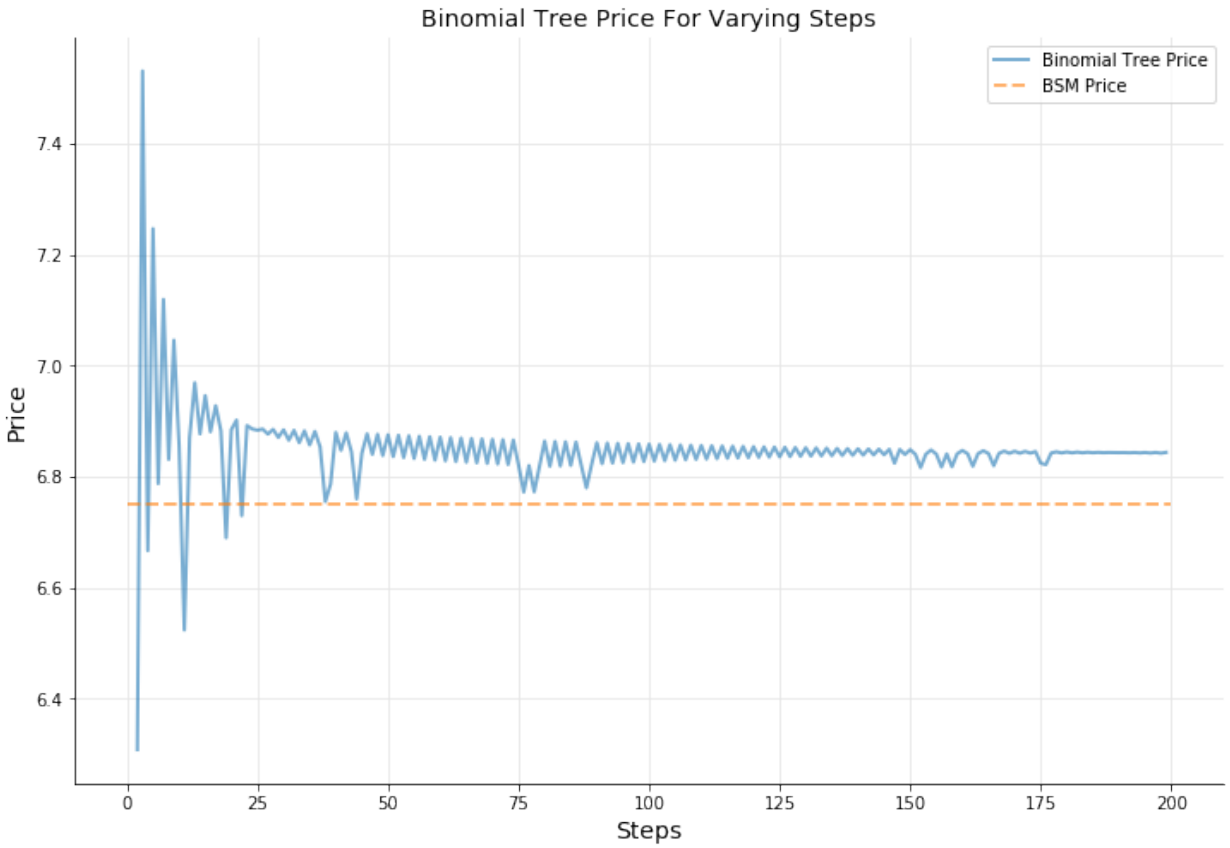
The above exercise was pedagogical, and introduces one to pricing using the binomial tree approach and compared with Black-Scholes. As a next step, we will use the Binomial pricing to value American options.

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 = PlainVanillaPayoff(option_type, strike_price)
        settlement = calculation_date
        am_exercise = AmericanExercise(settlement, maturity_date)
        american_option = 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 steps]

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.

25. Valuing options on commodity futures using the Black formula

The Black-Scholes equation is the well known model to price equity European options. In the case of equities, the spot price fluctuates and hence the intuition to model as a stochastic process makes sense. In the case of commodities, however, the spot price does not fluctuate as much, and hence cannot be modeled as a stochastic process to value options on commodities. In the 1976 paper [1], Fischer Black addressed this problem by modeling the futures price, which demonstrates fluctuations, as the lognormal stochastic process.

The futures price at time t, F_t with a is modeled as:

$$dF_t = \sigma_t F_t dW$$

where σ_t is the volatility of the underlying, and dW is the Weiner process.

The value of an option at strike K, maturing at T, risk free rate r with volatility σ of the underlying is given by the closed form expression:

$$\begin{aligned} c &= e^{-rT}[FN(d_1) - KN(d_2)] \quad (25..1) \\ p &= e^{-rT}[KN(-d_2) - FN(-d_1)] \quad (25..2) \end{aligned}$$

where

$$\begin{aligned} d_1 &= \frac{\ln(F/K) + (\sigma^2/2)T}{\sigma\sqrt{T}} \\ d_2 &= \frac{\ln(F/K) - (\sigma^2/2)T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T} \quad (25..3) \end{aligned}$$

This formula can be easily used to price caps, swaptions, futures options contract. Here we will use QuantLib to price the options on commodity futures.

```
In [1]: from QuantLib import *
        import math

In [2]: calendar = UnitedStates()
        business_convention = ModifiedFollowing
        settlement_days = 0
        day_count = ActualActual()
```

Option on Treasury Futures Contract

Options on treasury futures (10 Yr Note TYF6C 119) can be valued using the Black formula. Let us value a Call option maturing on December 24, 2015, with a strike of 119. The current futures price is 126.95, and the volatility is 11.567%. The risk free rate as of December 1, 2015 is 0.105%. Let us value this Call option as of December 1, 2015.

```
In [3]: interest_rate = 0.00105
        calc_date = Date(1,12,2015)
        yield_curve = FlatForward(calc_date,
                                  interest_rate,
                                  day_count,
                                  Compounded,
                                  Continuous)

In [4]:
        Settings.instance().evaluationDate = calc_date
        option_maturity_date = Date(24,12,2015)
        strike = 119
        spot = 126.953# futures price
        volatility = 11.567/100.
        flavor = Option.Call

        discount = yield_curve.discount(option_maturity_date)
        strikepayoff = PlainVanillaPayoff(flavor, strike)
        T = yield_curve.dayCounter().yearFraction(calc_date,
                                                  option_maturity_date)
        stddev = volatility*math.sqrt(T)

        black = BlackCalculator(strikepayoff,
                                spot,
                                stddev,
                                discount)

In [5]: print("%-20s: %4.4f" %("Option Price", black.value()))
        print("%-20s: %4.4f" %("Delta", black.delta(spot)))
        print("%-20s: %4.4f" %("Gamma", black.gamma(spot)))
        print("%-20s: %4.4f" %("Theta", black.theta(spot, T)))
```

```
        print("%-20s: %4.4f" %("Vega", black.vega(T)))
        print("%-20s: %4.4f" %("Rho", black.rho(T)))
```

```
Out[5]: Option Price        : 7.9686
        Delta               : 0.9875
        Gamma               : 0.0088
        Theta               : -0.9356
        Vega                : 1.0285
        Rho                 : 7.3974
```

Natural Gas Futures Option

I saw this[1] question on the QuantLib users group. Thought I will add this example as well.

Call option with a 3.5 strike, spot 2.919, volatility 0.4251. The interest rate is 0.15%.

```
In [6]: interest_rate = 0.0015
        calc_date = Date(23,9,2015)
        yield_curve = FlatForward(calc_date,
                                  interest_rate,
                                  day_count,
                                  Compounded,
                                  Continuous)
```

```
In [7]: Settings.instance().evaluationDate = calc_date
        T = 96.12/365.

        strike = 3.5
        spot = 2.919
        volatility = 0.4251
        flavor = Option.Call

        discount = yield_curve.discount(T)
        strikepayoff = PlainVanillaPayoff(flavor, strike)
        stddev = volatility*math.sqrt(T)

        strikepayoff = PlainVanillaPayoff(flavor, strike)
        black = BlackCalculator(strikepayoff, spot, stddev, discount)
```

```
In [8]: print("%-20s: %4.4f" %("Option Price", black.value()))
        print("%-20s: %4.4f" %("Delta", black.delta(spot)))
        print("%-20s: %4.4f" %("Gamma", black.gamma(spot)))
        print("%-20s: %4.4f" %("Theta", black.theta(spot, T)))
        print("%-20s: %4.4f" %("Vega", black.vega(T)))
        print("%-20s: %4.4f" %("Rho", black.rho(T)))
```

[1] http://quantlib.10058.n7.nabble.com/Quantlib-methods-for-option-pricing-td17018.html

```
Out[8]: Option Price        : 0.0789
        Delta               : 0.2347
        Gamma               : 0.4822
        Theta               : -0.3711
        Vega                : 0.4600
        Rho                 : 0.1597
```

Conclusion

In this notebook, I demonstrated how Black formula can be used to value options on commodity futures. It is worth pointing out that different vendors usually have different scaling conventions when it comes to reporting Greeks. One would needs to take that into account when comparing the values shown by QuantLib with that of other vendors.

References

[1] Fischer Black, *The pricing of commodity contracts*, Journal of Financial Economics, (3) 167-179 (1976)

26. Defining rho for the Black process

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

```
In [1]: from QuantLib import *
```

```
In [2]: today = Date(24,12,2016)
        Settings.instance().evaluationDate = today
```

The dangers of generalization

QuantLib provides a few classes to represent specific cases of the Black-Scholes-Merton process; for instance, the `BlackScholesProcess` class assumes that there are no dividends, and the `BlackProcess` class that the cost of carry is equal to 0. It is the latter, or rather a glitch in it, that is the subject of this notebook.

All such classes inherit from a base `GeneralizedBlackScholesProcess` class (I know, we're not that good at naming things) that models the more general case in which the underlying stock has a continuous dividend yield. The specific cases are implemented by inheriting from this class and setting a constraint on the dividends $q(t)$: for the Black-Scholes process, $q(t) = 0$; and for the Black process, $q(t) = r(t)$, which makes the cost of carry b equal 0.

We can check the constraint by creating two instances of such processes. Here are the quotes and term structures we'll use to model the dynamics of the underlying:

```
In [3]: u = SimpleQuote(100.0)
        r = SimpleQuote(0.01)
        sigma = SimpleQuote(0.20)

        risk_free_curve = FlatForward(today, QuoteHandle(r), Actual360())
        volatility = BlackConstantVol(today, TARGET(), QuoteHandle(sigma), Actual360())
```

The constructor of the `BlackScholesProcess` class doesn't take a dividend yield, and sets it to 0 internally:

```
In [4]: process_1 = BlackScholesProcess(QuoteHandle(u),
                                        YieldTermStructureHandle(risk_free_curve),
                                        BlackVolTermStructureHandle(volatility))

        print(process_1.dividendYield().zeroRate(1.0, Continuous))

Out[4]: 0.000000 % Actual/365 (Fixed) continuous compounding
```

The constructor of the `BlackProcess` class doesn't take a dividend yield either, and sets its handle as a copy of the risk free handle:

```
In [5]: process_2 = BlackProcess(QuoteHandle(u),
                                 YieldTermStructureHandle(risk_free_curve),
                                 BlackVolTermStructureHandle(volatility))

        print(process_2.riskFreeRate().zeroRate(1.0, Continuous))
        print(process_2.dividendYield().zeroRate(1.0, Continuous))

Out[5]: 1.000000 % Actual/360 continuous compounding
        1.000000 % Actual/360 continuous compounding
```

Now, the above processes can be used to price options on underlyings behaving accordingly; the first process describes, e.g., a stock that doesn't pay any dividends, and the second describes, e.g., a futures. The classes to use are the same: `EuropeanOption` for the instrument and `AnalyticEuropeanEngine` for the pricing engine. The constructor of the engine takes an instance of `GeneralizedBlackScholesProcess`, to which both our processes can be converted implicitly.

```
In [6]: option_1 = EuropeanOption(PlainVanillaPayoff(Option.Call, 100.0),
                                  EuropeanExercise(today+100))
        option_1.setPricingEngine(AnalyticEuropeanEngine(process_1))

        print(option_1.NPV())

Out[6]: 4.337597216336533

In [7]: option_2 = EuropeanOption(PlainVanillaPayoff(Option.Call, 100.0),
                                  EuropeanExercise(today+100))
        option_2.setPricingEngine(AnalyticEuropeanEngine(process_2))

        print(option_2.NPV())

Out[7]: 4.191615257389808
```

So far, so good. However, we can see the glitch when we ask the options for their Greeks. With this particular engine, they're able to calculate them by using closed formulas (none other, of course, that

those expressing the derivatives of the Black-Scholes-Merton formula). As I described in a previous notebook, we can also calculate the Greeks numerically, by bumping the inputs and repricing the option. If we compare the two approaches, they should yield approximately the same results.

For convenience, I'll define a utility function to calculate numerical Greeks. It takes the option, the quote to change and the amplitude of the bump.

```
In [8]: def greek(option, quote, dx):
            x0 = quote.value()
            quote.setValue(x0+dx)
            P_u = option.NPV()
            quote.setValue(x0-dx)
            P_d = option.NPV()
            quote.setValue(x0)
            return (P_u-P_d)/(2*dx)
```

By passing different quotes, we can calculate different Greeks. Bumping the underlying value will give us the delta, which we can compare to the analytic result:

```
In [9]: print(option_1.delta())
        print(greek(option_1, u, 0.01))

Out[9]: 0.5315063340142601
        0.531506323010289

In [10]: print(option_2.delta())
         print(greek(option_2, u, 0.01))

Out[10]: 0.5195711146255227
         0.5195711052036867
```

Bumping the volatility gives us the vega…

```
In [11]: print(option_1.vega())
         print(greek(option_1, sigma, 0.001))

Out[11]: 20.96050033373808
         20.960499909565833

In [12]: print(option_2.vega())
         print(greek(option_2, sigma, 0.001))

Out[12]: 20.938677847075486
         20.938677605407463
```

…and bumping the risk-free rate will give us the rho.

```
In [13]: print(option_1.rho())
         print(greek(option_1, r, 0.001))

Out[13]: 13.559176718080407
         13.55917453385036

In [14]: print(option_2.rho())
         print(greek(option_2, r, 0.001))

Out[14]: 13.268193390322908
         -1.1643375864700545
```

Whoops. Not what you might have expected.

What's happening here?

The problem is that the engine works with a generic process, and ρ is calculated as

$$\rho = \frac{\partial}{\partial r} C(u, r, q, \sigma)$$

where C is the Black-Scholes-Merton formula for the value of the call option.

However, not knowing about the specific process type we passed, the engine doesn't know about the constraint we set on the underlying variables: in this case, that $q = q(r) = r$. Therefore, the correct value for ρ should be

$$\rho = \frac{d}{dr} C(u, r, q(r), \sigma) = \frac{\partial C}{\partial r} + \frac{\partial C}{\partial q} \cdot \frac{\partial q}{\partial r} = \frac{\partial C}{\partial r} + \frac{\partial C}{\partial q}.$$

which is the sum of the rho as defined in the engine and the dividend rho. We can verify this by comparing the above with the numerical Greek:

```
In [15]: print(option_2.rho() + option_2.dividendRho())
         print(greek(option_2, r, 0.001))

Out[15]: -1.1643375714971693
         -1.1643375864700545
```

Now: is this a bug in the engine?

Well, it might be argued. The engine might detect the case of a Black process and change the calculation of rho accordingly; it's kind of a hack, and there goes the genericity, but it's possible to implement. However, the above might also happen with a usually well-behaved process if we use the same term structure for r and q:

```
In [16]: process_3 = BlackScholesMertonProcess(QuoteHandle(u),
                                                YieldTermStructureHandle(risk_free_curve),
                                                YieldTermStructureHandle(risk_free_curve),
                                                BlackVolTermStructureHandle(volatility))
         option_3 = EuropeanOption(PlainVanillaPayoff(Option.Call, 100.0),
                                   EuropeanExercise(today+100))
         option_3.setPricingEngine(AnalyticEuropeanEngine(process_3))

In [17]: print(option_3.delta())
         print(greek(option_3, u, 0.01))

Out[17]: 0.5195711146255227
         0.5195711052036867

In [18]: print(option_3.rho())
         print(greek(option_3, r, 0.001))
         print(option_3.rho() + option_3.dividendRho())

Out[18]: 13.268193390322908
         -1.1643375864700545
         -1.1643375714971693
```

The issue is not even limited to processes. You're defining a discount curve as the risk-free rate plus a credit spread? Bumping the risk-free rate will modify both, and your sensitivities will be affected accordingly (even though in this case the effect is probably what you wanted). In any case, this is something you should be aware of.

27. Using curves with different day-count conventions

(Based on a question by Vinod Rajakumar[1] on the QuantLib mailing list. Thanks!)

Like a number of other notebooks, this one describes a glitch in the library that you might want to be aware of.

The problem

Let's say we're pricing an option. We've already seen it in other notebooks, so I'll go through the setup without much details:

```
In [1]: from QuantLib import *
        import math

In [2]: today = Date(27, July, 2018)
        Settings.instance().evaluationDate = today
        calendar = UnitedStates(UnitedStates.NYSE)

In [3]: exercise_date = today+Period(3,Months)
        strike = 100.0
        option = EuropeanOption(PlainVanillaPayoff(Option.Call, strike),
                                EuropeanExercise(exercise_date))
```

I'll set up handles for the needed curves, so we can change them later...

```
In [4]: u = RelinkableQuoteHandle()
        r = RelinkableYieldTermStructureHandle()
        sigma = RelinkableBlackVolTermStructureHandle()

In [5]: process = BlackScholesProcess(u, r, sigma)
```

...and I'll use the process above to instantiate two different engines: the first uses the analytic Black-Scholes formula, and the second a finite-difference model.

[1] https://sourceforge.net/p/quantlib/mailman/message/36015671/

```
In [6]: analytic_engine = AnalyticEuropeanEngine(process)

        fd_engine = FDEuropeanEngine(process, 1000, 1000)
```

Now we get to pricing the option. First, I'll link the risk-free rate and the volatility to two constant curves with the same day-count convention (in this case, Actual/365 fixed). Let's say the risk-free rate is 0% and the volatility is 20%.

```
In [7]: u.linkTo(SimpleQuote(100.0))
        r.linkTo(FlatForward(today, 0.0, Actual365Fixed()))
        sigma.linkTo(BlackConstantVol(today, calendar, 0.20, Actual365Fixed()))
```

With this setup, the two engines give the same result (within numerical error) and everybody is happy:

```
In [8]: option.setPricingEngine(analytic_engine)
        option.NPV()

Out[8]: 4.004101982740124


In [9]: option.setPricingEngine(fd_engine)
        option.NPV()

Out[9]: 4.004154055896805
```

However, things are not always so simple. For instance, the volatility might have been quoted with a different day-count convention, as is practice on some markets. Let's say, for instance, that the 20% volatility was quoted based on the commonly used business/252 convention.

```
In [10]: sigma.linkTo(BlackConstantVol(today, calendar, 0.20, Business252(calendar)))
```

In this case, we're not so lucky; the results from the two engines differ significantly.

```
In [11]: option.setPricingEngine(analytic_engine)
         option.NPV()

Out[11]: 4.050510859367279


In [12]: option.setPricingEngine(fd_engine)
         option.NPV()

Out[12]: 4.004154055896805
```

By looking at the numbers, we can see that analytic engine reacts to the change, while the finite-differences engine doesn't.

What is happening?

This is not something that could be expected; unfortunately, it's an artifact of the implementation and could only be deduced by looking at the code. Specifically, the analytic engine is able to include in the calculation the day-count convention of the volatility curve, while the FD model is forced to use one single time grid and can't account for different conventions.

More in detail, what the FD engine does is to ask the curve for the volatility at the exercise date…

```
In [13]: vol = sigma.blackVol(exercise_date, strike)
         vol

Out[13]: 0.2
```

…and use it on the grid. However, the time grid on which the FD model works uses the day-count convention of the risk-free rates, resulting in a time to maturity that is inconsistent with the volatility quote…

```
In [14]: T_vol = sigma.dayCounter().yearFraction(today, exercise_date)
         T_vol

Out[14]: 0.25793650793650796


In [15]: T_grid = r.dayCounter().yearFraction(today, exercise_date)
         T_grid

Out[15]: 0.25205479452054796
```

…and therefore the wrong value for the variance of the stock price:

```
In [16]: vol*vol*T_grid

Out[16]: 0.01008219178082192


In [17]: var = sigma.blackVariance(exercise_date, strike)
         var

Out[17]: 0.01031746031746032
```

An attempt at a solution

In this case, and having assessed the problem, we can work around the problem; that is, we can find the volatility that, together with the day-count convention used on the grid, gives the correct variance.

```
In [18]: vol = math.sqrt(var/T_grid)
         vol

Out[18]: 0.20232004929429467
```

This synthetic value can be used to build a volatility curve with the same day-count convention as the rate. This allows the FD engine to return a more correct value.

```
In [19]: sigma.linkTo(BlackConstantVol(today, calendar, vol, Actual365Fixed()))

In [20]: option.setPricingEngine(analytic_engine)
         option.NPV()

Out[20]: 4.050510859367279


In [21]: option.setPricingEngine(fd_engine)
         option.NPV()

Out[21]: 4.050563403337715
```

Of course, this is more cumbersome if the volatility is not flat; you might have to convert multiple values if you're interpolating them, or sample multiple values and then convert them if the curve is of some other kind.

On the whole, it is unfortunate that the implementation is leaking into the use of the engine. We still don't have a solution, though. What I can suggest is, when possible, to perform sanity checks like the previous comparison between engine results. This will give you information on the underlying implementation and the precautions you'll have to take when a comparison is not possible (such as, for instance, when the option is American and there's no corresponding analytic engine).

Bonds

28. Modeling fixed rate bonds

In this chapter we will consider a simple example to model fixed rate bonds. Let's consider a hypothetical bond with a par value of 100, that pays 6% coupon semi-annually issued on January 15th, 2015 and set to mature on January 15th, 2016. The bond will pay a coupon on July 15th, 2015 and January 15th, 2016. The par amount of 100 will also be paid on the January 15th, 2016.

To make things simpler, lets assume that we know the spot rates of the treasury as of January 15th, 2015. The annualized spot rates are 0.5% for 6 months and 0.7% for 1 year point. Lets calculate the net present value of the cash flows directly as shown below.

```
In [1]: 3/pow(1+0.005, 0.5) + (100 + 3)/(1+0.007)
```

```
Out[1]: 105.27653992490681
```

Here, we discounted the coupon and par payments with the appropriate spot rates. Now we will replicate this calculation using the QuantLib framework. Let's start by importing the `QuantLib` module.

```
In [2]: from QuantLib import *
```

As a first step, we need to construct a yield curve with the given spot rates. This is done using the `ZeroCurve` class as discussed in an earlier chapter.

```
In [3]: calc_date = Date(15, 1, 2015)
        Settings.instance().evaluationDate = calc_date
        spot_dates = [Date(15, 1, 2015), Date(15, 7, 2015), Date(15, 1, 2016)]
        spot_rates = [0.0, 0.005, 0.007]
        day_count = Thirty360()
        calendar = UnitedStates()
        interpolation = Linear()
        compounding = Compounded
        compounding_frequency = Annual
        spot_curve = ZeroCurve(spot_dates, spot_rates, day_count, calendar,
                               interpolation, compounding, compounding_frequency)
        spot_curve_handle = YieldTermStructureHandle(spot_curve)
```

As a next step, we are going to build a fixed rate bond object. In order to construct the `FixedRateBond` object, we will create a schedule for the coupon payments. Creation of `Schedule` object was discussed in an earlier chapter.

```
In [4]: issue_date = Date(15, 1, 2015)
        maturity_date = Date(15, 1, 2016)
        tenor = Period(Semiannual)
        calendar = UnitedStates()
        business_convention = Unadjusted
        date_generation = DateGeneration.Backward
        month_end = False
        schedule = Schedule (issue_date, maturity_date, tenor,
                             calendar, business_convention,
                             business_convention , date_generation,
                             month_end)
```

Let us print the schedule to check if it is in agreement with what we expect it to be.

```
In [5]: list(schedule)

Out[5]: [Date(15,1,2015), Date(15,7,2015), Date(15,1,2016)]
```

Now that we have the schedule, we can create the `FixedRateBond` object. The `FixedRateBond` constructor has the following signature.

```
FixedRateBond(Natural settlementDays,
              Real faceAmount,
              const Schedule& schedule,
              const std::vector<Rate>& coupons,
              const DayCounter& accrualDayCounter,
              BusinessDayConvention paymentConvention = Following,
              Real redemption = 100.0,
              const Date& issueDate = Date(),
              const Calendar& paymentCalendar = Calendar(),
              const Period& exCouponPeriod = Period(),
              const Calendar& exCouponCalendar = Calendar(),
              const BusinessDayConvention exCouponConvention = Unadjusted,
              bool exCouponEndOfMonth = false)
```

Let us create the `FixedRateBond` object below.

```
In [6]: coupon_rate = .06
        coupons = [coupon_rate]
        settlement_days = 0
        face_value = 100

        fixed_rate_bond = FixedRateBond(settlement_days,
                                        face_value,
                                        schedule,
                                        coupons,
                                        day_count)
```

Now the we have the fixed rate bond instrument, we need a valuation engine in order to price this bond. The fixed rate bond can be priced using a `DiscountingBondEngine`. The `DiscountingBondEngine` takes the yield curve object as an argument in its constructor. The `setPricingEngine` method in the fixed rate bond instrument is used to set the pricing engine.

```
In [7]: bond_engine = DiscountingBondEngine(spot_curve_handle)
        fixed_rate_bond.setPricingEngine(bond_engine)
```

Now, the net present value of the bond can be extracted using the `NPV` method.

```
In [8]: fixed_rate_bond.NPV()

Out[8]: 105.27653992490683


In [9]: fixed_rate_bond.cleanPrice()

Out[9]: 105.27653992490683
```

We can also obtain various other analytics for the bond.

```
In [10]: fixed_rate_bond.accruedAmount()

Out[10]: 0.0


In [11]: fixed_rate_bond.dirtyPrice()

Out[11]: 105.27653992490683


In [12]: fixed_rate_bond.bondYield(day_count,
                                   compounding,
                                   compounding_frequency)

Out[12]: 0.006971154634952549
```

29. Building irregular bonds

(Based on a question[1] by *Stack Exchange* user user7922, another[2] by user Lisa Ann, and yet another[3] asked by Anthony Calleja on the QuantLib mailing list. Thanks!)

```
In [1]: from QuantLib import *
        from datetime import date
        from pandas import DataFrame
        import utils
```

Let me just define a small helper function before starting. It's just for visualization, nothing to write about.

```
In [2]: def rate_if_available(c):
            c = as_coupon(c)
            return utils.format_rate(c.rate()) if c else ''
```

The first question

user7922 had to price a bond that, curiously, had a last coupon date before the maturity date; e.g., the last coupon date is April 20th, 2020 and maturity date is April 20th, 2021. Yes, it's strange, but who are we to judge?

There's no way to express this directly in QuantLib, but we can get it with some work. In case of a fixed-rate bond, and if we want to do the simplest thing that can possibly work, we can cancel the last coupon by giving it a rate of 0%.

```
In [3]: today = Date(8, October, 2014)
        Settings.instance().evaluationDate = today

In [4]: issueDate = today+7
        maturityDate = issueDate+Period(10,Years)

        schedule = Schedule(issueDate, maturityDate,
                            Period(Annual), TARGET(), Following, Following,
                            DateGeneration.Backward, False)
```

The schedule we just built gives us the correct maturity date, as well as the date where we want the last real coupon. Now for the bond:

[1] http://quant.stackexchange.com/questions/11090/
[2] http://quant.stackexchange.com/questions/9080/
[3] https://sourceforge.net/p/quantlib/mailman/message/36170786/

```
In [5]: settlementDays = 3
        faceAmount = 100
        paymentDayCounter = Thirty360()

        coupon_rate = 0.02
        N = len(schedule)-1   # number of coupons
        coupons = [coupon_rate]*(N-1) + [0.0]

        bond = FixedRateBond(settlementDays,
                             faceAmount,
                             schedule,
                             coupons,
                             paymentDayCounter)
```

```
In [6]: DataFrame([ (c.date(), rate_if_available(c), c.amount())
                    for c in bond.cashflows() ],
                  columns = ('date', 'rate', 'amount'),
                  index=['']*len(bond.cashflows()))
```

Out[6]:

date	rate	amount
October 15th, 2015	2.00 %	2.000000
October 17th, 2016	2.00 %	2.011111
October 16th, 2017	2.00 %	1.994444
October 15th, 2018	2.00 %	1.994444
October 15th, 2019	2.00 %	2.000000
October 15th, 2020	2.00 %	2.000000
October 15th, 2021	2.00 %	2.000000
October 17th, 2022	2.00 %	2.011111
October 16th, 2023	2.00 %	1.994444
October 15th, 2024	0.00 %	0.000000
October 15th, 2024		100.000000

The same trick also works for floating-rate coupons, if we use a null gearing for the last coupon:

```
In [7]: euribor_curve = FlatForward(0, TARGET(), 0.002, Actual360())
        index = Euribor1Y(YieldTermStructureHandle(euribor_curve))
        N = len(schedule)-1   # number of coupons
        gearings = [1.0]*(N-1) + [0.0]
        bond = FloatingRateBond(settlementDays = 3,
                                faceAmount = 100,
                                schedule = schedule,
                                index = index,
                                paymentDayCounter = Thirty360(),
                                paymentConvention = Following,
                                fixingDays = index.fixingDays(),
```

```
                                    gearings = gearings,
                                    spreads = [],
                                    caps= [],
                                    floors = [],
                                    inArrears = False,
                                    redemption = 100.0,
                                    issueDate = issueDate)
```

```
In [8]: DataFrame([ (c.date(), rate_if_available(c), c.amount())
                    for c in bond.cashflows() ],
                   columns = ('date', 'rate', 'amount'),
                   index=['']*len(bond.cashflows()))
```

Out[8]:

date	rate	amount
October 15th, 2015	0.20 %	0.200203
October 17th, 2016	0.20 %	0.201317
October 16th, 2017	0.20 %	0.199646
October 15th, 2018	0.20 %	0.199646
October 15th, 2019	0.20 %	0.200203
October 15th, 2020	0.20 %	0.200203
October 15th, 2021	0.20 %	0.200203
October 17th, 2022	0.20 %	0.201316
October 16th, 2023	0.20 %	0.199646
October 15th, 2024	0.00 %	0.000000
October 15th, 2024		100.000000

However, that's a bit sloppy. The coupon paying a null rate looks strange, and might confuse cash-flow analysis. It's better, even if it takes a bit more work, to remove the coupon altogether. We can get the cash flows from the bond we created…

```
In [9]: cashflows = list(bond.cashflows())
```

…delete the ones we don't want to keep, that is, the one before the last (the last being the redemption)…

```
In [10]: del cashflows[-2]
```

…and use the cleaned-up cash flows to create a new bond:

```
In [11]: bond = Bond(3, TARGET(), 100.0,
                     maturityDate, issueDate,
                     cashflows)
```

This gives us the coupons we wanted:

```
In [12]: DataFrame([ (c.date(), rate_if_available(c), c.amount())
                     for c in bond.cashflows() ],
                   columns = ('date', 'rate', 'amount'),
                   index=['']*len(bond.cashflows()))
```

Out[12]:

date	rate	amount
October 15th, 2015	0.20 %	0.200203
October 17th, 2016	0.20 %	0.201317
October 16th, 2017	0.20 %	0.199646
October 15th, 2018	0.20 %	0.199646
October 15th, 2019	0.20 %	0.200203
October 15th, 2020	0.20 %	0.200203
October 15th, 2021	0.20 %	0.200203
October 17th, 2022	0.20 %	0.201316
October 16th, 2023	0.20 %	0.199646
October 15th, 2024		100.000000

The second question

Lisa Ann had to price a more common instrument (namely, a fixed-to-floater) for which, however, there's no specific class in the library. In this case, too, we can create the required bond by manipulating coupons; and in this case, too, we can choose how much work to do.

Let's say the bond pays three fixed-rate coupon first and then seven floating-rate coupons. We might create the fixed-rate coupons…

```
In [13]: schedule = Schedule(issueDate, issueDate+Period(3,Years),
                             Period(Annual), TARGET(), Following, Following,
                             DateGeneration.Backward, False)

         fixed = FixedRateLeg(schedule = schedule,
                              dayCount = Actual360(),
                              nominals = [100.0],
                              couponRates = [0.02])
```

…then the floating-rate coupons…

```
In [14]: schedule = Schedule(issueDate+Period(3,Years), maturityDate,
                             Period(Annual), TARGET(), Following, Following,
                             DateGeneration.Backward, False)

         floating = IborLeg(nominals = [100.0],
                            schedule = schedule,
                            index = index,
                            spreads = [0.001])
```

...and finally put them together, add the redemption, and build a custom bond:

```
In [15]: bond = Bond(3, TARGET(), 100.0,
                     maturityDate, issueDate,
                     fixed + floating + (Redemption(100.0, maturityDate),))

In [16]: DataFrame([ (c.date(), rate_if_available(c), c.amount())
                     for c in bond.cashflows() ],
                   columns = ('date', 'rate', 'amount'),
                   index=['']*len(bond.cashflows()))
```

Out[16]:

date	rate	amount
October 15th, 2015	2.00 %	2.027778
October 17th, 2016	2.00 %	2.044444
October 16th, 2017	2.00 %	2.022222
October 15th, 2018	0.30 %	0.303538
October 15th, 2019	0.30 %	0.304372
October 15th, 2020	0.30 %	0.305207
October 15th, 2021	0.30 %	0.304372
October 17th, 2022	0.30 %	0.306041
October 16th, 2023	0.30 %	0.303538
October 15th, 2024	0.30 %	0.304372
October 15th, 2024		100.000000

However, I'm not very comfortable with building the coupons with two half-schedules; I haven't looked very hard for a counter-example, but I suspect that some combination of holidays and business-day conventions might cause the coupon dates to be off.

A safer alternative would be to build both fixed and floating coupons over the full bond schedule, and just keep those we need:

```
In [17]: schedule = Schedule(issueDate, maturityDate,
                             Period(Annual), TARGET(), Following, Following,
                             DateGeneration.Backward, False)

         fixed = FixedRateLeg(schedule = schedule,
                              dayCount = Actual360(),
                              nominals = [100.0],
                              couponRates = [0.02])

         floating = IborLeg(nominals = [100.0],
                            schedule = schedule,
                            index = index,
                            spreads = [0.001])

         cashflows = fixed[:3] + floating[3:] + (Redemption(100.0, maturityDate),)

In [18]: bond = Bond(3, TARGET(), 100.0,
                     maturityDate, issueDate, cashflows)

In [19]: DataFrame([ (c.date(), rate_if_available(c), c.amount())
                     for c in bond.cashflows() ],
                   columns = ('date', 'rate', 'amount'),
                   index=['']*len(bond.cashflows()))

Out[19]:
```

date	rate	amount
October 15th, 2015	2.00 %	2.027778
October 17th, 2016	2.00 %	2.044444
October 16th, 2017	2.00 %	2.022222
October 15th, 2018	0.30 %	0.303538
October 15th, 2019	0.30 %	0.304372
October 15th, 2020	0.30 %	0.305207
October 15th, 2021	0.30 %	0.304372
October 17th, 2022	0.30 %	0.306041
October 16th, 2023	0.30 %	0.303538
October 15th, 2024	0.30 %	0.304372
October 15th, 2024		100.000000

Also, in a pinch (for instance, if you're using the QuantLib Excel module and can't create custom bonds easily) a fixed-rate coupon can be approximated by a floating-rate coupon with a null gearing and a spread equal to the desired rate, so you might get the same result this way:

```
In [20]: schedule = Schedule(issueDate, maturityDate,
                             Period(Annual), TARGET(), Following, Following,
                             DateGeneration.Backward, False)

         gearings = [0.0]*3 + [1.0]*7
         spreads = [0.02]*3 + [0.001]*7

         bond = FloatingRateBond(settlementDays = 3,
                                 faceAmount = 100,
                                 schedule = schedule,
                                 index = index,
                                 paymentDayCounter = Actual360(),
                                 paymentConvention = Following,
                                 fixingDays = index.fixingDays(),
                                 gearings = gearings,
                                 spreads = spreads,
                                 caps= [],
                                 floors = [],
                                 inArrears = False,
                                 redemption = 100.0,
                                 issueDate = issueDate)

In [21]: DataFrame([ (c.date(), rate_if_available(c), c.amount())
                     for c in bond.cashflows() ],
                   columns = ('date', 'rate', 'amount'),
                   index=['']*len(bond.cashflows()))

Out[21]:
```

date	rate	amount
October 15th, 2015	2.00 %	2.027778
October 17th, 2016	2.00 %	2.044444
October 16th, 2017	2.00 %	2.022222
October 15th, 2018	0.30 %	0.303538
October 15th, 2019	0.30 %	0.304372
October 15th, 2020	0.30 %	0.305207
October 15th, 2021	0.30 %	0.304372
October 17th, 2022	0.30 %	0.306041
October 16th, 2023	0.30 %	0.303538
October 15th, 2024	0.30 %	0.304372
October 15th, 2024		100.000000

However, I don't suggest doing this if you can get actual fixed- and floating-rate coupons.

The third question

This one requires a bit more work (and involved a swap, instead of a bond, but it doesn't matter; you can build custom swaps with the `Swap` class). Anthony needed a floating leg paying 6-months Euribor, but with a short initial stub paying the fixing of 3-months Euribor instead. In a vanilla leg, even with the correct schedule, the first coupon would pay the 6-months fixing instead:

```
In [22]: euribor_curve_3m = FlatForward(0, TARGET(), 0.0015, Actual360())
         index_3m = Euribor3M(YieldTermStructureHandle(euribor_curve_3m))

         euribor_curve_6m = FlatForward(0, TARGET(), 0.0020, Actual360())
         index_6m = Euribor6M(YieldTermStructureHandle(euribor_curve_6m))

In [23]: startDate = today + 7
         endDate = startDate + Period(3,Months) + Period(5,Years)
         schedule = Schedule(startDate, endDate,
                             Period(Semiannual), TARGET(), Following, Following,
                             DateGeneration.Backward, False)

         cashflows = IborLeg(nominals = [100.0],
                             schedule = schedule,
                             index = index_6m)

In [24]: DataFrame([ (c.date(),
                      as_coupon(c).accrualStartDate(), as_coupon(c).accrualEndDate(),
                      utils.format_rate(as_coupon(c).rate()), c.amount())
                     for c in cashflows ],
                   columns = ('payment date', 'start date', 'end date',
                              'rate', 'amount'),
                   index=['']*len(cashflows))

Out[24]:
```

payment date	start date	end date	rate	amount
January 15th, 2015	October 15th, 2014	January 15th, 2015	0.20 %	0.051124
July 15th, 2015	January 15th, 2015	July 15th, 2015	0.20 %	0.100606
January 15th, 2016	July 15th, 2015	January 15th, 2016	0.20 %	0.102274
July 15th, 2016	January 15th, 2016	July 15th, 2016	0.20 %	0.101162
January 16th, 2017	July 15th, 2016	January 16th, 2017	0.20 %	0.102831
July 17th, 2017	January 16th, 2017	July 17th, 2017	0.20 %	0.101162
January 15th, 2018	July 17th, 2017	January 15th, 2018	0.20 %	0.101162
July 16th, 2018	January 15th, 2018	July 16th, 2018	0.20 %	0.101162
January 15th, 2019	July 16th, 2018	January 15th, 2019	0.20 %	0.101718
July 15th, 2019	January 15th, 2019	July 15th, 2019	0.20 %	0.100606
January 15th, 2020	July 15th, 2019	January 15th, 2020	0.20 %	0.102274

The first coupon has the correct dates, but the rate is wrong. To use the right one, we have to build a custom first coupon with the correct index and use it instead of the current one. We also need to set it a pricer (which is usually done for us by `IborLeg`).

```
In [25]: first = as_floating_rate_coupon(cashflows[0])
         coupon3m = IborCoupon(first.date(), first.nominal(),
                               first.accrualStartDate(), first.accrualEndDate(),
                               first.fixingDays(), index_3m)
         coupon3m.setPricer(BlackIborCouponPricer())

         cashflows = (coupon3m,) + cashflows[1:]
```

```
In [26]: DataFrame([ (c.date(),
                      as_coupon(c).accrualStartDate(), as_coupon(c).accrualEndDate(),
                      utils.format_rate(as_coupon(c).rate()), c.amount())
                    for c in cashflows ],
                   columns = ('payment date', 'start date', 'end date',
                              'rate', 'amount'),
                   index=['']*len(cashflows))
```

Out[26]:

payment date	start date	end date	rate	amount
January 15th, 2015	October 15th, 2014	January 15th, 2015	0.15 %	0.038341
July 15th, 2015	January 15th, 2015	July 15th, 2015	0.20 %	0.100606
January 15th, 2016	July 15th, 2015	January 15th, 2016	0.20 %	0.102274
July 15th, 2016	January 15th, 2016	July 15th, 2016	0.20 %	0.101162
January 16th, 2017	July 15th, 2016	January 16th, 2017	0.20 %	0.102831
July 17th, 2017	January 16th, 2017	July 17th, 2017	0.20 %	0.101162
January 15th, 2018	July 17th, 2017	January 15th, 2018	0.20 %	0.101162
July 16th, 2018	January 15th, 2018	July 16th, 2018	0.20 %	0.101162
January 15th, 2019	July 16th, 2018	January 15th, 2019	0.20 %	0.101718
July 15th, 2019	January 15th, 2019	July 15th, 2019	0.20 %	0.100606
January 15th, 2020	July 15th, 2019	January 15th, 2020	0.20 %	0.102274

As before, the resulting cash flows can be used to instantiate a bond or a swap.

30. Valuation of bonds with credit spreads

In an earlier example on pricing fixed rate bonds, I demonstrated how to construct and value bonds using the given yield curve. In this example, let us take a look at valuing bonds with credit spreads. We will show how to add credit spreads to the give yield curve using different approaches.

As usual, let us start by importing the QuantLib library and pick a valuation date and set the calculation instance evaluation date.

```
In [1]: from QuantLib import *
        calc_date = Date(26, 7, 2016)
        Settings.instance().evaluationDate = calc_date
```

For simplicity, let us imagine that the treasury yield curve is flat. This makes it easier to construct the yield curve easily. This also allows us to directly shock the yield curve, and provides a validation for the more general treatment of shocks on yield curve.

```
In [2]: flat_rate = SimpleQuote(0.0015)
        rate_handle = QuoteHandle(flat_rate)
        day_count = Actual360()
        calendar = UnitedStates()
        ts_yield = FlatForward(calc_date, rate_handle, day_count)
        ts_handle = YieldTermStructureHandle(ts_yield)
```

Now let us construct the bond itself. We do that by first constructing the schedule, and then passing the schedule into the bond.

```
In [3]: issue_date = Date(15, 7, 2016)
        maturity_date = Date(15, 7, 2021)
        tenor = Period(Semiannual)
        calendar = UnitedStates()
        bussiness_convention = Unadjusted
        date_generation = DateGeneration.Backward
        month_end = False
        schedule = Schedule (issue_date, maturity_date,
                             tenor, calendar,
                             bussiness_convention,
                             bussiness_convention,
                             date_generation,
                             month_end)
```

```
In [4]: settlement_days = 2
        day_count = Thirty360()
        coupon_rate = .03
        coupons = [coupon_rate]

        # Now lets construct the FixedRateBond
        settlement_days = 0
        face_value = 100
        fixed_rate_bond = FixedRateBond(
            settlement_days,
            face_value,
            schedule,
            coupons,
            day_count)
```

Now that we have the `fixed_rate_bond` object, we can create a `DiscountingBondEngine` and value the bond.

```
In [5]: bond_engine = DiscountingBondEngine(ts_handle)
        fixed_rate_bond.setPricingEngine(bond_engine)
        fixed_rate_bond.NPV()

Out[5]: 114.18461651948999
```

So far, we have valued the bond under the treasury yield curve and have not incorporated the credit spreads. Let us assume that the market prices this bond with a `50BP` spread on top of the treasury yield curve. Now we can, in this case, directly shock the `flat_rate` used in the yield term structure. Let us see what the value is:

```
In [6]: flat_rate.setValue(0.0065)
        fixed_rate_bond.NPV()

Out[6]: 111.5097766266561
```

Above we shocked the `flat_rate` and since the yield term structure is an `Observer` observing the `Observable` `flat_rate`, we could just shock the rate, and QuantLib behind the scenes recalculates all the `Observers`. Though, this approach is not always viable, in cases such as a bootstrapped bond curve. So let us look at two different approaches that can be used. Before we do that, we need to reset the `flat_rate` back to what it was.

```
In [7]: flat_rate.setValue(0.0015)
        fixed_rate_bond.NPV()

Out[7]: 114.18461651948999
```

Parallel Shift of the Yield Curve

The whole yield curve can be shifted up and down, and the bond revalued with the help of the `ZeroSpreadedTermStructure`. The constructor takes the yield curve and the spread as argument.

```
In [8]: spread1 = SimpleQuote(0.0050)
        spread_handle1 = QuoteHandle(spread1)
        ts_spreaded1 = ZeroSpreadedTermStructure(ts_handle, spread_handle1)
        ts_spreaded_handle1 = YieldTermStructureHandle(ts_spreaded1)

        bond_engine = DiscountingBondEngine(ts_spreaded_handle1)
        fixed_rate_bond.setPricingEngine(bond_engine)

        # Finally the price
        fixed_rate_bond.NPV()

Out[8]: 111.50977662665609
```

Once we have constructed the spreaded term structure, it is rather easy to value for other spreads. All we need to do is change the `SimpleQuote` object `spread1` here.

```
In [9]: spread1.setValue(0.01)
        fixed_rate_bond.NPV()

Out[9]: 108.89999943320038
```

Non-Parallel Shift of the Yield Curve

The above method allows only for parallel shift of the yield curve. The `SpreadedLinearZeroInterpolatedTermStructure` class allows for non parallel shock. First, let us mimic a parallel shift using this class. For the constructor, we need to pass the yield term structure that we wish to shift, and the a list of spreads and a list of the corresponding dates.

```
In [10]: spread21 = SimpleQuote(0.0050)
         spread22 = SimpleQuote(0.0050)
         start_date = calc_date
         end_date = calendar.advance(start_date, Period(50, Years))
         ts_spreaded2 = SpreadedLinearZeroInterpolatedTermStructure(
             ts_handle,
             [QuoteHandle(spread21), QuoteHandle(spread22)],
             [start_date, end_date]
         )
         ts_spreaded_handle2 = YieldTermStructureHandle(ts_spreaded2)

         bond_engine = DiscountingBondEngine(ts_spreaded_handle2)
         fixed_rate_bond.setPricingEngine(bond_engine)

         # Finally the price
         fixed_rate_bond.NPV()


Out[10]: 111.50977662665609
```

Here, once again we can change the value of `spread2` to value for other shocks.

```
In [11]: spread21.setValue(0.01)
         spread22.setValue(0.01)
         fixed_rate_bond.NPV()

Out[11]: 108.89999943320038
```

We can easily do non-parallel shifts just by shocking one end.

```
In [12]: spread21.setValue(0.005)
         spread22.setValue(0.01)
         fixed_rate_bond.NPV()

Out[12]: 111.25358792334083
```

The `SpreadedLinearZeroInterpolatedTermStructure` is a powerful class and can be used to implement key-rate duration calculations.

31. Modeling callable bonds

In this chapter, lets take a look at valuing callable bonds in QuantLib Python. The approach to construct a callable bond is lot similar to modeling a fixed rate bond in QuantLib. The one additional input that we need to provide here is the details on the call or put schedule. If you follow the fixed rate bond example already, this should be fairly straight forward.

As always, we will start with some initializations and imports.

```
In [1]: from QuantLib import *
        import numpy as np
        import utils
        %matplotlib inline
        calc_date = Date(16,8,2016)
        Settings.instance().evaluationDate = calc_date
```

For simplicity, let us assume that the interest rate term structure is a flat yield curve at 3.5%. You can refer to constructing yield curves for more details on constructing yield curves.

```
In [2]: day_count = ActualActual(ActualActual.Bond)
        rate = 0.035
        ts = FlatForward(calc_date, rate,
                         day_count, Compounded,
                         Semiannual)
        ts_handle = YieldTermStructureHandle(ts)
```

The call and put schedules for the callable bond is created as shown below. We create a container for holding all the call and put dates using the `CallabilitySchedule` class. You can add each call using `Callability` class and noting as `Callability.Call` or `Callability.Put` for either a call or put.

```
In [3]: callability_schedule = CallabilitySchedule()
        call_price = 100.0
        call_date = Date(15,September,2016);
        null_calendar = NullCalendar();
        for i in range(0,24):
            callability_price  = CallabilityPrice(
                call_price, CallabilityPrice.Clean)
            callability_schedule.append(
                    Callability(callability_price,
                                Callability.Call,
```

```
                                call_date))

            call_date = null_calendar.advance(call_date, 3,
                                              Months)
```

What follows next is similar to the `Schedule` that we created in the vanilla fixed rate bond valuation.

```
In [4]: issue_date = Date(16,September,2014)
        maturity_date = Date(15,September,2022)
        calendar = UnitedStates(UnitedStates.GovernmentBond)
        tenor = Period(Quarterly)
        accrual_convention = Unadjusted

        schedule = Schedule(issue_date, maturity_date, tenor,
                            calendar, accrual_convention,
                            accrual_convention,
                            DateGeneration.Backward, False)
```

The callable bond is instantiated using the `CallableFixedRateBond` class, which accepts the bond inputs and the call or put schedule.

```
In [5]: settlement_days = 3
        face_amount = 100
        accrual_daycount = ActualActual(ActualActual.Bond)
        coupon = 0.025


        bond = CallableFixedRateBond(
            settlement_days, face_amount,
            schedule, [coupon], accrual_daycount,
            Following, face_amount, issue_date,
            callability_schedule)
```

In order to value the bond, we need an interest rate model to model the fact that the bond will get called or not in the future depending on where the future interest rates are at. The `TreeCallable-FixedRateBondEngine` can be used to value the callable bond. Below, the `value_bond` function prices the callable bond based on the Hull-White model parameter for mean reversion and volatility.

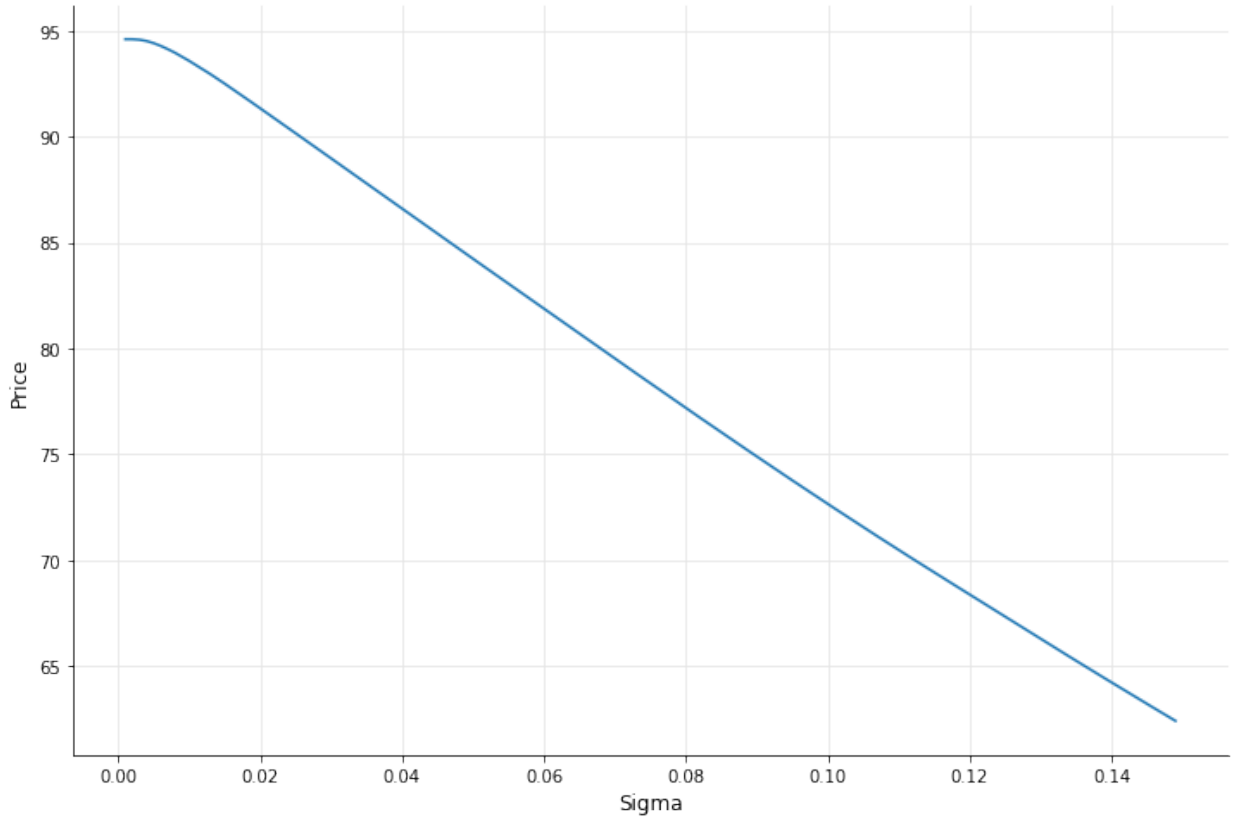
```
In [6]: def value_bond(a, s, grid_points, bond):
            model = HullWhite(ts_handle, a, s)
            engine = TreeCallableFixedRateBondEngine(model, grid_points)
            bond.setPricingEngine(engine)
            return bond
```

The callable bond value for a 3% mean reversion and 12% volatility is shown below.

```
In [7]: value_bond(0.03, 0.12, 40, bond)
        print("Bond price: %lf" % bond.cleanPrice())

Out[7]: Bond price: 68.376965
```

The price sensitivity of callable bonds to that of volatility parameter is shown below. As volatility increases, there is a higher chance of it being callable. Hence the value of the bond decreases.

```
In [8]: sigmas = np.arange(0.001, 0.15, 0.001)
        prices = [value_bond(0.03, s, 40, bond).cleanPrice()
                  for s in sigmas]

In [9]: _, ax = utils.plot()
        ax.plot(sigmas, prices)
        ax.set_xlabel("Sigma", size=12)
        ax.set_ylabel("Price", size=12);
```



The static cash flows can be accessed using the `cashflows` accessor.

```
In [10]: from pandas import DataFrame
         DataFrame(
             [(cf.date(), cf.amount()) for cf in bond.cashflows()],
             columns=["Date", "Amount"],
             index=range(1, len(bond.cashflows())+1))
```

Out[10]:

	Date	Amount
1	December 15th, 2014	0.618132
2	March 16th, 2015	0.625000
3	June 15th, 2015	0.625000
4	September 15th, 2015	0.625000
5	December 15th, 2015	0.625000
6	March 15th, 2016	0.625000
7	June 15th, 2016	0.625000
8	September 15th, 2016	0.625000
9	December 15th, 2016	0.625000
10	March 15th, 2017	0.625000
11	June 15th, 2017	0.625000
12	September 15th, 2017	0.625000
13	December 15th, 2017	0.625000
14	March 15th, 2018	0.625000
15	June 15th, 2018	0.625000
16	September 17th, 2018	0.625000
17	December 17th, 2018	0.625000
18	March 15th, 2019	0.625000
19	June 17th, 2019	0.625000
20	September 16th, 2019	0.625000
21	December 16th, 2019	0.625000
22	March 16th, 2020	0.625000
23	June 15th, 2020	0.625000
24	September 15th, 2020	0.625000
25	December 15th, 2020	0.625000
26	March 15th, 2021	0.625000
27	June 15th, 2021	0.625000
28	September 15th, 2021	0.625000
29	December 15th, 2021	0.625000
30	March 15th, 2022	0.625000
31	June 15th, 2022	0.625000
32	September 15th, 2022	0.625000
33	September 15th, 2022	100.000000

Conclusion

Here we explored a minimal example on pricing a callable bond.

32. Discount margin calculation

(Based on two[1] questions[2] by *Stack Exchange* users HookahBoy and Kyle. Thanks!)

```
In [1]: from QuantLib import *
```

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

The question

Given a floating-rate bond price, we want to find the corresponding discount margin. This is one in a class of similar problems: we have a calculation which is not immediate to do directly, but is straightforward to do in the opposite direction; in this case, find the price of a bond when discounting its coupons at a spread over LIBOR.

The general idea is to implement the inverse calculation (DM to price) and then to use a solver to determine the correct input given the result. First, we build the bond.

```
In [3]: forecast_curve = RelinkableYieldTermStructureHandle()
        discount_curve = RelinkableYieldTermStructureHandle()
```

```
In [4]: index = Euribor6M(forecast_curve)
```

```
In [5]: issueDate = Date(13,October,2014)
        maturityDate = Date(13,October,2024)

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

```
In [6]: bond = FloatingRateBond(settlementDays = 3,
                                faceAmount = 100,
                                schedule = schedule,
                                index = index,
                                paymentDayCounter = Actual360(),
                                paymentConvention = Following,
                                fixingDays = index.fixingDays(),
                                gearings = [],
```

[1] http://quant.stackexchange.com/questions/8965/
[2] https://quant.stackexchange.com/questions/37705/

```
                                  spreads = [],
                                  caps= [],
                                  floors = [],
                                  inArrears = False,
                                  redemption = 100.0,
                                  issueDate = issueDate)
        bond.setPricingEngine(DiscountingBondEngine(discount_curve))
```

Now we link the forecast curve to the current Euribor curve (whatever that is; I'm using a flat one as an example, but it could as well be a real one)...

```
In [7]: forecast_curve.linkTo(FlatForward(0, TARGET(), 0.002, Actual360()))
```

...and the discount curve to the Euribor curve plus the discount margin.

```
In [8]: DM = SimpleQuote(0.0)
        discount_curve.linkTo(ZeroSpreadedTermStructure(forecast_curve,
                                                        QuoteHandle(DM)))
```

Setting a value to the DM quote will affect the bond price: this gives us the knob to manipulate in order to find the solution of our problem.

```
In [9]: print(bond.cleanPrice())

Out[9]: 100.00000000000001

In [10]: DM.setValue(0.001)
         print(bond.cleanPrice())

Out[10]: 98.99979030764418
```

To invert the calculation, we encapsulate the above into a function. The Python language makes it easier to write it in a general way; the function below takes the target price, and returns another function that takes a value for the discount margin and returns the difference between the corresponding price and the target. In C++, we would create a function object taking the target price in its constructor and returning the difference from its `operator()`.

```
In [11]: def F(price):
             def _f(s):
                 DM.setValue(s)
                 return bond.cleanPrice() - price
             return _f

In [12]: f = F(98.9997903076)
         print(f(0.0))
         print(f(0.002))

Out[12]: 1.00020969240002
         -0.9901429992548856
```

We want to find the value of the discount margin that causes the calculated price to equal the target price, that is, that causes the error to be 0; and for that, we can use a solver.

```
In [13]: margin = Brent().solve(F(99.6), 1e-8, 0.0, 1e-4)
         print(margin)

Out[13]: 0.00039870328652332745
```

We can verify that this works by setting the margin to the returned value and checking that the bond price equals the input:

```
In [14]: DM.setValue(margin)
         print(bond.cleanPrice())

Out[14]: 99.59999988275108
```

However, note that the spread above is continuously compounded. You might want to see the discount margin in the same units as the index fixings:

```
In [15]: value_date = index.valueDate(today)
         maturity_date = index.maturityDate(value_date)
         print(InterestRate(margin, discount_curve.dayCounter(),
                            Continuous, NoFrequency)
               .equivalentRate(index.dayCounter(),
                               Simple, index.tenor().frequency(),
                               value_date, maturity_date))

Out[15]: 0.039874 % Actual/360 simple compounding
```

Not just for bonds

The approach I described can be generalized to any problem in this class. Here I'll use it to get the implied volatility of an Asian option: first I'll create the instrument...

```
In [16]: exerciseDate = today + Period(1,Years)
         fixingDates = [ today + Period(n,Months) for n in range(1,12) ]
         option = DiscreteAveragingAsianOption(Average.Arithmetic,
                                               0.0, 0,
                                               fixingDates,
                                               PlainVanillaPayoff(Option.Call, 100.0),
                                               EuropeanExercise(exerciseDate))
```

...and an engine, taking care of writing the input volatility as a quote.

```
In [17]: sigma = SimpleQuote(0.20)

         riskFreeCurve = FlatForward(0, TARGET(), 0.01, Actual360())
         volatility = BlackConstantVol(0, TARGET(), QuoteHandle(sigma), Actual360())

         process = BlackScholesProcess(QuoteHandle(SimpleQuote(100.0)),
                                       YieldTermStructureHandle(riskFreeCurve),
                                       BlackVolTermStructureHandle(volatility))

In [18]: option.setPricingEngine(MCDiscreteArithmeticAPEngine(process, "pseudorandom",
                                                              requiredSamples=1000,
                                                              seed=42))
```

Now we can use the same technique as above: the function below takes a target price and returns a function from the volatility to the pricing error:

```
In [19]: def F(price):
             def _f(v):
                 sigma.setValue(v)
                 return option.NPV() - price
             return _f
```

Using a solver, we can invert it to solve for any price:

```
In [20]: print(Brent().solve(F(5.0), 1e-8, 0.20, 1e-4))

Out[20]: 0.20081193864526342

In [21]: print(Brent().solve(F(6.0), 1e-8, 0.20, 1e-4))

Out[21]: 0.24362397543255393
```

33. Duration of floating-rate bonds

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

```
In [1]: from QuantLib import *
        from pandas import DataFrame

In [2]: today = Date(8,October,2014)
        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 = RelinkableYieldTermStructureHandle()
        forecast_curve.linkTo(FlatForward(today, 0.002, Actual360(),
                                          Compounded, 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 = Euribor6M(forecast_curve)
        index.addFixing(Date(6,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 = Date(8,August,2014)
        maturityDate = Date(8,August,2019)

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

        bond = FloatingRateBond(settlementDays = 3,
                                faceAmount = 100,
                                schedule = schedule,
                                index = index,
                                paymentDayCounter = 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 = InterestRate(0.002, Actual360(), Compounded, Semiannual)
        print(BondFunctions.duration(bond, y, 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 = SimpleQuote(0.002)
        yield_curve = FlatForward(bond.settlementDate(), QuoteHandle(y),
                                  Actual360(), Compounded, 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(DiscountingBondEngine(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 = ZeroSpreadedTermStructure(forecast_curve,
                                                    QuoteHandle(SimpleQuote(0.001)))
         bond.setPricingEngine(DiscountingBondEngine(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.

34. Treasury futures contracts

In this chapter, we will learn how to value treasury futures contracts using QuantLib. The treasury futures contract gives the buyer the right to buy the underlying by the time the contract expires. The underlying is usually delivered from a basket of securities. So in order to properly value the futures contract, we would need to find the deliverable. Here we start by doing a naive calculation by constructing a fictional security. We will see what is wrong about this approach. As a next step we will perform the cheapest to deliver calculation and subsequently use that deliverable to value the same contract.

```
In [1]: from QuantLib import *
        import math
        from pandas import DataFrame
```

```
In [2]: calc_date = Date(30,11,2015)
        Settings.instance().evaluationDate = calc_date
        day_count = ActualActual()
        calendar = UnitedStates()
        bussiness_convention = Following
        end_of_month = False
        settlement_days = 0
        face_amount = 100
        coupon_frequency = Period(Semiannual)
```

Build Yield Curve

As a first step, we build the treasury curve out of the treasury securities such as T-Bills, T-Notes and Treasury bonds.

```
In [3]: prices = [99.9935,99.9576,99.8119,99.5472,99.8867,
        100.0664,99.8711,100.0547,100.3047,100.2266]

        coupon_rates = [0.0000, 0.0000, 0.0000, 0.0000, 0.00875,
                        0.0125, 0.01625, 0.02, 0.0225, 0.03]
        maturity_dates = [Date(24,12,2015), Date(25,2,2016),
                          Date(26,5,2016), Date(10,11,2016),
                          Date(30,11,2017), Date(15,11,2018),
                          Date(30,11,2020), Date(30,11,2022),
                          Date(15,11,2025), Date(15,11,2045)]

        issue_dates = [Date(25,6,2015), Date(27,8,2015),
```

```
                        Date(28,5,2015), Date(12,11,2015),
                        Date(30,11,2015), Date(16,11,2015),
                        Date(30,11,2015), Date(30,11,2015),
                        Date(16,11,2015), Date(16,11,2015)]

        coupon_frequency = Period(6, Months)

        bond_helpers = []
        for coupon, issue_date, maturity_date, price \
            in zip(coupon_rates, issue_dates, maturity_dates, prices):
            schedule = Schedule(calc_date,
                            maturity_date,
                            coupon_frequency,
                            calendar,
                            bussiness_convention,
                            bussiness_convention,
                            DateGeneration.Backward,
                            False)

            helper = FixedRateBondHelper(QuoteHandle(SimpleQuote(price)),
                                         settlement_days,
                                         face_amount,
                                         schedule,
                                         [coupon],
                                         day_count,
                                         bussiness_convention
                                         )
            bond_helpers.append(helper)
```

```
In [4]: yield_curve = PiecewiseCubicZero(calc_date, bond_helpers, day_count)
        yield_curve_handle = YieldTermStructureHandle(yield_curve)
        discount_factors = [yield_curve.discount(d) for d in maturity_dates]
        DataFrame(list(zip(maturity_dates,discount_factors)),
                  columns= ["Dates", "Discount Factors"],
                  index=['']*len(maturity_dates))
```

Out[4]:

Dates	Discount Factors
December 24th, 2015	0.999935
February 25th, 2016	0.999576
May 26th, 2016	0.998119
November 10th, 2016	0.995472
November 30th, 2017	0.981524
November 15th, 2018	0.964278
November 30th, 2020	0.920306
November 30th, 2022	0.868533
November 15th, 2025	0.799447
November 15th, 2045	0.384829

Treasury Futures

Here we want to understand how to model treasury futures contract. Let us look at the TYZ5, the treasury futures on the 10 year note for delivery in December 2015. The notional deliverable is a 10-year 6% coupon note. In reality, the seller of the futures contract can deliver from a basket of securities.

For now, lets assume that the deliverable is actually a 6% coupon 10-year note issued as of the calculation date. Let us construct a 10 year treasury note and value this security. The futures price for the TYZ5 is 127.0625.

```
In [5]: def create_tsy_security(bond_issue_date,
                               bond_maturity_date,
                               coupon_rate,
                               coupon_frequency=Period(6, Months),
                               day_count=ActualActual(),
                               calendar=UnitedStates()
                               ):
            face_value = 100.
            settlement_days = 0

            schedule = Schedule(bond_issue_date,
                                bond_maturity_date,
                                coupon_frequency,
                                calendar,
                                ModifiedFollowing,
                                ModifiedFollowing,
                                DateGeneration.Forward,
                                False)

            security = FixedRateBond(settlement_days,
                                     face_value,
                                     schedule,
                                     [coupon_rate],
```

```
                                    day_count
                                    )
            return security

In [6]: bond_issue_date = calc_date
        delivery_date = Date(1,12,2015)

        bond_maturity_date = bond_issue_date + Period(10, Years)
        day_count = ActualActual()
        coupon_frequency = Period(6, Months)
        coupon_rate = 6/100.

        deliverable = create_tsy_security(bond_issue_date,
                                          bond_maturity_date,
                                          coupon_rate,
                                          coupon_frequency,
                                          day_count,
                                          calendar
                                          )
        bond_engine = DiscountingBondEngine(yield_curve_handle)
        deliverable.setPricingEngine(bond_engine)
```

Lets calculate the Z-Spread for this deliverable. The Z-Spread is the static spread added to the yield curve to match the price of the security. This spread is a measure of the risk associated with the security. For a treasury security, you would expect this to be zero.

```
In [7]: futures_price = 127.0625
        clean_price = futures_price*yield_curve.discount(delivery_date)

        zspread = BondFunctions.zSpread(deliverable, clean_price,
                                        yield_curve, day_count,
                                        Compounded, Semiannual,
                                        calc_date)*10000
        print("Z-Spread =%3.0fbp" % (zspread))

Out[7]: Z-Spread = 71bp
```

Here we get a spread of 71 basis points. This is unusually high for a treasury futures contract.

Cheapest To Deliver

Above we used a fictional 6% coupon bond as the deliverable. In reality, the deliverable is picked from a basket of securities based on what is the cheapest to deliver. Cheapest to deliver is not the cheapest in price. The seller of the futures contract, has to buy the delivery security from the market and sell it at an adjusted futures price. The adjusted futures price is given as:

Adjusted Futures Price = Futures Price x Conversion Factor

The gain or loss to the seller is given by the basis,

Basis = Cash Price - Adjusted Futures Price

So the cheapest to deliver is expected to be the security with the lowest basis. The conversion factor for a security is the price of the security with a 6% yield. Let us look at a basket of securities that is underlying this futures contract to understand this aspect.

```
In [8]: day_count = ActualActual()
        basket = [(1.625, Date(15,8,2022), 97.921875),
                  (1.625, Date(15,11,2022), 97.671875),
                  (1.75, Date(30,9,2022), 98.546875),
                  (1.75, Date(15,5,2023), 97.984375),
                  (1.875, Date(31,8,2022), 99.375),
                  (1.875, Date(31,10,2022),99.296875),
                  (2.0, Date(31,7,2022), 100.265625),
                  (2.0, Date(15,2,2023), 100.0625),
                  (2.0, Date(15,2,2025), 98.296875),
                  (2.0, Date(15,8,2025), 98.09375),
                  (2.125, Date(30,6,2022), 101.06250),
                  (2.125, Date(15,5,2025),99.25),
                  (2.25, Date(15,11,2024),100.546875),
                  (2.25, Date(15,11,2025),100.375),
                  (2.375, Date(15,8,2024),101.671875),
                  (2.5, Date(15,8,2023),103.25),
                  (2.5, Date(15,5,2024),102.796875),
                  (2.75, Date(15,11,2023),105.0625),
                  (2.75, Date(15,2,2024),104.875)
                  ]
        securities = []
        min_basis = 100; min_basis_index = -1
        for i, b in enumerate(basket):
            coupon, maturity, price = b
            issue = maturity - Period(10, Years)
            s = create_tsy_security(issue,maturity, coupon/100.)
            bond_engine = DiscountingBondEngine(yield_curve_handle)
            s.setPricingEngine(bond_engine)
            cf = BondFunctions.cleanPrice(s,0.06,
                                          day_count, Compounded,
                                          Semiannual, calc_date)/100.
            adjusted_futures_price = futures_price * cf
            basis = price-adjusted_futures_price
            if basis< min_basis:
                min_basis = basis
                min_basis_index = i
            securities.append((s,cf))
```

```
        ctd_info = basket[min_basis_index]
        ctd_bond,ctd_cf = securities[min_basis_index]
        ctd_price = ctd_info[2]
        print("%-30s = %lf" % ("Minimum Basis", min_basis))
        print("%-30s = %lf" % ("Conversion Factor", ctd_cf))
        print("%-30s = %lf" % ("Coupon", ctd_info[0]))
        print("%-30s = %s" % ("Maturity", ctd_info[1]))
        print("%-30s = %lf" % ("Price", ctd_info[2]))

Out[8]: Minimum Basis                  = 0.450601
        Conversion Factor              = 0.791830
        Coupon                         = 2.125000
        Maturity                       = June 30th, 2022
        Price                          = 101.062500
```

The basis is the loss for a notional of 100 that the seller accrues to close this contract. For a single futures contract (which has a 100000 notional), there is a loss of 450.60.

NOTE: You will need my pull request[1] to execute the `FixedRateBondForward` class since it is not exposed in SWIG at the moment.

```
In [9]: futures_maturity_date = Date(21,12,2015)
        futures = FixedRateBondForward(calc_date, futures_maturity_date,
                                       Position.Long, 0.0, settlement_days,
                                       day_count, calendar, bussiness_convention,
                                       ctd_bond, yield_curve_handle, yield_curve_handle)
```

The valuation of the futures contract and the underlying is shown below:

```
In [10]: model_futures_price = futures.cleanForwardPrice()/ctd_cf
         implied_yield = futures.impliedYield(ctd_price/ctd_cf, futures_price,
                                              calc_date, Compounded, day_count).rate()
         z_spread = BondFunctions.zSpread(ctd_bond, ctd_price, yield_curve,
                                          day_count, Compounded, Semiannual,
                                          calc_date)
         ytm = BondFunctions.bondYield(ctd_bond, ctd_price, day_count,
                                       Compounded, Semiannual, calc_date)

         print("%-30s = %lf" % ("Model Futures Price", model_futures_price))
         print("%-30s = %lf" % ("Market Futures Price", futures_price))
         print("%-30s = %lf" % ("Model Adjustment", model_futures_price-futures_price))
         print("%-30s = %2.3f%%" % ("Implied Yield", implied_yield*100))
         print("%-30s = %2.1fbps" % ("Forward Z-Spread", z_spread*10000))
         print("%-30s = %2.3f%%" % ("Forward YTM ", ytm*100))
```

[1] https://github.com/lballabio/quantlib-old/pull/370

```
Out[10]: Model Futures Price            = 127.610365
         Market Futures Price           = 127.062500
         Model Adjustment               = 0.547865
         Implied Yield                  = -7.473%
         Forward Z-Spread               = 1.6bps
         Forward YTM                    = 1.952%
```

References

[1] Understanding Treasury Futures[2], CME Group PDF

Conclusion

In this chapter, we looked into understanding and valuing treasury futures contracts. We used the QuantLib `FixedRateBondForward` class in order to model the futures contract. But, we also made a cheapest to deliver calculation to figure out the deliverable.

[2] https://www.cmegroup.com/education/files/understanding-treasury-futures.pdf

35. Mischievous pricing conventions

(Based on a question[1] by *Stack Exchange* user ducky. Thanks!)

```
In [1]: from QuantLib import *
        import pandas as pd
```

The case of the bond off par

Like our user, I'll instantiate a four-years floating-rate bond with three-months coupons. It's being issued on the evaluation date, January 5th 2010, and for simplicity I won't use any settlement days or holidays:

```
In [2]: today = Date(5, January, 2010)
        Settings.instance().evaluationDate = today

        discounting_curve = RelinkableYieldTermStructureHandle()
        forecasting_curve = RelinkableYieldTermStructureHandle()

        index = USDLibor(Period(3, Months), forecasting_curve)

        settlement_days = 0
        calendar = NullCalendar()

        face_amount = 100.0
        schedule = Schedule(today, today + Period(4, Years),
                            Period(3, Months), calendar,
                            Unadjusted, Unadjusted,
                            DateGeneration.Forward, False)

        bond = FloatingRateBond(settlement_days,
                                face_amount,
                                schedule,
                                index,
                                Thirty360(),
                                Unadjusted,
                                fixingDays = 0)

        bond.setPricingEngine(DiscountingBondEngine(discounting_curve))
```

To price it, we use a flat 10% quarterly rate for both forecasting and discounting...

[1] http://stackoverflow.com/questions/15273797/

```
In [3]: flat_rate = FlatForward(today, 0.10, Thirty360(), Compounded, Quarterly)
        forecasting_curve.linkTo(flat_rate)
        discounting_curve.linkTo(flat_rate)
```

...so we expect the bond to be at par. Is it?

```
In [4]: print(bond.cleanPrice())

Out[4]: 99.5433545426823
```

Hmm.

What is happening here?

We have mismatched a few conventions. The ones with the largest effect are the day-count conventions used for the curve and the index. Here they are:

```
In [5]: print(flat_rate.dayCounter())
        print(as_coupon(bond.cashflows()[0]).dayCounter())
        print(index.dayCounter())

Out[5]: 30/360 (Bond Basis) day counter
        30/360 (Bond Basis) day counter
        Actual/360 day counter
```

Thus, the coupons accrue for the expected time (given by their day-count convention); however, the rates are not the expected 10%. They are calculated from discount factors given by the curve according to its 30/360 convention and recombined by the index according to its Actual/360 convention, which doesn't end well.

```
In [6]: coupons = [ as_coupon(c) for c in bond.cashflows()[:-1] ]
        pd.DataFrame([(c.date(), c.rate(), c.accrualPeriod())
                      for c in coupons ],
                     columns=('Date', 'Rate', 'Accrual period'),
                     index=range(1,len(coupons)+1))

Out[6]:
```

	Date	Rate	Accrual period
1	April 5th, 2010	0.100014	0.25
2	July 5th, 2010	0.098901	0.25
3	October 5th, 2010	0.097826	0.25
4	January 5th, 2011	0.097826	0.25
5	April 5th, 2011	0.100000	0.25
6	July 5th, 2011	0.098901	0.25
7	October 5th, 2011	0.097826	0.25
8	January 5th, 2012	0.097899	0.25
9	April 5th, 2012	0.098952	0.25
10	July 5th, 2012	0.098849	0.25
11	October 5th, 2012	0.097826	0.25
12	January 5th, 2013	0.097826	0.25
13	April 5th, 2013	0.100014	0.25
14	July 5th, 2013	0.098901	0.25
15	October 5th, 2013	0.097826	0.25
16	January 5th, 2014	0.097826	0.25

The importance of being consistent

In order to reproduce the textbook value, we have to reconcile the different conventions (which are, well, conveniently glossed over in textbooks). The correct one to choose depends on the terms and conditions of the bond; it is likely to be the Actual/360 convention used by the USD libor, so we'll pass it to both the bond and the curve:

```
In [7]: bond = FloatingRateBond(settlement_days,
                                face_amount,
                                schedule,
                                index,
                                Actual360(),
                                Unadjusted,
                                fixingDays = 0)

        bond.setPricingEngine(DiscountingBondEngine(discounting_curve))

In [8]: flat_rate_2 = FlatForward(today, 0.10, Actual360(), Compounded, Quarterly)
        forecasting_curve.linkTo(flat_rate_2)
        discounting_curve.linkTo(flat_rate_2)

In [9]: print(bond.cleanPrice())

Out[9]: 100.00117521248728
```

There's still a small discrepancy, which is likely due to date adjustments in the underlying USD libor fixings. The coupon rates are much better overall, so we seem to be on the right track.

```
In [10]: coupons = [ as_coupon(c) for c in bond.cashflows()[:-1] ]
         pd.DataFrame([(c.date(), c.rate(), c.accrualPeriod())
                       for c in coupons ],
                      columns=('Date', 'Rate', 'Accrual period'),
                      index=range(1,len(coupons)+1))
```

Out[10]:

	Date	Rate	Accrual period
1	April 5th, 2010	0.100014	0.250000
2	July 5th, 2010	0.100014	0.252778
3	October 5th, 2010	0.100028	0.255556
4	January 5th, 2011	0.100028	0.255556
5	April 5th, 2011	0.100000	0.250000
6	July 5th, 2011	0.100014	0.252778
7	October 5th, 2011	0.100028	0.255556
8	January 5th, 2012	0.100055	0.255556
9	April 5th, 2012	0.100041	0.252778
10	July 5th, 2012	0.099986	0.252778
11	October 5th, 2012	0.100028	0.255556
12	January 5th, 2013	0.100028	0.255556
13	April 5th, 2013	0.100014	0.250000
14	July 5th, 2013	0.100014	0.252778
15	October 5th, 2013	0.100028	0.255556
16	January 5th, 2014	0.100028	0.255556

To get a (theoretical) par bond, we can use a custom index whose conventions match exactly those of the bond we wanted to use: no fixing days, 30/360 day-count convention, and no holidays. We'll use a curve with the same day-count convention, too.

```
In [11]: index = IborIndex('Mock Libor', Period(3, Months), 0, USDCurrency(),
                           NullCalendar(), Unadjusted, False, Thirty360(),
                           forecasting_curve)

         bond = FloatingRateBond(settlement_days,
                                 face_amount,
                                 schedule,
                                 index,
                                 Thirty360(),
                                 Unadjusted,
                                 fixingDays = 0)

         bond.setPricingEngine(DiscountingBondEngine(discounting_curve))

In [12]: forecasting_curve.linkTo(flat_rate)
         discounting_curve.linkTo(flat_rate)
```

And now, we finally hit the jackpot:

```
In [13]: print(bond.cleanPrice())

Out[13]: 100.00000000000001

In [14]: coupons = [ as_coupon(c) for c in bond.cashflows()[:-1] ]
         pd.DataFrame([(c.date(), c.rate(), c.accrualPeriod())
                       for c in coupons ],
                      columns=('Date', 'Rate', 'Accrual period'),
                      index=range(1,len(coupons)+1))

Out[14]:
```

	Date	Rate	Accrual period
1	April 5th, 2010	0.1	0.25
2	July 5th, 2010	0.1	0.25
3	October 5th, 2010	0.1	0.25
4	January 5th, 2011	0.1	0.25
5	April 5th, 2011	0.1	0.25
6	July 5th, 2011	0.1	0.25
7	October 5th, 2011	0.1	0.25
8	January 5th, 2012	0.1	0.25
9	April 5th, 2012	0.1	0.25
10	July 5th, 2012	0.1	0.25
11	October 5th, 2012	0.1	0.25
12	January 5th, 2013	0.1	0.25
13	April 5th, 2013	0.1	0.25
14	July 5th, 2013	0.1	0.25
15	October 5th, 2013	0.1	0.25
16	January 5th, 2014	0.1	0.25

36. More mischievous conventions

(Based on a question[1] by *Stack Exchange* user nickos556. Thanks!)

```
In [1]: from QuantLib import *
        from pandas import DataFrame
```

The case of the two slightly different prices

nickos556 instantiated a fixed-rate bond with semiannual payments and tried to deduce its price from a given yield:

```
In [2]: today = Date(27, January, 2011)
        Settings.instance().evaluationDate = today
```

```
In [3]: issueDate = Date(28, January, 2011)
        maturity = Date(31, August, 2020)
        schedule = Schedule(issueDate, maturity, Period(Semiannual),
                            UnitedStates(UnitedStates.GovernmentBond),
                            Unadjusted, Unadjusted,
                            DateGeneration.Backward, False)

        bond = FixedRateBond(1, 100.0, schedule,
                             [0.03625],
                             ActualActual(ActualActual.Bond),
                             Unadjusted,
                             100.0)
```

This can be done either by passing the yield directly...

```
In [4]: bond_yield = 0.034921

        P1 = bond.dirtyPrice(bond_yield, bond.dayCounter(), Compounded, Semiannual)
```

...or by first setting an engine that uses a corresponding flat term structure.

[1] http://quant.stackexchange.com/questions/12707/

```
In [5]: flat_curve = FlatForward(bond.settlementDate(), bond_yield,
                                 ActualActual(ActualActual.Bond),
                                 Compounded, Semiannual)

        engine = DiscountingBondEngine(YieldTermStructureHandle(flat_curve))
        bond.setPricingEngine(engine)
        P2 = bond.dirtyPrice()
```

Surprisingly, the results were different:

```
In [6]: DataFrame([(P1,P2)], columns=['with yield', 'with curve'], index=[''])
```

Out[6]:

	with yield	with curve
	101.076816	101.079986

What happened?

Mischievous conventions again. The bond uses the Actual/Actual(Bond) convention, which has a requirement: in the case of short or long coupons, we also need to pass a reference start and end date that determine the regular underlying period. Case in point: this bond has a short first coupon.

```
In [7]: DataFrame([(as_coupon(c).accrualStartDate(), as_coupon(c).accrualEndDate())
                   for c in bond.cashflows()[:-1]],
                  columns = ('start date','end date'),
                  index = range(1, len(bond.cashflows())))
```

Out[7]:

	start date	end date
1	January 28th, 2011	February 28th, 2011
2	February 28th, 2011	August 31st, 2011
3	August 31st, 2011	February 29th, 2012
4	February 29th, 2012	August 31st, 2012
5	August 31st, 2012	February 28th, 2013
6	February 28th, 2013	August 31st, 2013
7	August 31st, 2013	February 28th, 2014
8	February 28th, 2014	August 31st, 2014
9	August 31st, 2014	February 28th, 2015
10	February 28th, 2015	August 31st, 2015
11	August 31st, 2015	February 29th, 2016
12	February 29th, 2016	August 31st, 2016
13	August 31st, 2016	February 28th, 2017

	start date	end date
14	February 28th, 2017	August 31st, 2017
15	August 31st, 2017	February 28th, 2018
16	February 28th, 2018	August 31st, 2018
17	August 31st, 2018	February 28th, 2019
18	February 28th, 2019	August 31st, 2019
19	August 31st, 2019	February 29th, 2020
20	February 29th, 2020	August 31st, 2020

The accrual time for the coupon, starting January 27th 2011 and ending February 28th 2011, must be calculated as:

```
In [8]: dayCounter = ActualActual(ActualActual.Bond)

        T = dayCounter.yearFraction(Date(28, January, 2011), Date(28, February, 2011),
                                    Date(28,August,2010),  Date(28,February,2011))
        print(T)
```

```
Out[8]: 0.08423913043478261
```

If the coupon were annual, it would be:

```
In [9]: print(dayCounter.yearFraction(Date(28, January, 2011), Date(28, February, 2011),
                                      Date(28,February,2010),  Date(28,February,2011)))
```

```
Out[9]: 0.08493150684931507
```

The corresponding discount factor given the yield is as follows:

```
In [10]: y = InterestRate(bond_yield, dayCounter, Compounded, Semiannual)
         print(y.discountFactor(T))
```

```
Out[10]: 0.997087920498809
```

Yield-based calculation

The yield-based calculation uses the above to discount the first coupon, and combines it with discount factors corresponding to the regular coupon times to discount the others. We can write down the full computation:

```
In [11]: data = []
         for i, c in enumerate(bond.cashflows()[:-1]):
             c = as_coupon(c)
             A = c.amount()
             T = c.accrualPeriod()
             D = y.discountFactor(T)
             D_cumulative = D if i == 0 else D * data[-1][3]
             A_discounted = A*D_cumulative
             data.append((A,T,D,D_cumulative,A_discounted))
         data.append((100,'','',D_cumulative,100*D_cumulative))
         data = DataFrame(data,
                          columns = ('amount', 'T', 'discount', 'discount (cum.)', 'amount (di\
sc.)'),
                          index = ['']*len(data))
         data
```

Out[11]:

amount	T	discount	discount (cum.)	amount (disc.)
0.305367	0.0842391	0.997088	0.997088	0.304478
1.812500	0.5	0.982839	0.979977	1.776208
1.812500	0.5	0.982839	0.963160	1.745727
1.812500	0.5	0.982839	0.946631	1.715769
1.812500	0.5	0.982839	0.930386	1.686325
1.812500	0.5	0.982839	0.914420	1.657386
1.812500	0.5	0.982839	0.898728	1.628944
1.812500	0.5	0.982839	0.883305	1.600990
1.812500	0.5	0.982839	0.868146	1.573515
1.812500	0.5	0.982839	0.853248	1.546513
1.812500	0.5	0.982839	0.838606	1.519973
1.812500	0.5	0.982839	0.824215	1.493889
1.812500	0.5	0.982839	0.810070	1.468253
1.812500	0.5	0.982839	0.796169	1.443056
1.812500	0.5	0.982839	0.782506	1.418292
1.812500	0.5	0.982839	0.769077	1.393953
1.812500	0.5	0.982839	0.755879	1.370031
1.812500	0.5	0.982839	0.742908	1.346521
1.812500	0.5	0.982839	0.730159	1.323413
1.812500	0.5	0.982839	0.717629	1.300702
100.000000			0.717629	71.762879

The bond price is the sum of the discounted amounts...

```
In [12]: print(sum(data['amount (disc.)']))

Out[12]: 101.07681646503603
```

...and, not surprisingly, equals the yield-based bond price.

```
In [13]: print(bond.dirtyPrice(bond_yield, bond.dayCounter(), Compounded, Semiannual))

Out[13]: 101.07681646503603
```

Curve-based calculation

Long story short: the bond engine gets the first discount wrong. Given the curve interface, all it can do is ask for the discounts at the coupon dates, as follows:

```
In [14]: data = []
         for c in bond.cashflows()[:-1]:
             A = c.amount()
             D = flat_curve.discount(c.date())
             A_discounted = A*D
             data.append((A,D,A_discounted))
         data.append((100.0,D,100.0*D))
         data = DataFrame(data,
                          columns = ('amount', 'discount', 'amount (disc.)'),
                          index = ['']*len(data))
         data

Out[14]:
```

amount	discount	amount (disc.)
0.305367	0.997119	0.304487
1.812500	0.980008	1.776264
1.812500	0.963190	1.745782
1.812500	0.946661	1.715823
1.812500	0.930415	1.686378
1.812500	0.914449	1.657438
1.812500	0.898756	1.628995
1.812500	0.883332	1.601040
1.812500	0.868174	1.573565
1.812500	0.853275	1.546561
1.812500	0.838632	1.520021
1.812500	0.824240	1.493936
1.812500	0.810096	1.468299
1.812500	0.796194	1.443101

amount	discount	amount (disc.)
1.812500	0.782530	1.418336
1.812500	0.769102	1.393997
1.812500	0.755903	1.370074
1.812500	0.742931	1.346563
1.812500	0.730182	1.323455
1.812500	0.717651	1.300743
100.000000	0.717651	71.765129

The result equals the curve-based price.

```
In [15]: print(sum(data['amount (disc.)']))

Out[15]: 101.0799861183387

In [16]: print(bond.dirtyPrice())

Out[16]: 101.0799861183387
```

The problem is that the first call to the `discount` method, that is,

```
In [17]: flat_curve.discount(Date(28, February, 2011))

Out[17]: 0.9971191880350325
```

results in a call to:

```
In [18]: print(dayCounter.yearFraction(Date(28, January, 2011), Date(28, February, 2011)))

Out[18]: 0.08333333333333333
```

Compare this to the correct one:

```
In [19]: T = dayCounter.yearFraction(Date(28, January, 2011), Date(28, February, 2011),
                                     Date(28,August,2010),  Date(28,February,2011))
         print(T)

Out[19]: 0.08423913043478261
```

Does this account for the difference in price? Yes, it does. The two prices can be reconciled as follows:

```
In [20]: P_y = bond.dirtyPrice(bond_yield, bond.dayCounter(), Compounded, Semiannual)
         D_y = y.discountFactor(T)

         P_c = bond.dirtyPrice()
         D_c = flat_curve.discount(Date(28, February, 2011))

         print(P_y)
         print(P_c*(D_y/D_c))

Out[20]: 101.07681646503603
         101.0768164650361
```

Appendix

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]: from QuantLib import *
```

This line imports the `QuantLib` module and adds the classes and functions it contains to the global namespace. The C++ equivalent would be:

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

using namespace QuantLib;
```

Of course, the above is for illustration purposes. In production code, you're not forced (or even advised) to use the `using` directory; you can keep the names in their namespace and qualify them when you use them. It's also possible to include more specific headers, instead of the global `quantlib.hpp`.

```
In [2]: today = Date(7, March, 2014)
        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 first dot by a double colon. 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:

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

Next:

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

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

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

```
In [4]: u = SimpleQuote(100.0)
        r = SimpleQuote(0.01)
        sigma = 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 = FlatForward(0, TARGET(),
                                    QuoteHandle(r), Actual360())
        volatility = BlackConstantVol(0, TARGET(),
                                      QuoteHandle(sigma), 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 = BlackScholesProcess(QuoteHandle(u),
                                      YieldTermStructureHandle(riskFreeCurve),
                                      BlackVolTermStructureHandle(volatility))
```

turns into

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

and

```
In [7]: engine = 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())
```

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)
```

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++.