A QuantLib Guide

Luigi Ballabio

Also by the same author:

Implementing QuantLib
available in paperback format at your Amazon store
available in electronic format at www.leanpub.com/implementingquantlib

QuantLib Python Cookbook (with G. Balaraman)
available in electronic format at www.leanpub.com/quantlibpythoncookbook

Cover image by Tamas Tuzes-Katai on Unsplash.

A QuantLib Guide

An introduction to the usage
of the QuantLib library

Luigi Ballabio

Table of contents

About this book

This book tries to fill a gap in the documentation available for QuantLib. It's a gap that, unfortunately, I contributed to create; my book *Implementing QuantLib* is aimed more at developers wanting to extend the library and less at users, and the material I contributed to *QuantLib Python Cookbook* is a decent catalog of use cases but lacks a bit of direction.

This guide includes some updated material from those two books, as well as some later material I published on my blog or on Wilmott Magazine; the idea is to collect, update and present it all in a sequence that makes it suitable as a tutorial for someone approaching QuantLib or trying to get more fluent in its usage.

It's a work in progress; please let me know if it goes in the right direction. I'll be grateful for any corrections, suggestions or comments.

If you like this book, subscribe to my Substack at implementingquantlib.substack.com to get more of my content as soon as I publish it. It's free.

Chapter 1

A taste of QuantLib

(Originally published as an article in Wilmott Magazine, January 2023.)

Why don't I skip the usual introductions and show you some code instead? Let's see if you find something interesting in it. I'll try to keep this a quick tour and show the forest rather than the trees; so I'll gloss over, or completely ignore, a lot of details. Here we go.

```
import QuantLib as ql
```

Dates and calendars and conventions, oh my

After setting the evaluation date to be used, the first few lines create a sequence of regular dates between an initial and a final one and store them in a `Schedule` object. As you can see, we can specify parameters such as their frequency, what convention to use to roll a date when it falls on a holiday, and the calendar to use to determine holidays. We also specify an explicit first date; we'll use the schedule to specify coupon dates for a bond, and this allows us to have a long first coupon from August 2020 to May 2021, followed by regular semiannual coupons resetting in November and May until 2031.

```
value_date = ql.Date(8, ql.December, 2022)
ql.Settings.instance().evaluationDate = value_date

schedule = ql.MakeSchedule(
    effectiveDate=ql.Date(26, ql.August, 2020),
    terminationDate=ql.Date(26, ql.May, 2031),
    firstDate=ql.Date(26, ql.May, 2021),
    frequency=ql.Semiannual,
```

```
    calendar=ql.TARGET(),
    convention=ql.Following,
    backwards=True,
)

for d in schedule:
    print(d)
```

```
August 26th, 2020
May 26th, 2021
November 26th, 2021
May 26th, 2022
November 28th, 2022
May 26th, 2023
November 27th, 2023
May 27th, 2024
November 26th, 2024
May 26th, 2025
November 26th, 2025
May 26th, 2026
November 26th, 2026
May 26th, 2027
November 26th, 2027
May 26th, 2028
November 27th, 2028
May 28th, 2029
November 26th, 2029
May 27th, 2030
November 26th, 2030
May 26th, 2031
```

Some simple bond calculations

The next few lines create a fixed-rate bond. Besides the schedule that we just defined (used, of course, for the coupon dates), its constructor takes a few more parameters: the number of settlement days, the face amount, the coupon rate, and the day-count convention to use for accrual. Again, I'm not showing all the possibilities here. For instance, the list of coupon rates might contain multiple values, resulting in a step-up bond.

Once instantiated, the bond can start to provide useful figures. Here I'm asking it for its price given a quoted yield and for the amount accrued so far; that is, up to the evaluation date

set at the beginning of the program. We could get other results, such as its duration or the amount accrued at some other date; or we could extract its coupons and ask each of them for their amounts, rates, accrual times and so on.

```
settlement_days = 3
face_amount = 1000000.0
coupons = [0.02]
day_counter = ql.Thirty360(ql.Thirty360.BondBasis)

bond = ql.FixedRateBond(
    settlement_days,
    face_amount,
    schedule,
    coupons,
    day_counter,
)

bond_yield = 0.025
price = bond.cleanPrice(bond_yield, day_counter, ql.Compounded, ql.Annual)
accrual = bond.accruedAmount()

print(f"Price:   {price:.6}")
print(f"Accrual: {accrual:.4}")
```

```
Price:   96.3256
Accrual: 0.08333
```

Discount curves

Let's say that you have a discount curve available for the bond issuer. In this listing, I'm assuming that it's specified as a set of zero rates corresponding to a set of dates, and for the sake of illustration I'm hard-coding them; in a real use case, they would come from a database or some API. The `ZeroCurve` class turns them into a curve object that can return rates or discount factors at any date in between by using linear interpolation. Other classes can be used if, for instance, the curve is specified as a set of discount factors, or if you want to bootstrap or fit the curve based on a set of market quotes.

Whatever the curve, it can be used to calculate a theoretical price for the bond; instead of relying on a quoted yield, we can create a pricing engine that first uses the curve to discount each bond coupon to the settlement date and then accumulates the results. Once created, we pass the engine to the bond and ask it for its price.

```
curve_nodes = [
    value_date,
    value_date + ql.Period(1, ql.Years),
    value_date + ql.Period(2, ql.Years),
    value_date + ql.Period(5, ql.Years),
    value_date + ql.Period(10, ql.Years),
    value_date + ql.Period(15, ql.Years),
]
curve_rates = [
    0.015,
    0.015,
    0.018,
    0.022,
    0.025,
    0.028,
]
issuer_curve = ql.ZeroCurve(curve_nodes, curve_rates, ql.Actual360())

bond.setPricingEngine(
    ql.DiscountingBondEngine(ql.YieldTermStructureHandle(issuer_curve))
)

print(f"Price: {bond.cleanPrice():.6}")
```

```
Price: 96.6641
```

Pricing at a spread over a curve

For a variation on the last calculation, we can also price the bond at a z-spread over a curve; for instance, a government curve, which again I'm assuming to be tabulated on a set of dates. To account for the spread, I used the `ZeroSpreadedTermStructure` class, which adds it to the government curve and creates a new curve; the latter can be passed to a new pricing engine.

The spread (modeled as an instance of the `SimpleQuote` class) is initially set to zero; but it can be set afterwards to any other value, and the bond price will react accordingly. Adding a constant spread is not the only way we can modify an existing curve: other classes provide the means to model, say, flattening or steepening scenarios.

```
curve_rates_2 = [
    -0.005,
    -0.005,
```

```
    0.001,
    0.004,
    0.009,
    0.012,
]
govies_curve = ql.ZeroCurve(curve_nodes, curve_rates_2, ql.Actual360())

spread = ql.SimpleQuote(0.0)
discount_curve = ql.ZeroSpreadedTermStructure(
    ql.YieldTermStructureHandle(govies_curve), ql.QuoteHandle(spread)
)

bond.setPricingEngine(
    ql.DiscountingBondEngine(ql.YieldTermStructureHandle(discount_curve))
)

print(f"Price: {bond.cleanPrice():.6}")

spread.setValue(0.01)

print(f"Price: {bond.cleanPrice():.6}")
```

```
Price: 110.34
Price: 101.913
```

Finding the z-spread given a price

Finally, what if you have a price—for instance, the one we calculated given the quoted yield—and you want to find the corresponding z-spread over the government curve? The objects we already created can be reused to create a function (`discrepancy` in the code) that takes a value, sets it to the spread, and returns the difference between the resulting price and the target price. Once created, the function can be passed to a one-dimensional solver that finds its root; that is, the value of the spread for which the discrepancy is null and the calculated price equals the desired one.

```
def discrepancy(s):
    spread.setValue(s)
    return bond.cleanPrice() - price


solver = ql.Brent()
```

```
accuracy = 1e-5
guess = 0.015
step = 1e-4

spread_over_govies = solver.solve(discrepancy, accuracy, guess, step)

print(f"Spread: {spread_over_govies:.4}")
```

```
Spread: 0.01712
```

Not only Python

All the features showcased above are, of course, available in the underlying C++ library. There are a few differences, due in part to the different syntax and in part to the need in C++ for memory management which makes it necessary to use smart pointers, hidden in Python by the wrapper code. For comparison, here is the original C++ code from which the Python code above was translated (I also have a note on smart pointers, but it can wait until after the code.)

```
#include <ql/instruments/bonds/fixedratebond.hpp>
#include <ql/math/solvers1d/brent.hpp>
#include <ql/pricingengines/bond/discountingbondengine.hpp>
#include <ql/quotes/simplequote.hpp>
#include <ql/settings.hpp>
#include <ql/termstructures/yield/zerocurve.hpp>
#include <ql/termstructures/yield/zerospreadedtermstructure.hpp>
#include <ql/time/calendars/target.hpp>
#include <ql/time/daycounters/actual360.hpp>
#include <ql/time/daycounters/thirty360.hpp>
#include <ql/time/schedule.hpp>
#include <iostream>

int main() {

    using namespace QuantLib;

    auto valueDate = Date(8, December, 2022);
    Settings::instance().evaluationDate() = valueDate;

    // Dates and calendars and conventions, oh my
```

```
    Schedule schedule =
        MakeSchedule()
            .from(Date(26, August, 2020))
            .to(Date(26, May, 2031))
            .withFirstDate(Date(26, May, 2021))
            .withFrequency(Semiannual)
            .withCalendar(TARGET())
            .withConvention(Following)
            .backwards();

    for (auto d : schedule) {
        std::cout << d << "\n";
    }
    std::cout << std::endl;

    // Some simple bond calculations

    unsigned settlementDays = 3;
    double faceAmount = 1000000.0;
    std::vector<Rate> coupons = {0.02};
    auto dayCounter = Thirty360(Thirty360::BondBasis);

    auto bond =
        FixedRateBond(settlementDays, faceAmount, schedule,
                      coupons, dayCounter);

    Rate yield = 0.025;
    double price = bond.cleanPrice(yield, dayCounter,
                                   Compounded, Annual);
    double accrual = bond.accruedAmount();

    std::cout << price << "\n"
              << accrual << std::endl;

    // Discount curves

    std::vector<Date> curveNodes = {valueDate,
                                    valueDate + 1 * Years,
                                    valueDate + 2 * Years,
                                    valueDate + 5 * Years,
                                    valueDate + 10 * Years,
```

```
                                   valueDate + 15 * Years};
std::vector<Rate> curveRates = {0.015, 0.015, 0.018,
                                   0.022, 0.025, 0.028};
auto issuerCurve =
    ext::make_shared<InterpolatedZeroCurve<Linear>>(
        curveNodes, curveRates, Actual360());

bond.setPricingEngine(
    ext::make_shared<DiscountingBondEngine>(
        Handle<YieldTermStructure>(issuerCurve)));
std::cout << bond.cleanPrice() << std::endl;

// Pricing at a spread over a curve

std::vector<Rate> curveRates2 = {-0.005, -0.005, 0.001,
                                    0.004,  0.009,  0.012};
auto goviesCurve =
    ext::make_shared<InterpolatedZeroCurve<Linear>>(
        curveNodes, curveRates2, Actual360());

auto spread = ext::make_shared<SimpleQuote>(0.0);
auto discountCurve =
    ext::make_shared<ZeroSpreadedTermStructure>(
        Handle<YieldTermStructure>(goviesCurve),
        Handle<Quote>(spread));

bond.setPricingEngine(
    ext::make_shared<DiscountingBondEngine>(
        Handle<YieldTermStructure>(discountCurve)));
std::cout << bond.cleanPrice() << std::endl;

spread->setValue(0.01);
std::cout << bond.cleanPrice() << std::endl;

// Finding the z-spread given a price

auto discrepancy = [&spread, &bond, price](Spread s) {
    spread->setValue(s);
    return bond.cleanPrice() - price;
};
```

```
    auto solver = Brent();
    double accuracy = 1e-5, guess = 0.015, step = 1e-4;
    Spread spreadOverGovies =
        solver.solve(discrepancy, accuracy, guess, step);

    std::cout << spreadOverGovies << std::endl;
}
```

Smart pointers and other standard classes

And now, the note on smart pointers I promised before. They were added the the C++ standard in 2011, so I won't bore you by explaining what they are and why they are used; I assume you know all that. What you might be wondering, however, is why they seem to be in namespace `ext` in the code above.

In short: we have a compilation flag that allows us to define `ext::shared_ptr` as either `boost::shared_ptr` (the current default) or `std::shared_ptr`; the same flag also controls related functions such as `make_shared` or `dynamic_pointer_cast`. We did the same for other classes such as `std::function` or `std::tuple`, each with its own compilation flag.

This gives us a migration path from using the Boost classes (the only alternative for the first decade of QuantLib) to using the ones in the C++ standard, without breaking other people's code in the switch.

Boost is still the default for some of these classes; beside the smart pointers, also `any` and `optional`. A while ago, we changed the default for `function` and `tuple`, and recently we removed the switch so that the `std` implementation is always used. The long-term plan is to do the same for the other classes.

For `shared_ptr`, however, the decision will be a bit more complicated. There's a significant difference in behavior between `boost::shared_ptr` and `std::shared_ptr`; the former can be configured so that trying to access a null `shared_ptr` raises an exception, while the latter doesn't check for null pointers and lets you access them and crash the program.

Therefore, it's possible that we'll keep `boost::shared_ptr` around as an alternative for quite a while, for instance when generating the Python wrappers for QuantLib. Unlike the C++ community, Python programmers tend to frown upon segmentation faults.

Part I

From dates to bonds

This part goes into more detail about the classes we saw in the first example.

In the next few chapters, we will start from date-related classes such as dates, calendars and schedules and cover topics like date calculations, custom holidays and various schedule-generation rules. Afterwards, we will build up from that, finally instantiating a couple of simple bonds and inspecting their coupons.

Chapter 2

Dates and periods

The part of the library that deals with dates and calendars is relatively self-contained; in fact, I've been told that some people only use that part of the library. If you're an R user, there is even a package called qlcal that makes it available in R with an idiomatic syntax (kudos, Dirk!)

```
import QuantLib as ql
```

Depending on where you live, the order of the arguments in the `Date` constructor might seem unfamiliar; which is why, in C++, the month needs to be passed as an enumeration and not as a number. In that language, this makes it impossible to get confused and initialize the dates in the wrong way—well, it's still possible to get confused, but if you try to pass month, day, year as in the U.S. format you'll get a compile-time error, and if you pass year, month, day as in the ISO format you'll get a run-time error, so at least you won't do the wrong thing without noticing.

Unfortunately, we don't have strong-typed enums in Python, so you'll have to remember: day, month, year.

```
d = ql.Date(11, ql.May, 2021)
d
```

```
Date(11,5,2021)
```

Simple algebra

Some simple date calculations can be done by means of the `Date` and `Period` classes. They do what you would expect, without taking any holidays into account.

```
d + ql.Period(4, ql.Days)
```

```
Date(15,5,2021)
```

```
d + ql.Period(1, ql.Weeks)
```

```
Date(18,5,2021)
```

```
d + ql.Period(3, ql.Months)
```

```
Date(11,8,2021)
```

Some corner cases are treated as per market conventions:

```
ql.Date(31, 1, 2021) + ql.Period(1, ql.Months)
```

```
Date(28,2,2021)
```

```
ql.Date(29, 2, 2020) + ql.Period(1, ql.Years)
```

```
Date(28,2,2021)
```

Difference between dates gives you the number of days between them…

```
d2 = d + ql.Period(3, ql.Months)
```

```
d2 - d
```

```
92
```

…and adding an int to a date is shorthand for adding that number of days.

```
d + 5
```

```
Date(16,5,2021)
```

The same goes for incrementing:

```
d += ql.Period(2, ql.Months)
d
```

```
Date(11,7,2021)
```

```
d += 1
d
```

```
Date(12,7,2021)
```

```
d == d2
```

```
False
```

```
d < d2
```

```
True
```

There's some limited algebra for periods, too. They can be compared when the result is unambiguous…

```
p1 = ql.Period(3, ql.Months)
p2 = ql.Period(8, ql.Weeks)
```

```
p1 < p2
```

```
False
```

…but sometimes the comparison can't be decided.

```
p1 = ql.Period(1, ql.Months)
p2 = ql.Period(30, ql.Days)
```

```
try:
    p1 < p2
except Exception as e:
    print(f"{type(e).__name__}: {e}")
```

```
RuntimeError: undecidable comparison between 1M and 30D
```

The same goes for other operators like addition and subtraction.

```
ql.Period(3, ql.Months) + ql.Period(2, ql.Months)
```

```
Period("5M")
```

```
ql.Period(1, ql.Years) - ql.Period(8, ql.Months)
```

```
Period("4M")
```

```
try:
    ql.Period(3, ql.Weeks) + ql.Period(2, ql.Months)
except Exception as e:
    print(f"Error: {e}")
```

```
Error: impossible addition between 3W and 2M
```

Parsing

Just a couple of examples, in case you need to read dates from a file:

```
ql.DateParser.parseISO("2020-04-29")
```

```
Date(29,4,2020)
```

```
ql.DateParser.parseFormatted("04/29/20", "%m/%d/%y")
```

```
Date(29,4,2020)
```

Conversions to and from native Python dates

Also, in case you used pandas or some other module to read data, you might have dates as instances of Python's `date` class. You can convert them to QuantLib dates with the static `from_date` method...

```
import datetime

d = datetime.date(2021, 5, 11)

ql.Date.from_date(d)
```

```
Date(11,5,2021)
```

...and you can do the opposite by means of the `to_date` method:

```
d = ql.Date(11, 5, 2021)
d.to_date()
```

```
datetime.date(2021, 5, 11)
```

Chapter 3

Calendars and holidays

(Originally published as an article in Wilmott Magazine, January 2024.)

```
import QuantLib as ql
```

In QuantLib, calendars are usually declared by country, possibly with an additional enumeration specifying different calendars for particular exchanges or purposes; for example, the second calendar declared here is the one used for the U.S. government bond market. The first calendar we're declaring is an exception to the rule; it's the super-national TARGET calendar used for Euro. All in all, the library declares a few dozen calendars plus, as we'll see later, a few classes that make it possible to build more.

```
cal1 = ql.TARGET()
cal2 = ql.UnitedStates(ql.UnitedStates.GovernmentBond)
```

Holidays, adjustments, end of month

Now, what can we do with calendars? Their basic functionality is to tell us whether a given day is a holiday or not; for instance, you can see below how the TARGET calendar tells us that May 1st, Labor Day, is a holiday, while the U.S. Government Bond calendar tells us that it's a business day; the opposite holds if we look at June 19th, which is celebrated as Juneteenth in the United States. There is no particular treatment for weekends (they are simply considered holidays) and of course they, too, can change depending on the calendar.

```
d1 = ql.Date(1, ql.May, 2023)
d2 = ql.Date(19, ql.June, 2023)
```

```
cal1.isHoliday(d1)
```

```
True
```

```
cal2.isHoliday(d1)
```

```
False
```

```
cal1.isBusinessDay(d2)
```

```
True
```

```
cal2.isBusinessDay(d2)
```

```
False
```

When passed a holiday, calendars can also adjust it to the nearest business day, where "nearest" is determined according to a given convention; for instance, the cells below write out that the first business day after May 1st 2023, according to TARGET, was May 2nd, whereas the first business day before Juneteenth 2023 was June 16th (the 17th and 18th being Saturday and Sunday). When passed a business day, the `adjust` method returns it unmodified.

```
print(cal1.adjust(d1, ql.Following))
```

```
May 2nd, 2023
```

```
print(cal2.adjust(d2, ql.Preceding))
```

```
June 16th, 2023
```

Other methods can tell us whether a given date is the last business day of the month; or, given a date, which date is the last business day of that month. For some reason, there are no corresponding methods for the first of the month.

```
cal1.isEndOfMonth(d1)
```

```
False
```

```
print(cal1.endOfMonth(d1))
```

```
May 31st, 2023
```

Jumping...

I'm not going to show how to build coupon schedules in this notebook, but we can see how calendars provide the basic building block for that task; namely, how they can advance a date by a number of months or years and get the nearest business day. The basic functionality for

jumping (either forwards or backwards) is already in the `Date` class; and the `Calendar` class combines, in its `advance` method, a jump and an adjustment.

```
print(d2 + ql.Period(2, ql.Months))
```

```
August 19th, 2023
```

```
print(cal1.adjust(d2 + ql.Period(2, ql.Months)))
```

```
August 21st, 2023
```

```
print(cal1.advance(d2, ql.Period(2, ql.Months)))
```

```
August 21st, 2023
```

...and walking

There is a case, though, in which calling the `advance` method gives a different result than increasing the date and then calling `adjust`; and that's when the period we're passing is measured in days. In this specific case, the `advance` method increases the date by the given number of business days; as we need to do, for instance, when calculating a settlement date, or a lagged payment date.

```
print(d2 + ql.Period(7, ql.Days))
```

```
June 26th, 2023
```

```
print(cal1.adjust(d2 + ql.Period(7, ql.Days)))
```

```
June 26th, 2023
```

```
print(cal1.advance(d2, ql.Period(7, ql.Days)))
```

```
June 28th, 2023
```

Unforeseen holidays

Most holidays are, as you might expect, defined as rules. Some are more or less easy to express in terms of the Western calendar; they are something like "May 1st, every year", or "January 1st, possibly moved to Monday if it falls on a weekend", or again "the last Monday of August". In time, rules can change, too: for instance, Juneteenth was declared as a holiday in 2022 and markets started closing on that day in 2023, so the corresponding rule must include that condition.

Other holidays, mostly from Asian markets, have rules which are based on a lunar calendar and are therefore harder to express. In this case, we usually code them as specific days

and update them as they are published on the market site; "we" meaning our esteemed contributors. The one exception is Easter, which is also based on a lunar calendar but can be found tabulated for the next couple of centuries and more.

Finally, some specific days can be declared one-off holidays by the relevant authority; for instance, in correspondence of a general election or, like this past year, for the death of a sovereign. Another, more idiosyncratic case? In the past years, SIFMA decided that Good Friday would only be a half-closure (and therefore a business day for the corresponding calendar) but SOFR would not be fixed on that day—making us realize that, grr, we needed a specialized SOFR calendar besides the existing ones.

So, what are you to do if a new holiday is published or declared but the next QuantLib release is three months away? Fortunately, you don't need to wait for us to update the code; you don't even need to update the code yourself and recompile the library, if you're not so inclined. Calendars provide a method, `addHoliday`, that can declare a date as a new holiday at run time; you can see it in action below. You can also see that, when you add a holiday to a calendar, all instances of the same calendar (TARGET, in this case) are affected by the change. A corresponding method `removeHoliday` is available, even though it's less commonly used.

```
d3 = ql.Date(7, ql.June, 2023)
cal1.isHoliday(d3)
```

```
False
```

```
cal1.addHoliday(d3)

cal1b = ql.TARGET()
cal1b.isHoliday(d3)
```

```
True
```

Joint calendars

Another case in which the provided calendars are not quite enough: building the schedules for some deals (or, in general, some sets of dates) requires to consider holidays from two or more calendars. In this case, the `JointCalendar` class can be used to create the required calendar from a few underlying ones; as the cells below show, it's possible to specify if we want to join the sets of their holidays (i.e., a day is considered a holiday if it is a holiday for at least one of the underlying calendars) or the sets of their business days (i.e., a day is considered a holiday if it is a holiday for all of the underlying calendars.)

```
cal3 = ql.JointCalendar(cal1, cal2, ql.JoinHolidays)
cal4 = ql.JointCalendar(cal1, cal2, ql.JoinBusinessDays)
```

```
cal3.isHoliday(d1)
```

```
True
```

```
cal4.isHoliday(d1)
```

```
False
```

```
d4 = ql.Date(25, ql.December, 2023)
cal4.isHoliday(d4)
```

```
True
```

Bespoke calendars

Finally, in a shocking turn of events, you can also choose to disregard all the calendars we're providing. I did it myself on a past project; in that case, we already had a data provider who was feeding us updated lists of holidays for a number of calendars and joint calendars. Any new holiday would go into our database before we were aware of it, freeing us from having to maintain lists of holidays to add or remove from the calendars built into the library.

For that use case, the right thing to do was to use the `BespokeCalendar` class. As you can see, each instance of the class is a blank slate to be filled with a specification of the weekends and a list of holidays. This gave us a generic way to instantiate any calendar by reading the corresponding information from the database. Also, using bespoke calendars gives you a way to create a new calendar if you're using QuantLib from another language (say, Python or C#) and you don't want to modify and recompile the underlying C++ code.

```
cal5 = ql.BespokeCalendar("my-cal")
cal5.addWeekend(ql.Saturday)
cal5.addWeekend(ql.Sunday)

holidays = [
    ql.Date(15, ql.April, 2022),
    ql.Date(18, ql.April, 2022),
    ql.Date(26, ql.December, 2022),
    ql.Date(7, ql.April, 2023),
    ql.Date(10, ql.April, 2023),
    ql.Date(1, ql.May, 2023),
    ql.Date(7, ql.June, 2023),
    ql.Date(25, ql.December, 2023),
    ql.Date(26, ql.December, 2023),
    ql.Date(1, ql.January, 2024),
```

```
]

for d in holidays:
    cal5.addHoliday(d)
```

```
cal5.isHoliday(d1)
```

```
True
```

Chapter 4

Schedules

(Originally published as an article in Wilmott Magazine, March 2024.)

In this notebook, I'll assume you already read about calendars and holidays. I'll start from there and show how to use QuantLib to generate schedules, i.e., regular sequences of dates, choosing from a number of market conventions. In turn, those can be used to create sequences of coupons; but that's something for another time.

```
import QuantLib as ql
import pandas as pd
```

A simple example

We can build a schedule with as little information as a start date, an end date, and a frequency. Here is the corresponding call:

```
s = ql.MakeSchedule(
    effectiveDate=ql.Date(11, ql.May, 2021),
    terminationDate=ql.Date(11, ql.May, 2025),
    frequency=ql.Semiannual,
)

pd.DataFrame(list(s), columns=["dates"])
```

	dates
0	May 11th, 2021

	dates
1	November 11th, 2021
2	May 11th, 2022
3	November 11th, 2022
4	May 11th, 2023
5	November 11th, 2023
6	May 11th, 2024
7	November 11th, 2024
8	May 11th, 2025

Unsurprisingly, it is a sequence of alternating May 11th and November 11th from the start date to the end date; they are both included.

When building coupons from this schedule, the understanding is that the first date in the schedule is the start of the first coupon; the second date is both the end of the first coupon and the start of the second; the third date is the end of the second coupon and the start of the third; until we get to the last date, which is the end of the last coupon.

Adjusting for holidays

The above schedule didn't make a distinction between holidays and business days. If we want holidays to be adjusted, we need to choose a calendar:

```
s = ql.MakeSchedule(
    effectiveDate=ql.Date(11, ql.May, 2021),
    terminationDate=ql.Date(11, ql.May, 2025),
    frequency=ql.Semiannual,
    calendar=ql.TARGET(),
)

pd.DataFrame(list(s), columns=["dates"])
```

	dates
0	May 11th, 2021
1	November 11th, 2021
2	May 11th, 2022
3	November 11th, 2022
4	May 11th, 2023
5	November 13th, 2023
6	May 13th, 2024

	dates
7	November 11th, 2024
8	May 12th, 2025

As you can see, a few dates are no longer the 11th of the month and were replaced with the next business day. In this case, those dates fell on Saturdays or Sundays, but of course the adjustment would also be performed if they were mid-week holidays.

For convenience, it's possible to pass a calendar and at the same time specify that dates should be unadjusted; here is the corresponding call:

```
s = ql.MakeSchedule(
    effectiveDate=ql.Date(11, ql.May, 2021),
    terminationDate=ql.Date(11, ql.May, 2025),
    frequency=ql.Semiannual,
    calendar=ql.TARGET(),
    convention=ql.Unadjusted,
)

pd.DataFrame(list(s), columns=["dates"])
```

	dates
0	May 11th, 2021
1	November 11th, 2021
2	May 11th, 2022
3	November 11th, 2022
4	May 11th, 2023
5	November 11th, 2023
6	May 11th, 2024
7	November 11th, 2024
8	May 11th, 2025

As I said, this is a convenience; when reading data from a file or a DB, it makes it unnecessary to write logic that chooses whether or not to pass a calendar.

Short and long coupons

If the start and end dates don't bracket a whole number of periods, it becomes important to specify whether the dates should be generated forwards from the start date or backwards

from the end date; the default is to generate them backwards, but it's probably better to be explicit. The corresponding calls are as follows:

```
s1 = ql.MakeSchedule(
    effectiveDate=ql.Date(11, ql.May, 2021),
    terminationDate=ql.Date(11, ql.February, 2025),
    frequency=ql.Semiannual,
    calendar=ql.TARGET(),
    forwards=True,
)

s2 = ql.MakeSchedule(
    effectiveDate=ql.Date(11, ql.May, 2021),
    terminationDate=ql.Date(11, ql.February, 2025),
    frequency=ql.Semiannual,
    calendar=ql.TARGET(),
    backwards=True,
)

pd.DataFrame(zip(s1, s2), columns=["forwards", "backwards"])
```

	forwards	backwards
0	May 11th, 2021	May 11th, 2021
1	November 11th, 2021	August 11th, 2021
2	May 11th, 2022	February 11th, 2022
3	November 11th, 2022	August 11th, 2022
4	May 11th, 2023	February 13th, 2023
5	November 13th, 2023	August 11th, 2023
6	May 13th, 2024	February 12th, 2024
7	November 11th, 2024	August 12th, 2024
8	February 11th, 2025	February 11th, 2025

In the first case, we ended up with a short last coupon; in the second, with a short first coupon. It's also possible to specify a long coupon by passing an explicit stub:

```
s = ql.MakeSchedule(
    effectiveDate=ql.Date(11, ql.February, 2021),
    terminationDate=ql.Date(11, ql.May, 2025),
    frequency=ql.Semiannual,
    firstDate=ql.Date(11, ql.November, 2021),
    calendar=ql.TARGET(),
```

```
    forwards=True,
)

pd.DataFrame(list(s), columns=["dates"])
```

	dates
0	February 11th, 2021
1	November 11th, 2021
2	May 11th, 2022
3	November 11th, 2022
4	May 11th, 2023
5	November 13th, 2023
6	May 13th, 2024
7	November 11th, 2024
8	May 12th, 2025

In case of a long last coupon, you can use the `nextToLastDate` keyword instead of `firstDate`; the two can also be used together.

End of month

When the dates are close to the end of their month, other conventions can come into play. The default behavior is to generate dates as usual; in the call

```
s = ql.MakeSchedule(
    effectiveDate=ql.Date(28, ql.February, 2019),
    terminationDate=ql.Date(28, ql.February, 2023),
    frequency=ql.Semiannual,
    calendar=ql.TARGET(),
    forwards=True,
)

pd.DataFrame(list(s), columns=["dates"])
```

	dates
0	February 28th, 2019
1	August 28th, 2019
2	February 28th, 2020
3	August 28th, 2020

	dates
4	March 1st, 2021
5	August 30th, 2021
6	February 28th, 2022
7	August 29th, 2022
8	February 28th, 2023

you can see that, for instance, February 28th 2021, a Sunday, is adjusted to March 1st according to the "following" convention. However, if another convention such as "modified following" needs to be used, it can be passed to the call:

```
s = ql.MakeSchedule(
    effectiveDate=ql.Date(28, ql.February, 2019),
    terminationDate=ql.Date(28, ql.February, 2023),
    frequency=ql.Semiannual,
    calendar=ql.TARGET(),
    convention=ql.ModifiedFollowing,
    forwards=True,
)

pd.DataFrame(list(s), columns=["dates"])
```

	dates
0	February 28th, 2019
1	August 28th, 2019
2	February 28th, 2020
3	August 28th, 2020
4	February 26th, 2021
5	August 30th, 2021
6	February 28th, 2022
7	August 29th, 2022
8	February 28th, 2023

The result shows that February 28th 2021 is adjusted back to February 26th so that it doesn't change month.

Also, in some cases, the terms of an instrument might stipulate that coupons reset at on the last business day of the month; in that case, the schedule can be generated with:

```
s = ql.MakeSchedule(
    effectiveDate=ql.Date(28, ql.February, 2019),
    terminationDate=ql.Date(28, ql.February, 2023),
    frequency=ql.Semiannual,
    calendar=ql.TARGET(),
    endOfMonth=True,
    forwards=True,
)

pd.DataFrame(list(s), columns=["dates"])
```

	dates
0	February 28th, 2019
1	August 30th, 2019
2	February 28th, 2020
3	August 31st, 2020
4	February 26th, 2021
5	August 31st, 2021
6	February 28th, 2022
7	August 31st, 2022
8	February 28th, 2023

By comparing this result with the ones above, you can see the difference of behavior for the dates at the end of August.

Specialized rules

The `forwards` and `backwards` methods shown above are shorthand for calls to a more general method `withRule` that allows to specify a generation rule ("forwards" and "backwards" being two such rules.) Other, more specialized rules are available; for instance, if you needed to generate the schedule for the payments of a standard 5-years credit default swap, you would do it as follows:

```
tradeDate = ql.Date(11, ql.March, 2021)
tenor = ql.Period(5, ql.Years)
maturityDate = ql.cdsMaturity(tradeDate, tenor, ql.DateGeneration.CDS2015)
```

```
s = ql.MakeSchedule(
    effectiveDate=tradeDate,
    terminationDate=maturityDate,
```

```
    frequency=ql.Quarterly,
    calendar=ql.TARGET(),
    rule=ql.DateGeneration.CDS2015,
)

pd.DataFrame(list(s), columns=["dates"])
```

	dates
0	December 21st, 2020
1	March 22nd, 2021
2	June 21st, 2021
3	September 20th, 2021
4	December 20th, 2021
5	March 21st, 2022
6	June 20th, 2022
7	September 20th, 2022
8	December 20th, 2022
9	March 20th, 2023
10	June 20th, 2023
11	September 20th, 2023
12	December 20th, 2023
13	March 20th, 2024
14	June 20th, 2024
15	September 20th, 2024
16	December 20th, 2024
17	March 20th, 2025
18	June 20th, 2025
19	September 22nd, 2025
20	December 20th, 2025

First, the `cdsMaturity` function returns the standardized maturity date for the passed trade date; for March 11th 2021, that would be December 20th 2025 (it would roll to June 2025 only later in March.) Then, we pass the calculated maturity date to `MakeSchedule` while also specifying a `CDS2015` date-generation rule; this recalculates the start date of the CDS and also adjusts all the dates in the schedule to the twentieth of their months of the next business day.

Schedule syntax in C++

In C++, the calls to `MakeSchedule` above would have the following syntax:

```
Schedule s = MakeSchedule()
    .from(Date(28, February, 2019))
    .to(Date(28, February, 2023))
    .withFrequency(Semiannual)
    .withCalendar(TARGET())
    .forwards();
```

And if you're not familiar with this idiom, you might be asking: what about the syntax of `MakeSchedule`? Why aren't we using a constructor like all decent folks, and what happens when we chain all those method calls?

The Named Parameter idiom

The `Schedule` class does have a constructor, of course, but it's a bit awkward to use. As the time of this writing, corresponding to QuantLib 1.32, its signature is:

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

This means it requires a whole lot of parameters, even in the simplest case. Reasonable defaults exist for some of them (a null calendar, following for the conventions, backwards for the generation rule, false for the end of month, and no first or next-to-last date) but if we added them, we'd run into another problem. When we're good with most of the default parameters but want to change one of the last ones (say, `firstDate`), there's no easy syntax we can use for the call. In Python, which supports named parameters, we'd say

```
s = Schedule(
    Date(11, February, 2021),
    Date(11, May, 2025),
    Period(6, Months),
    firstDate = Date(11, November, 2021),
)
```

but in C++, we'd have to pass all the parameters before `firstDate`, even if they all equal the defaults.

The solution? The Named Parameter idiom (a.k.a Fluent Interface). We write a helper class, `MakeSchedule` in our case, which contains the parameters needed to build a schedule and gives them sensible default values:

```
class MakeSchedule {
    ...
  private:
    Calendar calendar_;
    Date effectiveDate_;
    Date terminationDate_;
    Period tenor_;
    BusinessDayConvention convention_ = Following;
    DateGeneration::Rule rule_ = DateGeneration::Backward;
    bool endOfMonth_ = false;
    Date firstDate_ = Date();
    Date nextToLastDate_ = Date();
};
```

Settings the parameters

To set the values of the parameters, we give `MakeSchedule` a number of setter methods; the twist here is that each of these methods returns the object itself, making it possible to chain them.

```
class MakeSchedule {
  public:
    MakeSchedule& from(const Date& effectiveDate) {
        effectiveDate_ = effectiveDate;
        return *this;
    }

    MakeSchedule& to(const Date& terminationDate) {
        terminationDate_ = terminationDate;
        return *this;
    }

    MakeSchedule& withTenor(const Period& tenor) {
        tenor_ = tenor;
        return *this;
    }

    MakeSchedule& forwards() {
        rule_ = DateGeneration::Forward;
```

```
        return *this;
    }

    ...
};
```

Getting our schedule

At this point, we're able to write

```
MakeSchedule()
    .from(Date(11, May, 2021))
    .to(Date(11, February, 2025))
    .withFrequency(Semiannual)
    .withCalendar(TARGET())
    .forwards()
```

but the result is still a `MakeSchedule` instance, not a schedule. In order to build the latter, we could add an explicit `to_schedule()` method that calls the `Schedule` constructor and returns the result. However, we went for a fancier solution.

A little-used feature of C++ are user-defined conversion functions. You can google them for details, but the gist is that, if `A` and `B` are two unrelated classes, you can give `B` a method which returns an instance of `A`, declared as

```
class B {
  public:
    operator A() const;
    ...
};
```

and if you then write

```
B b;
A a = b;
```

the compiler will look first for an `A` constructor taking a `B` instance, and then (after seeing it isn't there) it will look into `B`, find the conversion method, invoke it, and assign to `a` the instance of `A` returned by it.

In our case, the conversion function will be declared in `MakeSchedule` as

```
class MakeSchedule {
  public:
    ...
```

```
    operator Schedule() const;
};
```

and its implementation will call the `Schedule` constructor with the required parameters and return the resulting schedule. Putting everything together, we get the syntax

```
Schedule s = MakeSchedule()
    .from(Date(11, May, 2021))
    .to(Date(11, May, 2025))
    .withFrequency(Semiannual)
    .withCalendar(TARGET());
```

And in case you were wondering why I didn't use a shorter syntax, note that using `auto s` above would not trigger the assignment operator and the conversion to a schedule; it would declare `s` as a `MakeSchedule` instance.

Chapter 5

Cash flows and bonds

In another notebook, I've shown how to build sequences of regular dates (schedules, in QuantLib lingo) and I hinted that they could be used in turn to build sequences of coupons. In this notebook, we'll do just that; and furthermore, we'll see how to use those coupons to build a few simple bonds.

```
import pandas as pd
import QuantLib as ql

ql.Settings.instance().evaluationDate = ql.Date(15, ql.January, 2024)
```

A simple fixed-rate bond

As an example, we're going to create a simple 5-years bond paying 3% semiannually. The first thing we do is a call to `MakeSchedule`, like the ones I described in the previous issue; it takes the start and end date for the coupons, their frequency, and some other information such as the calendar used to adjust dates when they fall on a holiday.

As a reminder, the understanding is that the first date in the schedule is the start of the first coupon; the second date is both the end of the first coupon and the start of the second; the third date is the end of the second coupon and the start of the third; until we get to the last date, which is the end of the last coupon.

```
schedule = ql.MakeSchedule(
    effectiveDate=ql.Date(8, ql.February, 2021),
    terminationDate=ql.Date(8, ql.February, 2026),
    frequency=ql.Semiannual,
```

```
    calendar=ql.TARGET(),
    convention=ql.Following,
    backwards=True,
)
```

In C++, the same call would be written as

```
Schedule schedule =
    MakeSchedule()
    .from(Date(8, February, 2021))
    .to(Date(8, February, 2026))
    .withFrequency(Semiannual)
    .withCalendar(TARGET())
    .withConvention(Following)
    .backwards();
```

The actual task of building the coupons is done by `FixedRateLeg`, another function provided by the library, which is the next thing we call (after defining some other bits of information such as the day-count convention for accruing the coupon rate). It takes the schedule, to be used as described above, and the other data we need.

```
day_count = ql.Thirty360(ql.Thirty360.BondBasis)

fixed_coupons = ql.FixedRateLeg(
    schedule=schedule,
    couponRates=[0.03],
    dayCount=day_count,
    nominals=[10_000],
)
```

`FixedRateLeg`, like `MakeSchedule`, has a number of parameters we can pass to specify the coupons—or, in C++, of methods we can chain: in this case, the C++ code would be

```
Leg fixedCoupons =
    FixedRateLeg(schedule)
    .withCouponRates(0.03, dayCount)
    .withNotionals(10000.0);
```

Here we're doing the base minimum; we're passing the rate to be paid by the coupons, their day-count convention, and the notional used for the calculation of the interest. If needed, we could pass additional specifications: for instance, a payment lag with respect to the end of the coupons, or a number of ex-coupon days.

The name of the C++ return type, `Leg`, is probably more suited to swaps than to bonds. Like many things in the library, which grew bit by bit, it was defined in a particular context

and it stuck. It's not a class in its own right, but an alias for `std::vector<ext::shared_ptr<CashFlow>>` which would be a mouthful to use; `CashFlow` is the base class for cash flows, and derived classes model both coupons (an interest paid based on some kind of rate, fixed or otherwise) and more simple payments like the final redemption of a bond.

The last step, in order to get a working bond instance, is passing the coupons and some additional information to the constructor of the `Bond` class.

```
settlement_days = 3
issue_date = ql.Date(5, ql.February, 2021)

bond1 = ql.Bond(
    settlement_days,
    ql.TARGET(),
    issue_date,
    fixed_coupons,
)
```

The constructor not only stores the coupons, but also figures out that the bond must make a final payment to reimburse its notional. We can see it by asking the bond for its cash flows (via its `cashflows` method) and inspecting them: the last row corresponds to the redemption.

Note that the `cashflow` method returns a list of instances of the base `CashFlow` class; in order to access their more specific methods, we need to downcast them. In C++, we would use `dynamic_pointer_cast<Coupon>`; in Python, where template are not available, the latter is exported as the `as_coupon` function, which returns `None` if the cast doesn't succeed (as in the case of the redemption).

```
data = []
for cf in bond1.cashflows():
    c = ql.as_coupon(cf)
    if c is not None:
        data.append((c.date(), c.nominal(), c.rate(), c.amount()))
    else:
        data.append((cf.date(), None, None, cf.amount()))

pd.DataFrame(
    data, columns=["date", "nominal", "rate", "amount"]
).style.format({"nominal": "{:.0f}", "rate": "{:.3%}", "amount": "{:.3f}"})
```

	date	nominal	rate	amount
0	August 9th, 2021	10000	3.000%	150.833
1	February 8th, 2022	10000	3.000%	149.167

	date	nominal	rate	amount
2	August 8th, 2022	10000	3.000%	150.000
3	February 8th, 2023	10000	3.000%	150.000
4	August 8th, 2023	10000	3.000%	150.000
5	February 8th, 2024	10000	3.000%	150.000
6	August 8th, 2024	10000	3.000%	150.000
7	February 10th, 2025	10000	3.000%	151.667
8	August 8th, 2025	10000	3.000%	148.333
9	February 9th, 2026	10000	3.000%	150.833
10	February 9th, 2026	nan	nan%	10000.000

Finally, we can check that the bond is working by asking, for instance, what would be the yield for a clean price of 98. In C++, the corresponding `Bond` method is called `yield`; in Python, it needed to be renamed to the redundant `bondYield` because `yield` is a reserved keyword.

```
bond1.bondYield(98.0, day_count, ql.Compounded, ql.Semiannual)
```

```
0.04021360635757447
```

A shortcut to the same bond

As you might have guessed, a fixed-rate bond is common enough that we have a more convenient way to instantiate it. Instead of the two separate calls to `FixedRateLeg` and the `Bond` constructor, we can call the constructor of the derived `FixedRateBond` class and pass the same information. Displaying the cash flows of the resulting bond would show that they are the same as the ones above.

```
bond1b = ql.FixedRateBond(
    settlement_days,
    10000.0,
    schedule,
    [0.03],
    day_count,
)
```

A note on overloading

If you were to head over to GitHub and look at changes in the `FixedRateBond` class across versions, you would see that it used to have multiple constructors, but most of them were deprecated and now only one remains. Is that some kind of design principle that should be followed?

Not really, no. In this case, it was a matter of balancing convenience of use in C++ against other languages, such as Python, to which the library is exported. Overloaded constructors would be more convenient in C++, as they would provide shortcuts to different use cases. However, with our current toolchain (the excellent SWIG) they would make it impossible to use keyword arguments in Python; instead, with a single constructor, we can enable them in our Python wrappers and allow writing calls like

```
bond = FixedRateBond(
    settlement_days,
    10000.0,
    schedule,
    [0.03],
    day_count,
    firstPeriodDayCounter=another_day_count,
)
```

where we don't need to pass (like in C++) all the default parameters defined by the constructor between the two day counts.

So, who should win? C++ or Python? I don't have any figures about the number of people that use QuantLib in either language, but in this case C++ could graciously yield without suffering much harm; as we've seen, `FixedRateLeg` already gave it a pretty convenient way to instantiate various use cases. As usual, the mileage in your code might vary.

An amortizing fixed-rate bond

I mentioned above that `FixedRateLeg` has a number of additional methods I didn't show; moreover, the ones I did show can take additional information. In the next call, for instance, the code passes a sequence of different notionals for the generated coupons, starting from 10000 in the first year and decreasing by 2000 each year; that is, every second coupon, since the schedule is semiannual.

```
amortizing_coupons = ql.FixedRateLeg(
    schedule,
    couponRates=[0.03],
    dayCount=day_count,
    nominals=[
        10000.0,
        10000.0,
        8000.0,
        8000.0,
        6000.0,
        6000.0,
```

```
        4000.0,
        4000.0,
        2000.0,
        2000.0,
    ],
)
```

The C++ call would be

```
Leg amortizingCoupons =
    FixedRateLeg(schedule)
    .withCouponRates(0.03, dayCount)
    .withNotionals({
        10000.0, 10000.0,
        8000.0, 8000.0,
        6000.0, 6000.0,
        4000.0, 4000.0,
        2000.0, 2000.0
    });
```

In this case, passing the returned coupons to the `Bond` constructor results in an amortizing bond; again, we can check that by displaying its cash flows. As before, the machinery in the constructor looked at the notionals of the coupons and figured out that, when the notional varies between two consecutive coupons, a notional payment needed to be inserted.

```
bond2 = ql.Bond(
    settlement_days,
    ql.TARGET(),
    issue_date,
    amortizing_coupons,
)
```

```
data = []
for cf in bond2.cashflows():
    c = ql.as_coupon(cf)
    if c is not None:
        data.append((c.date(), c.nominal(), c.rate(), c.amount()))
    else:
        data.append((cf.date(), None, None, cf.amount()))

pd.DataFrame(
    data, columns=["date", "nominal", "rate", "amount"]
).style.format({"nominal": "{:.0f}", "rate": "{:.3%}", "amount": "{:.3f}"})
```

	date	nominal	rate	amount
0	August 9th, 2021	10000	3.000%	150.833
1	February 8th, 2022	10000	3.000%	149.167
2	February 8th, 2022	nan	nan%	2000.000
3	August 8th, 2022	8000	3.000%	120.000
4	February 8th, 2023	8000	3.000%	120.000
5	February 8th, 2023	nan	nan%	2000.000
6	August 8th, 2023	6000	3.000%	90.000
7	February 8th, 2024	6000	3.000%	90.000
8	February 8th, 2024	nan	nan%	2000.000
9	August 8th, 2024	4000	3.000%	60.000
10	February 10th, 2025	4000	3.000%	60.667
11	February 10th, 2025	nan	nan%	2000.000
12	August 8th, 2025	2000	3.000%	29.667
13	February 9th, 2026	2000	3.000%	30.167
14	February 9th, 2026	nan	nan%	2000.000

More cash flows and bonds

The library, of course, is not limited to fixed-rate coupons. For instance, a C++ call like

```
Leg cashflows =
    IborLeg(schedule, euribor6m)
    .withSpreads(0.001)
    .withNotionals(10000.0);
```

would generate coupons paying the fixings of the 6-months Euribor rate plus 10 basis points. You'll forgive me for only showing this in passing; adding it as an example to the code would require setting up the index and its forecast curve, and that is probably best done in another notebook with more details.

Part II

Library infrastructure

Before we go deeper into specific instruments and how to price them, we need to lay some more necessary groundwork.

There are some classes and patterns in QuantLib that define its basic infrastructure—its architecture, if you will—and need to be known so that you can use the library in an idiomatic way, making it work with you and not against you.

These facilities include handles, the base `Instrument` and `PricingEngine` class with their interplay, the base behavior of the `TermStructure` class, quotes, and the Observable pattern embedded in most of the library classes. They are the topic of the next few chapters.

Chapter 6

Handling dependencies

(Originally published as an article in Wilmott Magazine, March 2023.)

In this notebook, I'll show a class which is used everywhere in the code, is a critical part of the design of the library, and might just be a confusing complication if I don't explain what problem it actually solves for you. It's the `Handle` class.

A use case for handles: setting up

Let's say we hold a number of different bonds. In the first part of the code, we build four of them and collect them in a list. In the interest of brevity, they're all zero-coupon bonds with different maturity dates; note, though, that we could have easily mixed different kinds of bonds there.

```
import QuantLib as ql
```

```
value_date = ql.Date(7, ql.March, 2023)
ql.Settings.instance().evaluationDate = value_date

settlement_days = 3
face_amount = 100.0
calendar = ql.TARGET()

bonds = [
    ql.ZeroCouponBond(
        settlement_days,
        calendar,
```

```
        face_amount,
        ql.Date(10, ql.February, 2025),
    ),
    ql.ZeroCouponBond(
        settlement_days,
        calendar,
        face_amount,
        ql.Date(26, ql.August, 2030),
    ),
    ql.ZeroCouponBond(
        settlement_days, calendar, face_amount, ql.Date(13, ql.May, 2024)
    ),
    ql.ZeroCouponBond(
        settlement_days,
        calendar,
        face_amount,
        ql.Date(15, ql.August, 2040),
    ),
]
```

Then, let's assume that the bonds were issued by two different entities, and that we have discount curves available for both. They might come from an external system, or you might have bootstrapped or fitted them using some other facility provided by QuantLib. In this example, we build them using the `ZeroCurve` class, which interpolates a set of given zero rates at different dates; a real system would provide us with a lot more nodes but, again, I tried to keep the code short. As for bonds, the specific class doesn't matter; any class inherited from the base `YieldTermStructure` class would do.

```
curve_nodes = [
    value_date,
    value_date + ql.Period(10, ql.Years),
    value_date + ql.Period(20, ql.Years),
]
curve_rates_1 = [0.015, 0.025, 0.028]
curve_rates_2 = [0.005, 0.009, 0.012]

issuer_curve_1 = ql.ZeroCurve(curve_nodes, curve_rates_1, ql.Actual360())
issuer_curve_2 = ql.ZeroCurve(curve_nodes, curve_rates_2, ql.Actual360())
```

And here comes the step which might seem unnecessary: instead of using the curves directly, we wrap them in handles, namely, we pass each curve to an instance of `RelinkableYield-TermStructureHandle`. Yes, it's a mouthful.

```
issuer_handle_1 = ql.RelinkableYieldTermStructureHandle(issuer_curve_1)
issuer_handle_2 = ql.RelinkableYieldTermStructureHandle(issuer_curve_2)
```

To complete our setup, we pass each of the handles to an instance of `DiscountingBondEngine`, which can calculate bond prices by discounting their coupons according to the given curve. Assuming that the first two bonds come from the first issuer and the second two from the other, we set the relevant engine to each bond; and finally, we can output their prices.

```
engine_1 = ql.DiscountingBondEngine(issuer_handle_1)
engine_2 = ql.DiscountingBondEngine(issuer_handle_2)

bonds[0].setPricingEngine(engine_1)
bonds[1].setPricingEngine(engine_1)
bonds[2].setPricingEngine(engine_2)
bonds[3].setPricingEngine(engine_2)

for b in bonds:
    print(f"{b.cleanPrice():.4f}")
```

```
96.7459
84.3484
99.3479
81.9757
```

The benefits of using handles

Now, let's suppose you want to create a modified curve in order to simulate what happens if the credit rating of the first issuer worsens; or maybe you want to change the set of instruments on which you're fitting one of your curves, or again your system tells you that one of the curves has changed.

Whatever the reason, we can create a new curve and pass it to the `linkTo` method of the corresponding handle: in this example, to simulate a worsening of the credit, we use a `ZeroSpreadedTermStructure` instance which adds a spread on top of the existing curve. After doing this, we ask the bonds for their prices, and the first two bonds will return updated figures. No need to tell the bonds or the engines that their discount curve has changed; the library takes care of that.

```
shifted_curve_1 = ql.ZeroSpreadedTermStructure(
    ql.YieldTermStructureHandle(issuer_curve_1),
    ql.QuoteHandle(ql.SimpleQuote(0.001)),
)
```

```
issuer_handle_1.linkTo(shifted_curve_1)

for b in bonds:
    print(f"{b.cleanPrice():.4f}")
```

```
96.5572
83.7121
99.3479
81.9757
```

This, quite simply, is the reason for the existence of handles and the central role they play in the design of QuantLib: the ability to manage different term structures, quotes, and any kind of market dependencies without having to update each instrument explicitly or without recreating the engine.

And now, we can dig deeper in the way handles work.

Not just pointers

You probably guessed that in the C++ library code, the `Handle` class works as some kind of pointer, keeping a reference to an external object and thus providing the means to observe the updated state of the object if it changes. You might also be thinking, "Wasn't `shared_ptr` enough for that?" Yes, it would be enough for some cases, but not for all.

As a general rule which might deserve to be stated explicitly (and please bear with me a minute if this is old news for you), when we say that we're "passing an object by pointer", as in the constructor of the following class:

```
class C {
    A* p2;
  public:
    C(A* p1) : p2(p1) {}
};
```

what really happens is that we're passing a pointer by copy; i.e., the stored `p2` is a copy of the passed `p1`. Since they happen to be pointers, the end result is that they contain the same address and thus refer to the same instance of the `A` class.

For instance, in the context of QuantLib: `C` might be an instrument, and might be given a pointer `p1` to a curve (`A`) bootstrapped over a set of quoted OIS rates. If the curve updated its nodes because the quotes changed, the copied pointer `p2` would allow you to access the updated curve—which updated its nodes but is still the same instance.

What happens if we do the following, though?

```
A a1;
A* p1 = &a1;

C c(p1);

A a2;
p1 = &a2;
```

We have an instance `a1`; we take a pointer `p1`, let it point to `a1` and pass it to `c`, which stores a copy of the pointer. Later, we create another instance `a2` and update `p1` so that it points to `a2`.

Flash quiz: what instance of `A` does `c.p2` access, now? Exactly: the original instance `a1`. We modified `p1`, but this didn't modify `c.p2` which is a distinct copy.

Adding another level of indirection

If you wanted `c` to access the new object `a2`, the semantics you need are those of *pointers to pointers*. The class `C` needs to be written as:

```
class C {
    A** pp2;
  public:
    C(A** pp1) : pp2(pp1) {}
};
```

What does this give us? We can now write:

```
A a1;
A* p1 = &a1;
A** pp1 = &p1;

C c(pp1);

A a2;
p1 = &a2;
```

The class `C` now takes a copy of the pointer-to-pointer `pp1`, which points to `p1`. Flash quiz: when we update `p1`, what instance of `A` can `c` access? As before, the pointer-to-pointer `c.pp2` was not modified; but this time, this means that it still points to `p1` and it can see its updated value, which now stores the address of (drum roll) `a2`.

This is exactly the semantics of handles in QuantLib; they allow you to change the object they point to, and make the change accessible to any other object that stores a copy of the

same handle. I'll gloss over the implementation here; you can find it described in my book *Implementing QuantLib*.

It sounds complicated. Why not setters?

Well, yes, you could have a `setDiscountCurve` method declared in the engine. But you'd have to keep track of which engines are using each curve. And of which indexes need forecast curves. And any number of other objects.

Also, if you, say, model your discount curves as a risk-free curve plus a credit spread, you'd want to set those components, not the whole discount curve. And as I mentioned, if you have other instruments in the portfolio, you'll soon find yourself in a mess of `setDiscountCurve`, `setForecastCurve`, `setDefaultProbability` and so on. All in all, handles sound less complicated to me.

Why not just pointers to pointers then?

That wouldn't be wrong, either. However, using a specific `Handle` class is more convenient, for a number of reasons. The simplest is that in 2000, when we started (or in 2010, for that matter,) C++ didn't have `auto` to infer types, or `using` to declare template aliases, or even `shared_ptr`; and writing `boost::shared_ptr<boost::shared_ptr<YieldTermStructure> >` was a lot less convenient than `Handle<YieldTermStructure>`.

However—and here we go into some general principles—there are far stronger reasons for creating a wrapper class. If we used the pointer classes provided by the standard (or, in 2000, by Boost,) we would provide to the user the interface of that class; nothing more and also nothing less. Instead, a wrapper class allows us to define a custom interface more suited to our needs.[1]

The advantage of doing this is twofold. On the one hand, it makes it possible to add behavior; for instance, relinking a handle also sends notifications to all dependent objects (again, a lot more details are in *Implementing QuantLib*.) But on the other hand, and just as importantly, it makes it possible to remove behavior and thus hide all the methods of `shared_ptr` that might cause problems.

Case in point: in QuantLib we have `RelinkableHandle`, which declares a `linkTo` method, as well as a `Handle` class which doesn't. A `RelinkableHandle` can be converted to a `Handle`, but not the other way around; much like a non-`const` object can be converted to a `const`. The idea is that the main function, or some higher-level procedure that manages the calculations, will create relinkable handles and manage them. When the handles are passed and stored into instruments or other objects, they are passed as `Handle` so they can't be relinked.

[1] This also holds for containers such as `vector` or `list`, and often also for basic types such as `int`, `double` or `string`. Search for "primitive obsession" if you want more info. Among others, GeePaw Hill and Ron Jeffries have some good content about it.

If we didn't restrict the interface (or if we passed a pointer to pointer) it would be possible for an instrument to link, say, its volatility handle to something else in order to perform some internal calculation; but in doing so, it would affect other instruments which might also share the same handle (like the bonds from the same issuer, in the code example).

How do we know this? Years ago, we had to pull a release because of a problem like the one I just described. We only had `Handle` at the time, and it always allowed relinking—and options would merrily relink their handles while calculating their implied volatility and leave them modified. Hence, `RelinkableHandle` and `Handle`. Like for `const`, conventions are great, but having the compiler check them is even better.

Chapter 7

Instruments and pricing engines

This notebook showcases a couple of features that the infrastructure of the library makes available; namely, it will show how instruments can use different so-called pricing engines to calculate their prices (each engine implementing a given model and/or numerical method) and how engines and instruments can be notified of changes in their input data and react accordingly.

Setup

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

```
import QuantLib as ql

today = ql.Date(7, ql.March, 2024)
ql.Settings.instance().evaluationDate = today
```

The instrument

In this notebook, we'll leave fixed-income and take a textbook instrument 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. The instrument doesn't take any market data; they will be selected and passed later, depending on the calculation method.

```
option = ql.EuropeanOption(
    ql.PlainVanillaPayoff(ql.Option.Call, 100.0),
    ql.EuropeanExercise(ql.Date(7, ql.June, 2024)),
)
```

A first pricing method

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: they 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%.

```
u = ql.SimpleQuote(100.0)
r = ql.SimpleQuote(0.01)
σ = ql.SimpleQuote(0.20)
```

In order to build the engine, the market data are encapsulated in a Black-Scholes process object. The process can use full-fledged term structures, so it can include time-dependency and smiles. In this case, for simplicity, we build flat curves for the risk-free rate and the volatility.

```
riskFreeCurve = ql.FlatForward(
    0, ql.TARGET(), ql.QuoteHandle(r), ql.Actual360()
)
volatility = ql.BlackConstantVol(
    0, ql.TARGET(), ql.QuoteHandle(σ), ql.Actual360()
)
```

Now we can 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.

```
process = ql.BlackScholesProcess(
    ql.QuoteHandle(u),
    ql.YieldTermStructureHandle(riskFreeCurve),
    ql.BlackVolTermStructureHandle(volatility),
)
```

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

```
engine = ql.AnalyticEuropeanEngine(process)
```

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

```
option.setPricingEngine(engine)
```

```
print(option.NPV())
```

```
4.155543462156205
```

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

```
print(option.delta())
print(option.gamma())
print(option.vega())
```

```
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.

```
u.setValue(105.0)
print(option.NPV())
```

```
7.275563579278455
```

Other market data also affect the value, of course.

```
r.setValue(0.02)
print(option.NPV())
```

```
7.448878025811252
```

```
σ.setValue(0.15)
print(option.NPV())
```

```
6.596556078273314
```

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…

```
u.setValue(105.0)
r.setValue(0.01)
σ.setValue(0.20)
print(option.NPV())
```

```
7.275563579278455
```

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

```
ql.Settings.instance().evaluationDate = ql.Date(7, ql.April, 2024)
```

Again, we don't have to do anything explicitly: we just ask the option for its value, and we see that it has decreased as expected.

```
print(option.NPV())
```

```
6.535204576446802
```

A note on the option value on its exercise date

In the default library configuration, the instrument is considered to have expired when it reaches the exercise date, so its returned value goes down to 0.

```
ql.Settings.instance().evaluationDate = ql.Date(7, ql.June, 2024)
```

```
print(option.NPV())
```

```
0.0
```

It's possible to tweak the configuration so that the instrument is still considered alive.

```
ql.Settings.instance().includeReferenceDateEvents = True
```

The above changes the settings, but doesn't send a notification to the instrument so we need to trigger an explicit recalculation. Normally, though, one would change the setting at the start of one's program so this step would be unnecessary.

```
option.recalculate()

print(option.NPV())
```

```
5.0
```

However, this is not guaranteed to work for all pricing engines, since each one must manage this case specifically; and even when they return a price, they are not guaranteed to return meaningful values for all available results. For instance, at the time of this writing, the cell below will print two NaNs; if it doesn't, please send me a line so I can update this text.

```
print(option.delta())
print(option.vega())
```

```
nan
nan
```

Other pricing methods

As I mentioned, the instrument machinery 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.

```
ql.Settings.instance().evaluationDate = today
u.setValue(105.0)
r.setValue(0.01)
σ.setValue(0.20)
```

```
print(option.NPV())
```

```
7.275563579278455
```

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 (here I'll skip the calibration and pass precalculated parameters)…

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

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

```
engine = ql.AnalyticHestonEngine(model)
option.setPricingEngine(engine)
```

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

```
print(option.NPV())
```

```
7.295356086978647
```

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 we asked it, 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.

```
engine = ql.MCEuropeanEngine(
    process, "PseudoRandom", timeSteps=20, requiredSamples=500_000
)
option.setPricingEngine(engine)
```

When we ask for the option value, we have to wait a noticeable time for the calculation to finish (for those of you reading this in a non-interactive way, I'll also have the notebook output the time)…

```
%time print(option.NPV())
```

```
7.248648366455495
CPU times: user 1.94 s, sys: 14.9 ms, total: 1.95 s
Wall time: 1.95 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.

```
%time print(option.NPV())
```

```
7.248648366455495
CPU times: user 29 µs, sys: 2 µs, total: 31 µs
Wall time: 32.9 µs
```

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

```
u.setValue(104.0)
```

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

```
%time print(option.NPV())
```

```
6.5774920681332025
CPU times: user 1.93 s, sys: 15.3 ms, total: 1.95 s
Wall time: 1.95 s
```

Chapter 8

The Observer pattern

This chapter was originally published as an article in the November 2023 issue of Wilmott Magazine, and took inspiration from a discussion which was taking place as I wrote it and whose result you will find implemented in the library by the time you read this.

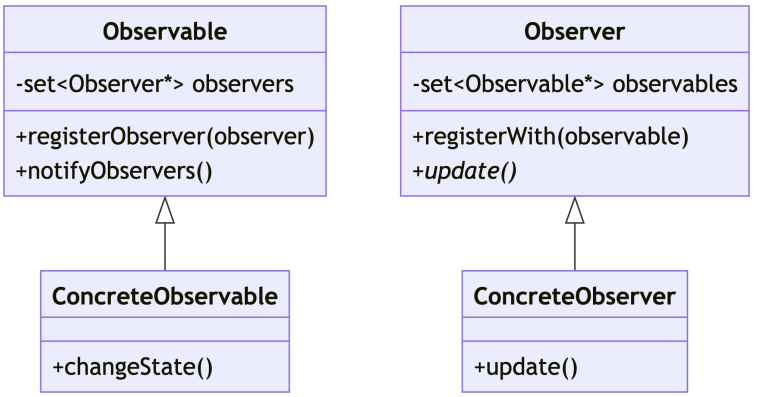
The subject of the discussion was the implementation of the Observer pattern in QuantLib. I'll try to use it to show some of the trade-offs and changes that a design pattern might have to undergo, in order to address constraints that come from real-world usage of a library. As will be clear to anyone that looks at the version-control logs of QuantLib, the ideas and code in the library are by no means the result of my work alone; and in this case especially, the ideas I'll describe come from a number of clever people of which yours truly is not one.

I will also provide sample C++ code to show some possible setup where the pattern comes into play. Let's take the headers out of the way before we start:

```
#include <ql/indexes/ibor/estr.hpp>
#include <ql/indexes/ibor/euribor.hpp>
#include <ql/instruments/makevanillaswap.hpp>
#include <ql/pricingengines/swap/discountingswapengine.hpp>
#include <ql/termstructures/yield/oisratehelper.hpp>
#include <ql/termstructures/yield/piecewiseyieldcurve.hpp>
#include <ql/time/calendars/target.hpp>
#include <ql/time/daycounters/actual360.hpp>
#include <ql/time/daycounters/thirty360.hpp>
#include <iostream>
```

The Observer pattern

The basic structure of the pattern is the one described in the classic Gang of Four book[1] and shown in the figure below.



The `Observable` class (called Subject in *Design Patterns*) models anything that can change in time; for instance a market quote, or a term structure of some kind. The `Observer` class, instead, models anything that depends on one or more observables and needs to be notified when they change; for instance, an instrument whose price depends on some quote or term structure.

Note that a given object can be both an observer and an observable; for instance, an interest-rate curve can be observed by instruments and in turn can observe a number of quoted market rates over which it is bootstrapped.

The implementation (not shown here; I'd need to oversimplify it) provides the means for observables to broadcast their changes and for interested observers to react. In short: an observer will ask to register with one or more observables, each of whom will add it to the set of its observers. When a concrete observable object (say, a quote) executes a method that changes its state, that method will also call `notifyObservers`, which will (unsurprisingly) notify all its observers; this is done by looping over them and calling the `update` method of each one. In a concrete observer object, that method will perform whatever specific action it needs to react to the change.

Oh, and of course, we're not barbarians. The diagram shows sets of pointers for the sake of conciseness, but those sets actually contain instances of `shared_ptr` and, to avoid cycles, `weak_ptr`.

[1] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

Some assembly required

The authors of the original *Design Patterns* book warned that the patterns they examined were (I'm paraphrasing) some kind of blueprint, and the implementations they provided were not prescriptive but just examples that should be adapted depending on the context, the specific requirements of one's use case, and the language used among other things.

This was certainly true in our case. As we started to use the pattern, we became aware of questions that our usage caused, and whose answer can't be prescribed or even considered in a generic description. For instance: what if, during the notification loop, one of the observers' `update` methods raises an exception? Do we let it escape the loop or do we catch it? Our compromise in this case was to catch it (so that the loop doesn't end and all observers are notified) and to raise an exception at the end of the loop.

If you look at our implementation, you'll probably see a number of these concerns. In this article, though, I'll be focusing on one particular problem that's been with us for a couple of decades and that, as I mentioned in the introduction, we're still wrestling with.

Large notification chains

Let's say we have a dual-curve framework for pricing interest-rate swaps: an ESTR curve is bootstrapped over a set of OIS quotes, and then it is used as discount curve in the bootstrap of a forecast curve for the 6-months Euribor index, based on a set of swap quotes. The two curves are then used to price a fixed-vs-floating swap.

Here is the code to set up the above. First we need to create the quotes for the set of OIS, the handles holding them, and the helpers object used for the bootstrap, as well as the ESTR curve itself and a handle that contains it. I know I still haven't covered this in this guide; you'll have to trust me for the time being.

```
int main() {

   using namespace QuantLib;

   auto valueDate = Date(8, August, 2023);
   Settings::instance().evaluationDate() = valueDate;

   struct RateQuote {
      Period tenor;
      Rate rate;
   };

   std::vector<RateQuote> oisData = {
```

```
    {Period(1,  Years), 3.85},
    {Period(2,  Years), 3.52},
    {Period(3,  Years), 3.29},
    {Period(5,  Years), 3.11},
    {Period(7,  Years), 3.04},
    {Period(10, Years), 3.02},
    {Period(12, Years), 3.01},
    {Period(15, Years), 2.97},
    {Period(20, Years), 2.90},
    {Period(30, Years), 2.85}
};

auto estr = ext::make_shared<Estr>();

std::vector<ext::shared_ptr<SimpleQuote>> oisQuotes;
std::vector<ext::shared_ptr<RateHelper>> oisHelpers;
for (const auto& q : oisData) {
    Real rate = q.rate / 100.0;
    auto quote = ext::make_shared<SimpleQuote>(rate);
    auto helper = ext::make_shared<OISRateHelper>(
        2, q.tenor, Handle<Quote>(quote), estr);
    oisQuotes.push_back(quote);
    oisHelpers.push_back(helper);
}

auto estrCurve = ext::make_shared<
    PiecewiseYieldCurve<Discount, LogLinear>>(
    0, TARGET(), oisHelpers, Actual360());
auto estrHandle = Handle<YieldTermStructure>(estrCurve);
```

Then we repeat the process for the Euribor curve, using the ESTR curve as an additional dependency for the second set of helpers.

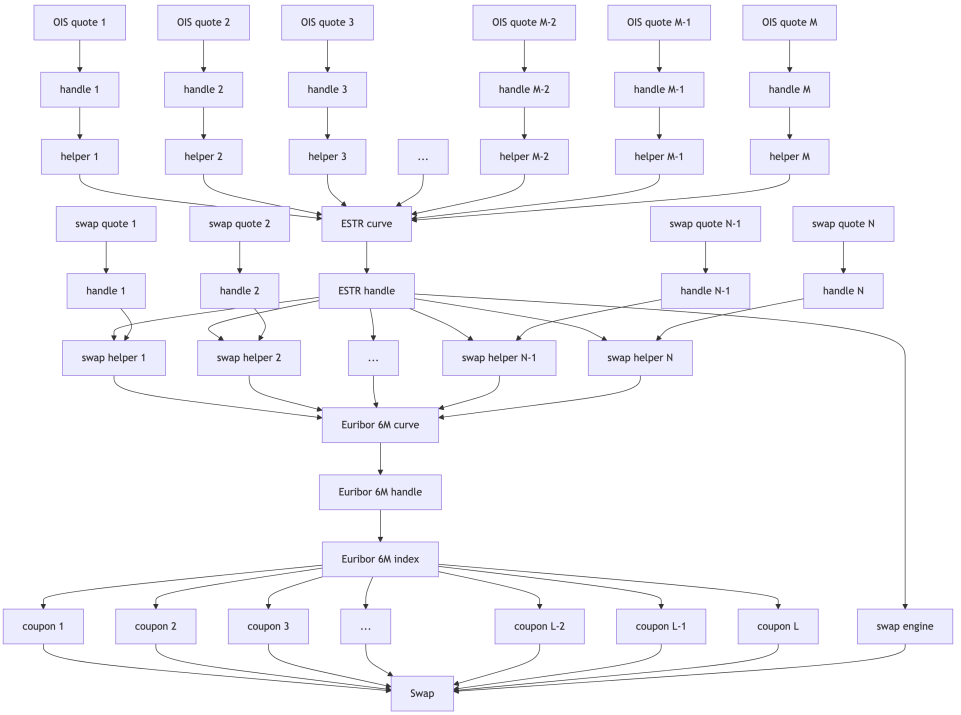
```
std::vector<RateQuote> swapData = {
    {Period(1,  Years), 4.13},
    {Period(2,  Years), 3.87},
    {Period(3,  Years), 3.54},
    {Period(5,  Years), 3.37},
    {Period(7,  Years), 3.22},
    {Period(10, Years), 3.20},
    {Period(20, Years), 3.10},
    {Period(30, Years), 2.91}
```

```
};

auto euribor6m = ext::make_shared<Euribor6M>();
std::vector<ext::shared_ptr<RateHelper>> swapHelpers;
for (const auto& q : swapData) {
   Real rate = q.rate / 100.0;
   auto quote = ext::make_shared<SimpleQuote>(rate);
   auto helper = ext::make_shared<SwapRateHelper>(
      Handle<Quote>(quote), q.tenor, TARGET(), Annual,
      Unadjusted, Thirty360(Thirty360::European),
      euribor6m, Handle<Quote>(), 0 * Days, estrHandle);
   swapHelpers.push_back(helper);
}

auto euriborCurve = ext::make_shared<
   PiecewiseYieldCurve<Discount, LogLinear>>(
   0, TARGET(), swapHelpers, Actual360());
auto euriborHandle =
   Handle<YieldTermStructure>(euriborCurve);
```

Finally, we build an instance of the `Euribor6M` class that can turn the forecast curve into predicted fixings, we pass it to the constructor of the swap we want to price (which also builds all its coupons) and we set it a pricing engine that uses the ESTR curve for discounting.

```
euribor6m = ext::make_shared<Euribor6M>(euriborHandle);
ext::shared_ptr<VanillaSwap> swap =
   MakeVanillaSwap(10 * Years, euribor6m, 0.03);
swap->setPricingEngine(
   ext::make_shared<DiscountingSwapEngine>(estrHandle));

std::cout << swap->NPV() << std::endl;
```

With the exception of the quotes (which are only observed), most of these objects observe their dependencies and are in turn observed by other objects built on top of them. In this case, which I would call a run-of-the-mill one, this leads to the large chain of dependencies shown in the next figure.



When the market moves and the quotes change, as in the next few lines of the code (where, just as an example, all rates increase by one basis point), they start a deluge of notifications.

```
   for (auto& q : oisQuotes)
      q->setValue(q->value() + 0.0001);

   std::cout << swap->NPV() << std::endl;
}
```

Each quote notifies its handle, which in turn notifies its helper and the ESTR curve. In a naive implementation, the ESTR curve would forward each notification to every swap helper, and each of them would notify the Euribor curve—and so on, with the Euribor index later notifying each coupon and each coupon notifying the swap.

The number of OIS quotes, the number of swap helpers and the number of coupons compound. That's way too many notifications going around and telling objects to update themselves. We need to reduce the work done.

Lazy objects

Our first order of business, though, is to reduce needless recalculations. I'm referring to objects like the ESTR or Euribor curve; when the market moves, we want them to recalculate only when the values of all the quotes have been changed, not each time they receive a notification because a single quote has moved.

This was done a long time ago by writing the `LazyObject` class and inheriting the piecewise curves from it. You can look up the details in *Implementing QuantLib* if you want, but the gist is that its `update` method (which, as you remember, is the one that gets called when a notification comes) doesn't do anything except setting a dirty bit which means that the lazy object is out of date. Recalculation is only performed (and the dirty bit is reset) when the object is asked for results; for instance, when the curves are asked for discount factors or forward rates. A number of other classes in the library inherit from `LazyObject`; in particular, all instruments and, as of the next version of the library, all cash flows.

Lazy notifications

The behavior of lazy objects suggested (again, a long time ago) another optimization. The reasoning went like this: when a `LazyObject` instance is up to date and receives a notification, it sets its dirty bit and forwards the notification to its own observers. Now, as long as the dirty bit is not reset, we know that the object wasn't yet asked for any results. This, in turn, means (we thought) that its observers have not yet tried to recalculate. Therefore, when another notification comes while the dirty bit is still not reset, it's pointless to forward it, because the object's observers have already received the first one and didn't yet act upon it; they will in good time.

This works great for reducing the number of notifications. When each quote sends a notification that eventually reaches the ESTR curve, the latter only forwards the first to each swap helper, and so does the Euribor curve. The number of OIS quotes, the number of swap helpers and the number of coupons no longer compound.

In fact, this worked so great that it took us several years to find out that our assumptions didn't work in some specific cases. As I wrote, we assumed that if a lazy object was not asked for results, it meant that its observers didn't yet try to recalculate; however, it was also possible that an instrument started the calculation, but it found out it was already expired so it didn't bother asking the lazy object for results. From that point onward, the lazy object would happily swallow genuine notifications.

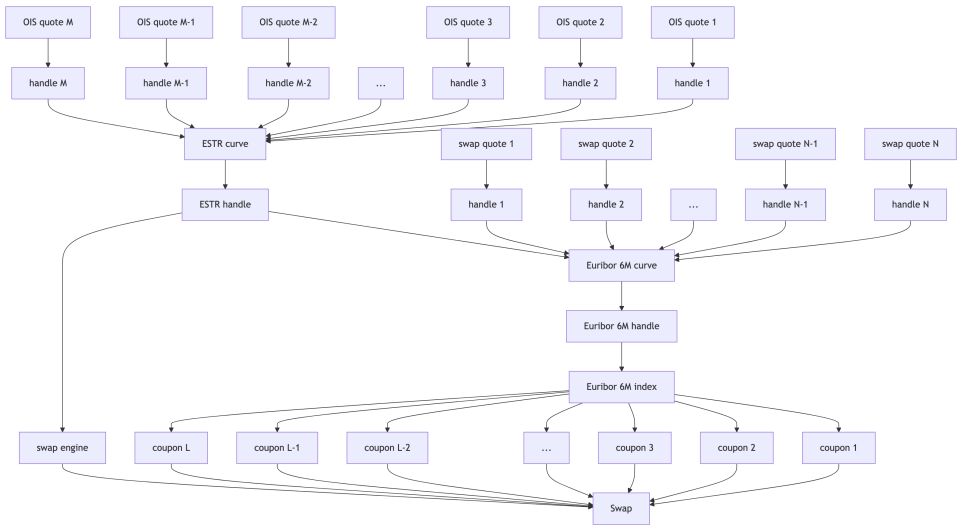
The solution was to add the possibility for a lazy object to forward all notifications, and to enable it on a per-object basis in the specific cases we had found. Recently, though, we've been talking again about this, and in the end we also added the possibility to trade some speed for more safety (quite some speed, at times) and enable this behavior for all lazy objects. That's not the default, and I'm not sure it will ever be. However, it spurred another discussion on

whether it's possible to reduce the number of notifications.

Simplifying the notification chain

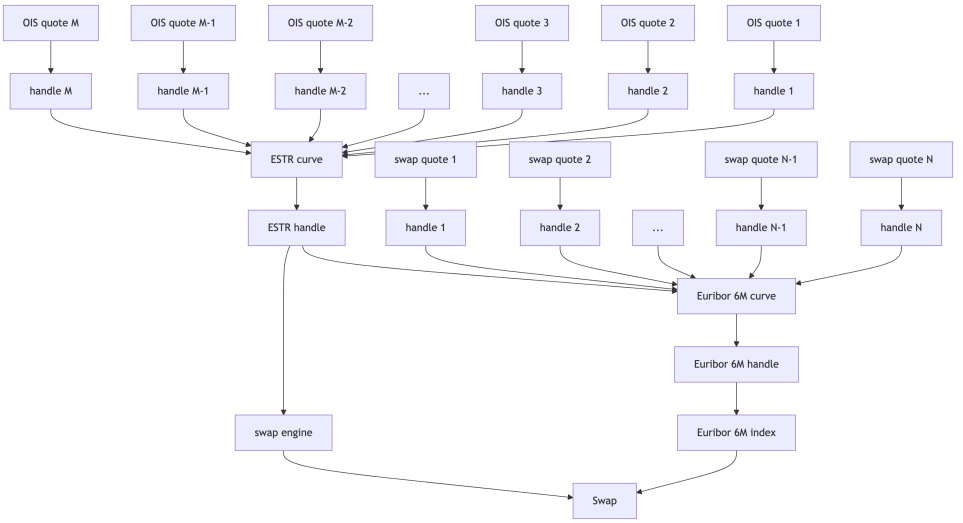
Well, it turns out it is possible. One recent pull request simplifies the chain in a particular case: the OIS and swap helpers don't generate notifications by themselves, they only forward them, and they are not going to change once the curve is built, so we can let the curves register directly with whatever their helpers are observing (the `Observer` class has a method, `registerWithObservables`, for this purpose).

This yields the smaller notification chain shown in the figure below, and it means, e.g., that the ESTR handle no longer sends a notification to each swap helper, but it only sends a single notification directly to the Euribor curve.



More pruning

A similar simplification could be made for the swap coupons, if we know they're not going to change. In general, this can't be guaranteed (for instance, coupons can be set different pricers during their lifetime) and therefore the pruning can't be done automatically; but if you're the one doing the setup and you know you're not going to change the coupon pricer, there's a `simplifyNotificationGraph` function available that you can call manually. In our sample case, this would result in the smaller notification graph in the next figure, in which the Euribor index sends a single notification to the swap instead of sending one to each coupon.



Summary

This article was not meant to defend our implementation, but to give an example of the problems and the constraints that come up in real-world usage. The idea was to give you some food for thought, and maybe to cause you to ask some questions—such as, for example, "Dear QuantLib developers, did you by any chance paint yourselves in a corner by choosing the Observer pattern?"

Dear reader, it's possible. The Observer pattern influences heavily the way the library was implemented and is used. As usual, it's all about tradeoffs. In a way, yes, we did bind our own hands; but this made it possible to use the library in what we feel is a natural way. I'm not suggesting that you should do the same, if you were to write your library; your context might vary.

That's all for today. If we ever meet for a beer, I can tell you about a second, thread-safe implementation of the pattern in the library. It's slower but, when using QuantLib from C# or Java, it prevents the garbage collector from deleting objects while they're being notified and from causing a segmentation fault. All about tradeoffs, remember?

Chapter 9

Updating multiple market quotes

We have already seen in other examples how instances of `SimpleQuote` can be updated and how they notify their observers. In this notebook, I'll show a possible pitfall to avoid when multiple quotes need to be updated.

```
import numpy as np
from matplotlib import pyplot as plt
import QuantLib as ql

today = ql.Date(17, ql.October, 2016)
ql.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 bonds. Here are the maturities in years and the coupons of the bonds I'll use:

```
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. Of course, this is not a requirement for the fit: when using real data, each bond will have its own schedule and price.

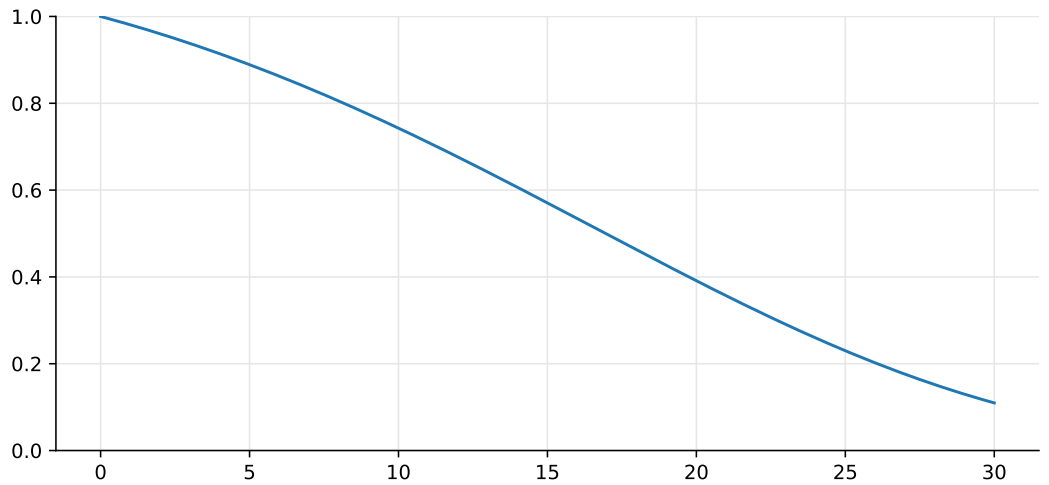
I'll skip over the details of building the curve now; the one thing you'll need to remember is that it depends on the input quotes modeling the bond prices.

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

```
            [coupon],
            ql.SimpleDayCounter(),
            ql.ModifiedFollowing,
        )
    )

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

Just for kicks, here is a visualization of the curve as discount factors versus time in years:

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

ax = plt.figure(figsize=(9, 4)).add_subplot(1, 1, 1)
ax.set_ylim(0.0, 1.0)
ax.plot(sample_times, sample_discounts);
```



And here is a bond priced by discounting its coupons on the curve:

```
schedule = ql.Schedule(
    today,
    calendar.advance(today, 15, ql.Years),
    ql.Period(ql.Semiannual),
    calendar,
    ql.ModifiedFollowing,
```

```
    ql.ModifiedFollowing,
    ql.DateGeneration.Backward,
    False,
)
bond = ql.FixedRateBond(3, 100.0, schedule, [0.04], ql.Actual360())
bond.setPricingEngine(
    ql.DiscountingBondEngine(ql.YieldTermStructureHandle(curve))
)
print(bond.cleanPrice())
```

```
105.77449627458306
```

"It looked like a good idea at the time"

The bond we created is, indirectly, an observer of the market quotes. However, it is also an observable, and will forward to its own observers (if any) the notifications it receives.

Therefore, if you're writing some kind of interactive application, it might be tempting to 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 creating an instance of `Observer` (with a function to be executed when it receives a notification) and by registering it 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 corresponding helper, which in turn will notify the curve, and so on; the notification will reach the pricing engine, the bond and finally our observer.

```
prices = []


def print_price():
    p = bond.cleanPrice()
    prices.append(p)
    print(p)


o = ql.Observer(print_price)
o.registerWith(bond)
```

The function we created 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: when we set a new value to one of the quotes,

`print_price` should be called and a bond price should be printed out.

```
quotes[2].setValue(101.0)
```

```
105.77449627458306
105.86560416829894
```

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.

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

```
quotes[2].setValue(100.0)
```

```
105.86560416829894
105.77449623091606
```

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.

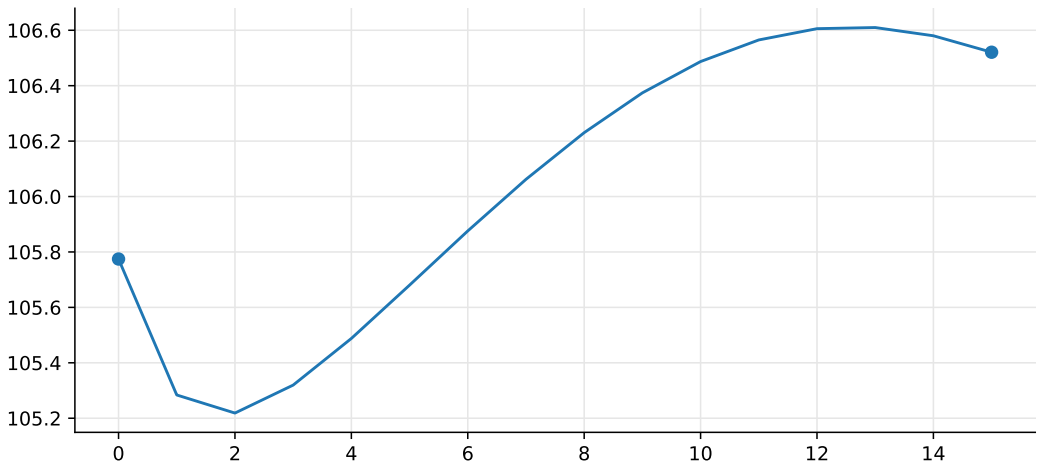
```
prices = []
for q in quotes:
    q.setValue(101.0)
```

```
105.77449623091606
105.2838840874108
105.2838840874108
105.21862925930098
105.21862925930098
105.31959069144436
105.31959069144436
105.48786663523776
105.48786663523776
105.68032070019959
105.68032070019959
105.87580390525216
105.87580390525216
106.06201692440669
106.06201692440669
106.23044635889315
106.23044635889315
106.37409242335241
106.37409242335241
```

```
106.48708841640764
106.48708841640764
106.56505211620107
106.56505211620107
106.60570737717579
106.60570737717579
106.60980192172282
106.60980192172282
106.58011151613943
106.58011151613943
106.52070702691613
```

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.

```
initial_price = prices[0]
updated_prices = prices[1::2]
ax = plt.figure(figsize=(9, 4)).add_subplot(1, 1, 1)
(p,) = ax.plot([initial_price] + updated_prices, "-")
ax.plot(0, initial_price, "o", color=p.get_color())
ax.plot(len(updated_prices), updated_prices[-1], "o", color=p.get_color());
```



The first value is the original bond price, and the last value is the final price after all the quotes were updated; but all the prices in between were calculated based on an incomplete set of changes in which some quotes were updated and some others weren't. They are all incorrect, and (since they went both above and below the range of the real prices) also

outright dangerous: in case your application defined any triggers on price levels, they might have fired incorrectly. 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.

```
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.

```
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.

```
bond.unfreeze()
```

```
106.85839342065753
```

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, and if you have to keep track of your bonds and call `freeze` and `unfreeze` you might as well ask them for their new price instead. 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, 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.

```
del o
```

Our application code knows when a set of changes is published together (reflecting a change of the whole market subset we're tracking). Therefore, it should apply all the changes, and only afterwards it should send some kind of application-level notification that your bonds should be asked for new prices. The ones that depend on the changed quotes will have received their notifications, and will recalculate when asked for a price; other bonds that don't depend on the changed quotes won't have received any notifications and won't recalculate.

```
for q in quotes:
    q.setValue(101.0)

bond.cleanPrice()
```

```
106.52070699239374
```

Chapter 10

Term structures and their reference dates

Independently of their specific type (interest rates, volatility, inflation or whatnot) the term structures implemented in the library share the management of their reference date; in particular, they can be set up so that they track (or don't track) the global evaluation date. This notebook shows an example of this.

Let's first 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.

```
import QuantLib as ql
```

```
ql.Settings.instance().evaluationDate = ql.Date(3, ql.October, 2024)
```

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. When we create a term structure, we have two possibilities to specify its reference date—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 simple flat interest-rate curve: note the `0` and `TARGET()` arguments, specifying the number of days and the calendar used to determine business days.

```
r = ql.SimpleQuote(0.05)
```

```
curve1 = ql.FlatForward(0, ql.TARGET(), ql.QuoteHandle(r), ql.Actual360())
```

The second possibility is to specify the reference date explicitly; in the cell below I use a second overload of the `FlatForward` class takes a given date instead of the reference days and calendar. By pass the same date we set as the current evaluation date (October 3rd) we're creating a curve which is the same as the first—at least for the time being.

```
curve2 = ql.FlatForward(
    ql.Date(3, ql.October, 2024), ql.QuoteHandle(r), ql.Actual360()
)
```

We can check that both curves have the same reference date…

```
print(curve1.referenceDate())
print(curve2.referenceDate())
```

```
October 3rd, 2024
October 3rd, 2024
```

…and return the same discount factors, whether we ask for a given time (for instance, $t = 5$)…

```
print(curve1.discount(5.0))
print(curve2.discount(5.0))
```

```
0.7788007830714049
0.7788007830714049
```

…or for a given date: for instance, 5 years from the reference date, which will be different from the above because the corresponding time is greater than 5 for the Actual/360 day count convention we passed to the curve.

```
print(curve1.discount(ql.Date(3, ql.October, 2029)))
print(curve2.discount(ql.Date(3, ql.October, 2029)))
```

```
0.7759935186328056
0.7759935186328056
```

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?

```
ql.Settings.instance().evaluationDate = ql.Date(19, ql.September, 2024)
```

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

```
print(curve1.referenceDate())
print(curve2.referenceDate())
```

```
September 19th, 2024
October 3rd, 2024
```

And of course the discount factors have changed, too…

```
print(curve1.discount(5.0))
print(curve2.discount(5.0))
```

```
0.7788007830714049
0.7788007830714049
```

…but only 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 (that is, a given time from the reference) we'll get the same discount factor; in other words, we're asking for the discount factor over five years after the reference date, and that remains the same for a rigid translation of the curve. If we ask for the discount factor at a given date, though, we'll see the effect:

```
print(curve1.discount(ql.Date(3, ql.October, 2029)))
print(curve2.discount(ql.Date(3, ql.October, 2029)))
```

```
0.7744861083592839
0.7759935186328056
```

Notifications

Finally, we can see that the two curves behave differently also with respect to notifications. Let's make two observers: each of them will output a message when they receive a notification.

```
def make_observer(i):
    def say():
        s = "Observer %d notified" % i
        print("-" * len(s))
        print(s)
        print("-" * len(s))

    return ql.Observer(say)
```

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

And now, let's connect each of them to one of 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.

```
obs1.registerWith(curve1)
obs2.registerWith(curve2)
```

To check that notifications work, we can change the value of the quoted rate that we used to build both curves. Both observers should print their messages, and in fact, they do so twice. (That's because the curves inherit from two different base classes, each one doing some work when notified and then forwarding the notification.)

```
r.setValue(0.03)
```

```
-------------------
Observer 2 notified
-------------------
-------------------
Observer 2 notified
-------------------
-------------------
Observer 1 notified
-------------------
-------------------
Observer 1 notified
-------------------
```

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

```
ql.Settings.instance().evaluationDate = ql.Date(23, ql.September, 2024)
```

```
-------------------
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.

Part III

Interest-rate term structures

So far, I mostly used flat interest-rates in my examples; that was for simplicity's sake, since interest rates were not the focus of the previous chapters and I didn't want to put too many irons in the fire.

In the next few chapters, I'll finally cover a few classes that can be used to build interest-rate curves from market data. I'll also show a couple of cases where unexpected effects can occur, or where things can go wrong.

Chapter 11

Fitted bond curves

(Originally published as an article in Wilmott Magazine, September 2023.)

```
import QuantLib as ql

today = ql.Date(21, ql.September, 2021)
ql.Settings.instance().evaluationDate = today
```

Let's say we have quoted prices for a set of bonds and we want to infer a discount curve from them. Each of the tuples below contains the start date, the maturity date, the coupon rate, and the quoted price for a bond. I'll sort them by maturity; this is not required for fitting, but it will make it easier to plot the results. For simplicity, all bonds will have the same frequency, calendar and day-count convention, but that's not a requirement; I might have added the extra information to the data.

```
data = [
    (ql.Date(1, 9, 2012), ql.Date(1, 9, 2049), 3.85, 150.41),
    (ql.Date(1, 9, 2018), ql.Date(1, 9, 2036), 2.25, 115.27),
    (ql.Date(15, 9, 2006), ql.Date(15, 9, 2041), 2.97, 160.42),
    (ql.Date(1, 9, 2017), ql.Date(1, 9, 2024), 3.75, 111.75),
    (ql.Date(1, 11, 2017), ql.Date(1, 11, 2022), 5.5, 106.91),
    (ql.Date(1, 12, 2008), ql.Date(1, 12, 2025), 2.0, 109.07),
    (ql.Date(1, 6, 2020), ql.Date(1, 6, 2027), 2.2, 111.63),
    (ql.Date(1, 8, 2018), ql.Date(1, 8, 2034), 5.0, 149.45),
    (ql.Date(1, 9, 2012), ql.Date(1, 9, 2046), 3.25, 135.1),
    (ql.Date(1, 9, 2015), ql.Date(1, 9, 2040), 5.0, 162.3),
    (ql.Date(1, 2, 2017), ql.Date(1, 2, 2037), 4.0, 140.85),
    (ql.Date(1, 3, 2021), ql.Date(1, 3, 2030), 3.5, 125.21),
```

```
    (ql.Date(1, 3, 2018), ql.Date(1, 3, 2040), 3.1, 128.9),
    (ql.Date(1, 3, 2008), ql.Date(1, 3, 2025), 5.0, 118.43),
    (ql.Date(1, 3, 2010), ql.Date(1, 3, 2047), 2.7, 123.23),
    (ql.Date(1, 6, 2020), ql.Date(1, 6, 2026), 1.6, 107.96),
    (ql.Date(1, 8, 2021), ql.Date(1, 8, 2023), 4.75, 109.65),
    (ql.Date(15, 5, 2021), ql.Date(15, 5, 2025), 1.45, 106.09),
    (ql.Date(1, 12, 2010), ql.Date(1, 12, 2028), 2.8, 117.3),
    (ql.Date(15, 7, 2013), ql.Date(15, 7, 2026), 2.1, 109.93),
    (ql.Date(1, 3, 2013), ql.Date(1, 3, 2048), 3.45, 140.06),
    (ql.Date(15, 9, 2015), ql.Date(15, 9, 2024), 2.53, 113.2),
    (ql.Date(1, 11, 2009), ql.Date(1, 11, 2026), 7.25, 137.49),
    (ql.Date(1, 4, 2016), ql.Date(1, 4, 2030), 1.35, 107.11),
    (ql.Date(1, 9, 2018), ql.Date(1, 9, 2044), 4.75, 163.84),
    (ql.Date(15, 5, 2021), ql.Date(15, 5, 2028), 1.39, 117.72),
    (ql.Date(1, 7, 2017), ql.Date(1, 7, 2024), 1.75, 105.96),
    (ql.Date(15, 9, 2012), ql.Date(15, 9, 2032), 1.33, 123.95),
    (ql.Date(1, 3, 2021), ql.Date(1, 3, 2032), 1.65, 109.41),
    (ql.Date(1, 3, 2020), ql.Date(1, 3, 2024), 4.5, 112.1),
    (ql.Date(15, 9, 2016), ql.Date(15, 9, 2026), 3.52, 124.85),
    (ql.Date(1, 11, 2018), ql.Date(1, 11, 2023), 9.0, 120.68),
    (ql.Date(1, 2, 2016), ql.Date(1, 2, 2028), 2.0, 111.43),
    (ql.Date(1, 8, 2017), ql.Date(1, 8, 2029), 3.0, 119.77),
    (ql.Date(1, 5, 2018), ql.Date(1, 5, 2031), 6.0, 151.21),
    (ql.Date(15, 9, 2017), ql.Date(15, 9, 2023), 3.18, 109.61),
    (ql.Date(15, 11, 2008), ql.Date(15, 11, 2024), 1.45, 105.51),
    (ql.Date(15, 9, 2013), ql.Date(15, 9, 2035), 3.01, 143.25),
    (ql.Date(1, 11, 2005), ql.Date(1, 11, 2027), 6.5, 138.74),
    (ql.Date(1, 8, 2009), ql.Date(1, 8, 2027), 2.05, 111.12),
    (ql.Date(1, 10, 2009), ql.Date(1, 10, 2023), 2.45, 106.02),
    (ql.Date(1, 9, 2019), ql.Date(1, 9, 2028), 4.75, 130.95),
    (ql.Date(1, 12, 2009), ql.Date(1, 12, 2026), 1.25, 106.28),
    (ql.Date(1, 9, 2014), ql.Date(1, 9, 2038), 2.95, 125.9),
    (ql.Date(1, 6, 2006), ql.Date(1, 6, 2025), 1.5, 105.91),
    (ql.Date(1, 3, 2009), ql.Date(1, 3, 2026), 4.5, 120.27),
    (ql.Date(1, 9, 2020), ql.Date(1, 9, 2033), 2.45, 117.91),
    (ql.Date(1, 8, 2007), ql.Date(1, 8, 2039), 5.0, 160.49),
    (ql.Date(15, 9, 2016), ql.Date(15, 9, 2022), 1.45, 102.15),
    (ql.Date(15, 5, 2017), ql.Date(15, 5, 2024), 1.85, 106.18),
    (ql.Date(1, 12, 2008), ql.Date(1, 12, 2024), 2.5, 109.19),
    (ql.Date(1, 11, 2009), ql.Date(1, 11, 2029), 5.25, 138.98),
    (ql.Date(1, 3, 2012), ql.Date(1, 3, 2035), 3.35, 129.69),
```

```
    (ql.Date(1, 5, 2012), ql.Date(1, 5, 2023), 4.5, 108.22),
    (ql.Date(15, 11, 2016), ql.Date(15, 11, 2025), 2.5, 110.5),
    (ql.Date(1, 2, 2017), ql.Date(1, 2, 2033), 5.75, 154.11),
    (ql.Date(1, 3, 2016), ql.Date(1, 3, 2067), 2.8, 122.41),
    (ql.Date(1, 9, 2007), ql.Date(1, 9, 2050), 2.45, 117.71),
    (ql.Date(1, 3, 2021), ql.Date(1, 3, 2036), 1.45, 103.93),
    (ql.Date(1, 7, 2012), ql.Date(1, 7, 2025), 1.85, 107.46),
    (ql.Date(1, 12, 2012), ql.Date(1, 12, 2030), 1.65, 109.51),
    (ql.Date(1, 3, 2007), ql.Date(1, 3, 2041), 1.8, 106.42),
    (ql.Date(1, 9, 2007), ql.Date(1, 9, 2051), 1.7, 98.715),
    (ql.Date(30, 4, 2020), ql.Date(30, 4, 2045), 1.5, 97.746),
    (ql.Date(1, 3, 2020), ql.Date(1, 3, 2072), 2.15, 101.04),
]
```

```
data.sort(key=lambda t: t[1])
```

For each row in the data, we create a coupon schedule based on the given start and maturity date (plus, as I mentioned, a few assumptions about the frequency, calendar and business-day convention) and with the schedule, a corresponding coupon-paying bond. As we loop over the rows, we keep two lists: one contains the bonds themselves, and the other contains so-called bond helpers, which wrap the bonds and their quoted price and will be passed to the curve constructor as part of the objective function of the fit.

```
helpers = []
bonds = []
for start, maturity, coupon, price in data:
    schedule = ql.Schedule(
        start,
        maturity,
        ql.Period(1, ql.Years),
        ql.TARGET(),
        ql.ModifiedFollowing,
        ql.ModifiedFollowing,
        ql.DateGeneration.Backward,
        False,
    )
    bond = ql.FixedRateBond(
        3,
        100.0,
        schedule,
        [coupon / 100.0],
        ql.Actual360(),
```

```
        ql.ModifiedFollowing,
    )
    bonds.append(bond)
    helpers.append(
        ql.BondHelper(ql.QuoteHandle(ql.SimpleQuote(price)), bond)
    )
```

Lastly, we create a pricing engine and set it to all bonds; once we have a fitted discount curve, we'll pass it to its handle, now still empty, and use it to check the resulting bond prices.

```
discount_curve = ql.RelinkableYieldTermStructureHandle()
bond_engine = ql.DiscountingBondEngine(discount_curve)
for b in bonds:
    b.setPricingEngine(bond_engine)
```

Fitting to a few curve models

The library implements a few parametric models such as Nelson-Siegel; in this example we'll also use exponential splines, cubic B splines, and Svensson. The models are stored in a Python dictionary so that they can be easily retrieved based on a tag. A couple of them take parameters, but you'll forgive me if I ignore them for brevity; they are documented in the library.

```
methods = {
    "Nelson/Siegel": ql.NelsonSiegelFitting(),
    "Exp. splines": ql.ExponentialSplinesFitting(True),
    "B splines": ql.CubicBSplinesFitting(
        [
            -30.0,
            -20.0,
            0.0,
            5.0,
            10.0,
            15.0,
            20.0,
            25.0,
            30.0,
            40.0,
            50.0,
        ],
        True,
    ),
```

```
    "Svensson": ql.SvenssonFitting(),
}
```

The fitting is performed by the `FittedBondDiscountCurve` class, which takes a list of bond helpers and the parametric model to calibrate, plus a few other parameters. As I mentioned, each bond helper contains a bond and its quoted price; the fitting process iterates over candidate values for the model parameters, reprices each of the bonds based on the resulting discount factors, and tries to minimize the difference between the resulting prices and the passed quotes.

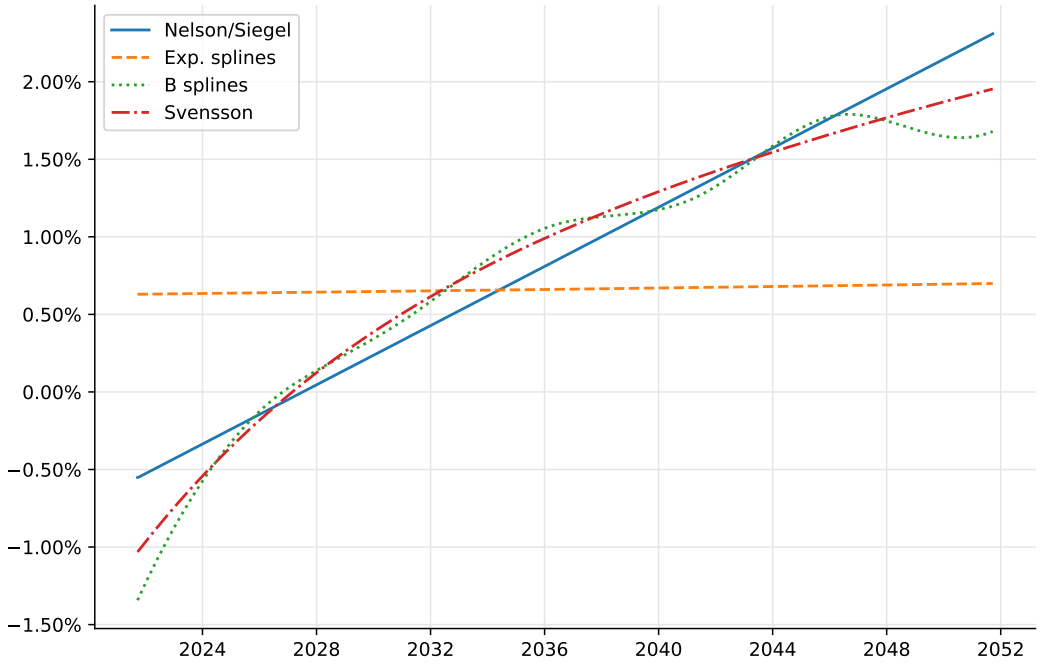
```
tolerance = 1e-8
max_iterations = 5000
day_count = ql.Actual360()

curves = {
    tag: ql.FittedBondDiscountCurve(
        today,
        helpers,
        day_count,
        methods[tag],
        tolerance,
        max_iterations,
    )
    for tag in methods
}
```

Here are the resulting curves:

```
from matplotlib import pyplot as plt
from matplotlib.ticker import PercentFormatter
```

```
dates = [today + ql.Period(i, ql.Months) for i in range(12 * 30 + 1)]
styles = iter(["-", "--", ":", "-."])

ax = plt.figure(figsize=(9, 6)).add_subplot(1, 1, 1)
ax.yaxis.set_major_formatter(PercentFormatter(1.0))
for tag in curves:
    rates = [
        curves[tag].zeroRate(d, day_count, ql.Continuous).rate()
        for d in dates
    ]
    ax.plot(
        [d.to_date() for d in dates],
```

```
        rates,
        next(styles),
        label=tag,
    )
ax.legend(loc="best");
```



The curve are a diverse bunch, to say the least: Nelson-Siegel and Svensson look sensible; exponential splines are almost flat and, compared to the others, look like the fit failed and we got some best-effort result; and cubic B splines are too wavy for my tastes. How can we get a better sense of their quality?

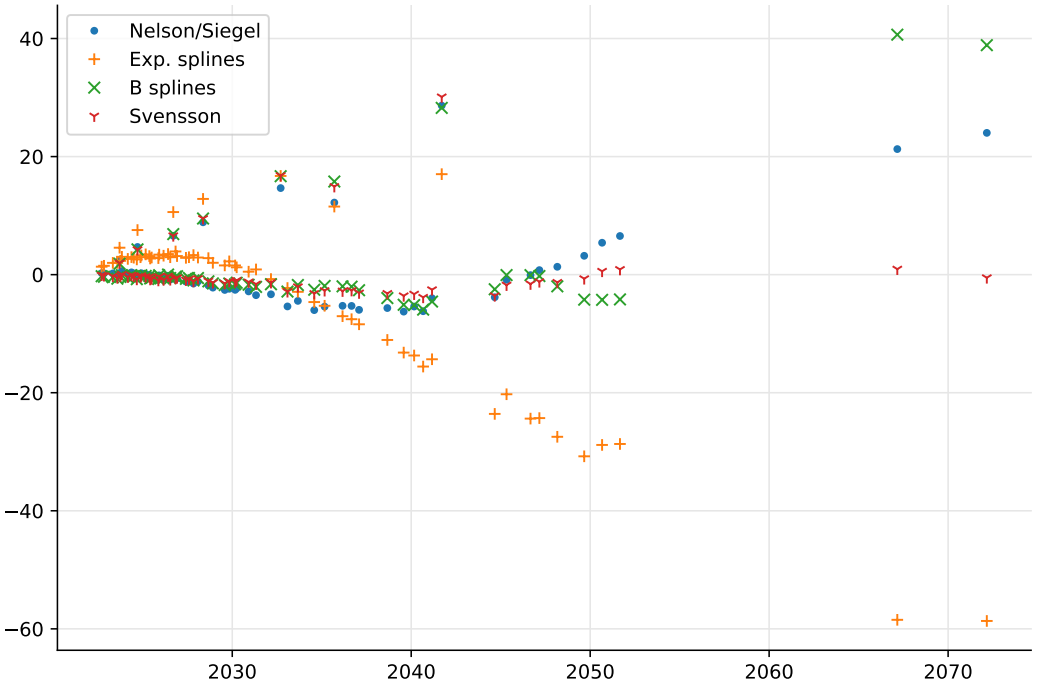
Plotting the prices

With more visualization, of course. The quoted prices can be read from the data; they're the last column. The bond prices implied by any of the curves are also not hard to get—do you remember we created a discount engine and set it to each bond? This pays off now: we can link its discount handle to the desired curve and ask each bond for its price; that's what the `prices` function does. The `error` function calls the latter and returns the difference between calculated and quoted prices.

```
quoted_prices = [row[-1] for row in data]


def prices(tag):
    discount_curve.linkTo(curves[tag])
    return [b.cleanPrice() for b in bonds]


def errors(tag):
    return [q - p for p, q in zip(prices(tag), quoted_prices)]
```

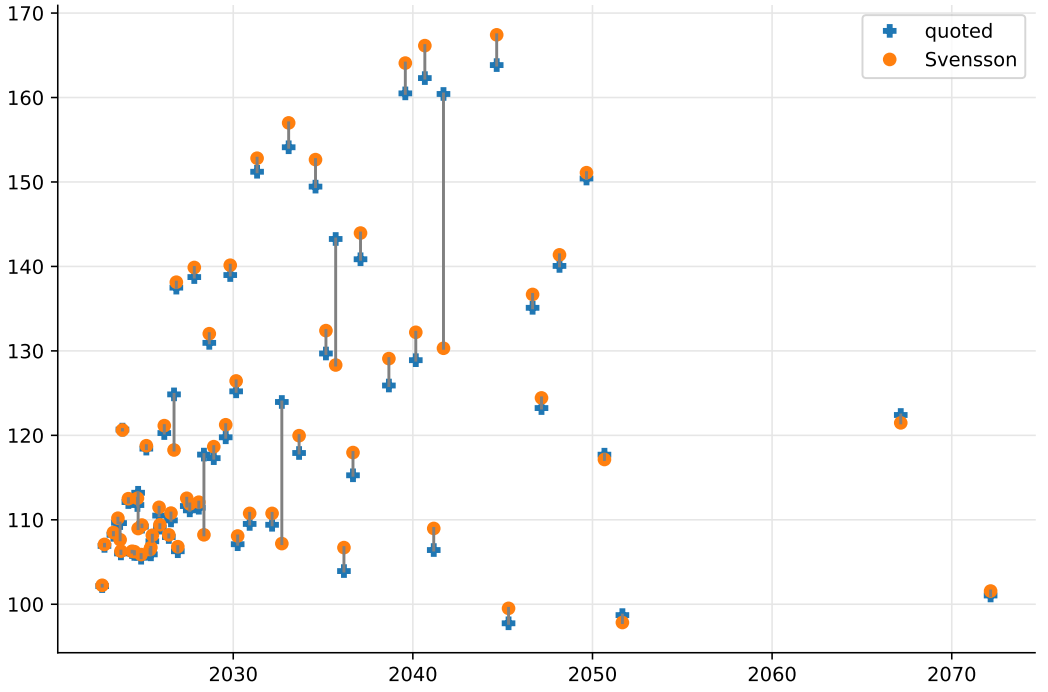
The bit of code that follows the functions plots the errors for each curve as function of the bond maturities. As you might have expected from the previous figure, Nelson-Siegel and Svensson yield smaller errors. Of the two, Svensson looks better in general but especially at longer maturities.

```
ax = plt.figure(figsize=(9, 6)).add_subplot(1, 1, 1)
maturities = [r[1].to_date() for r in data]
markers = iter([".", "+", "x", "1"])
for tag in curves:
    ax.plot(
        maturities,
        errors(tag),
        next(markers),
        label=tag,
    )
ax.legend(loc="best");
```



If we restrict ourselves to just one curve (let's single out Svensson here as the best candidate) we can visualize the calibration errors in yet another way; that's what the next plot does. Instead of the errors, it plots both quoted and calculated prices against maturities. The figure shows a fit which is not great overall, with noticeable errors for most bonds and a few much larger errors that don't seem to follow any particular pattern.

```
ax = plt.figure(figsize=(9, 6)).add_subplot(1, 1, 1)

ps = prices("Svensson")
qs = quoted_prices
ax.plot(maturities, qs, "P", label="quoted")
ax.plot(maturities, ps, "o", label="Svensson")
ax.legend(loc="best")
for m, p, q in zip(maturities, ps, qs):
    ax.plot([m, m], [p, q], "-", color="grey")
```



Another point of view

It's not unusual that, while I bring to the table my inside knowledge about QuantLib, other people in the room are better quants. And when people with different expertise get together, good things usually happen. One time, during a training, I was showing the plot above and one of the trainees said what you, too, might be thinking: "we should probably look at yields."

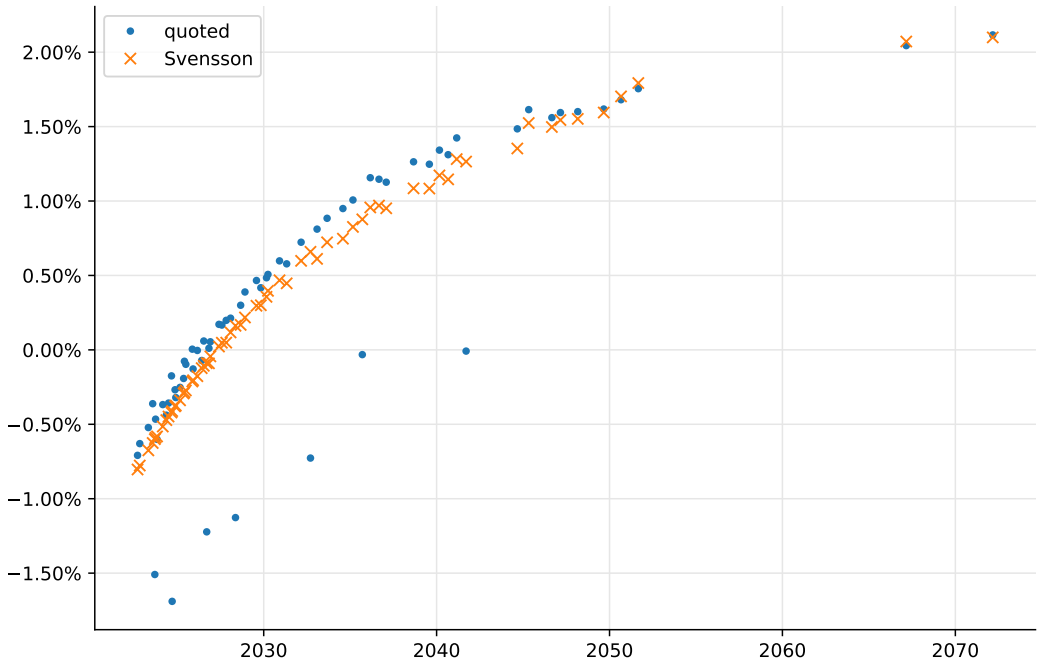
The new plot is not difficult to obtain. The yield corresponding to a quoted price can be calculated by passing it to the `bondYield` method of the corresponding bond; and after linking the desired discount curve, calling the same method without a target price will return the yield corresponding to the calculated price (the method is called simply `yield` in C++, but that's a reserved keyword in Python, so we had to rename it while we export it.)

The result was enlightening. Most of the yields were on some kind of curve, but we had a few obvious outliers and they were pulling the fit away from the rest of the bonds.

```
quoted_yields = [
    b.bondYield(p, day_count, ql.Compounded, ql.Annual)
    for b, p in zip(bonds, quoted_prices)
```

```
]


def yields(tag):
    discount_curve.linkTo(curves[tag])
    return [
        b.bondYield(day_count, ql.Compounded, ql.Annual) for b in bonds
    ]
```

```
ax = plt.figure(figsize=(9, 6)).add_subplot(1, 1, 1)

ax.yaxis.set_major_formatter(PercentFormatter(1.0))
ys = yields("Svensson")
qys = quoted_yields
ax.plot(maturities, qys, ".", label="quoted")
ax.plot(maturities, ys, "x", label="Svensson")
ax.legend(loc="best");
```



Removing the outliers

Fortunately, it's also relatively easy to improve the fit. Given the yields we calculated, we can filter out the helpers whose calculated yield differs by more than 50 basis points from the yield implied by the quoted price. We then create a new curve, passing only the filtered helpers.

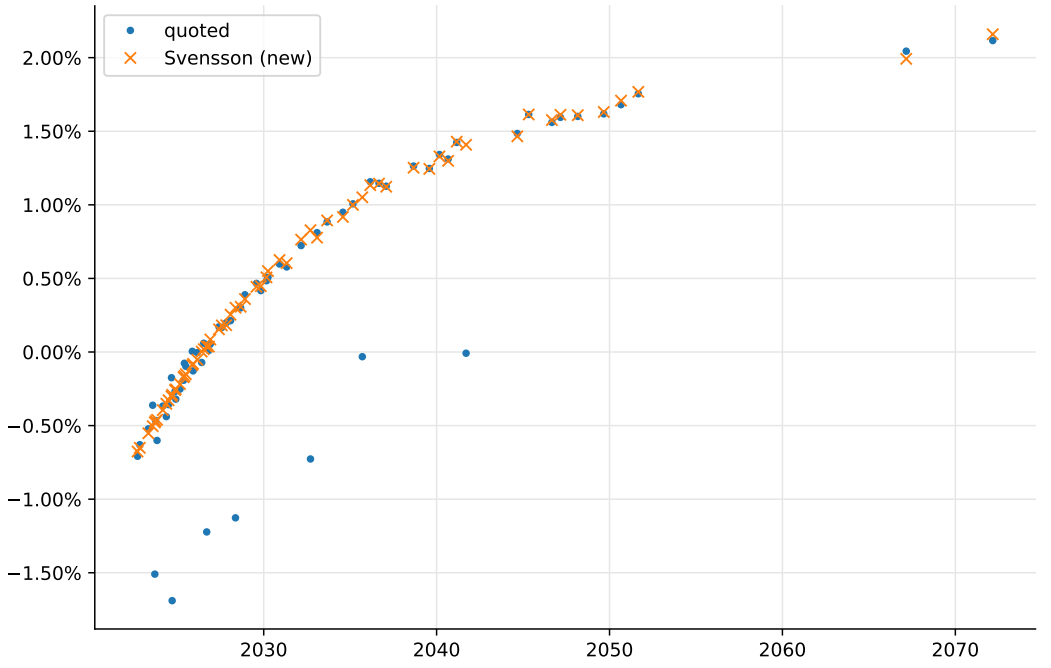
```
ys = yields("Svensson")
qys = quoted_yields

filtered_helpers = [
    h for h, y1, y2 in zip(helpers, ys, qys) if abs(y1 - y2) < 0.005
]
```

```
curves["Svensson (new)"] = ql.FittedBondDiscountCurve(
    today,
    filtered_helpers,
    day_count,
    ql.SvenssonFitting(),
    tolerance,
    max_iterations,
)
```

Improved results

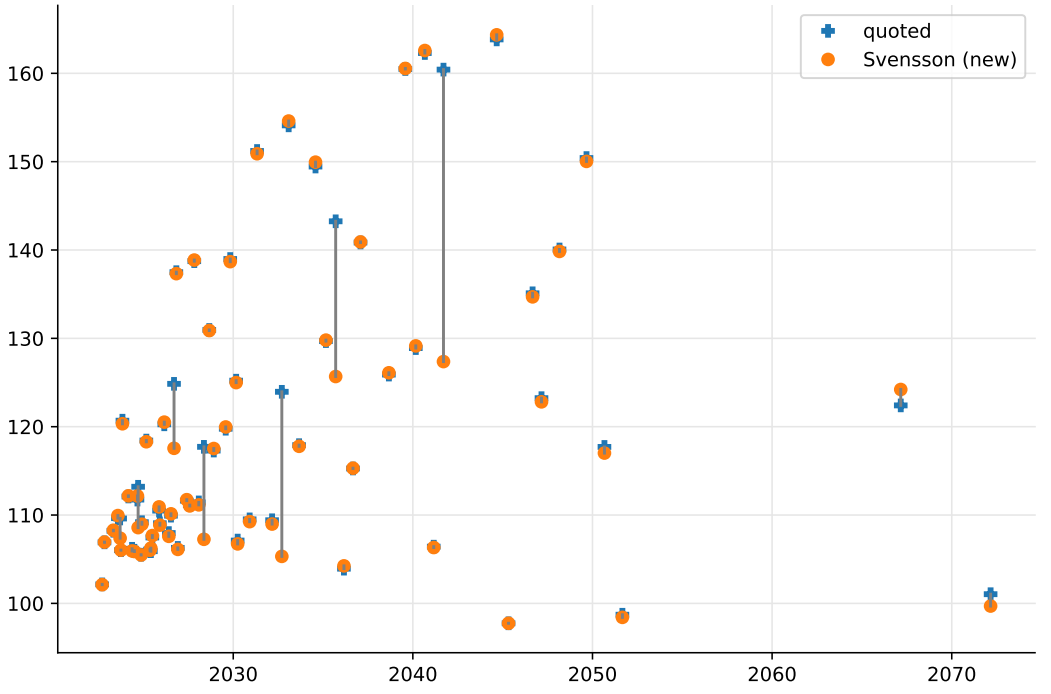
We can now recreate the plots for yields…

```
ax = plt.figure(figsize=(9, 6)).add_subplot(1, 1, 1)

ax.yaxis.set_major_formatter(PercentFormatter(1.0))
ys = yields("Svensson")
ys2 = yields("Svensson (new)")
qys = quoted_yields
ax.plot(maturities, qys, ".", label="quoted")
ax.plot(maturities, ys2, "x", label="Svensson (new)")
ax.legend(loc="best");
```



…and prices; both show a definite improvement of the quality of the fit. The mood of everyone involved in that training was also markedly improved.

```
ax = plt.figure(figsize=(9, 6)).add_subplot(1, 1, 1)

ps = prices("Svensson (new)")
qs = quoted_prices
ax.plot(maturities, qs, "P", label="quoted")
ax.plot(maturities, ps, "o", label="Svensson (new)")
ax.legend(loc="best")
for m, p, q in zip(maturities, ps, qs):
    ax.plot([m, m], [p, q], "-", color="grey")
```


quoted
Svensson (new)
160
150
140
130
120
110
100
2030
2040
2050
2060
2070

Chapter 12

Curve bootstrapping

```
import QuantLib as ql

today = ql.Date(25, ql.October, 2021)
ql.Settings.instance().evaluationDate = today

bps = 1e-4
```

The process of bootstrapping, unlike fitting, uses an iterative algorithm to create a curve that reprices a number of instruments exactly. Each instrument, starting with the one with the shortest maturity, adds a node to the curve keeping the previous ones unchanged until the full curve is built.

When instruments need different curves, we can build them one by one, usually starting from the curve corresponding to the overnight index, e.g., SOFR in a US setting or ESTR in the Euro zone; this is because overnight-based instrument (usually futures and overnight-indexed swaps) only need that one curve, while other instruments need the overnight curve for discounting as well as other curves for forecasting.

In this notebook, I will provide a few short examples. However, there are lots of details that can be added; for a review of those (including, for instance, the possible addition of end-of-year jumps and synthetic instruments) you can read Ametrano and Bianchetti, *Everything You Always Wanted to Know About Multiple Interest Rate Curve Bootstrapping but Were Afraid to Ask*. A much longer notebook reproducing in Python the calculations in that paper is available on my blog.

Finally, *Implementing QuantLib* describes in detail the implementation of the algorithm, including the template machinery that makes it possible in C++ code to choose

whether to interpolate discount factors, zero rates or instantaneous forwards, as well as to choose the kind of interpolation to be used, by declaring curves such as `PiecewiseYieldCurve<Discount, LogLinear>`, `PiecewiseYieldCurve<ZeroYield, Linear>` or `PiecewiseYieldCurve<ForwardRate, BackwardFlat>`.

And now, the examples.

Risk-free curves

These curves are used to forecast risk-free rates such as SOFR, SONIA or ESTR, as well as for risk-free discounting. In this case, I'll use SOFR; of course, we have a class for that.

```
index = ql.Sofr()
```

As I mentioned, the curve is built based on different kinds of instruments. The library defines helper classes corresponding to each of them, each instance acting as an objective function for the solver underlying the bootstrapping process.

For futures, the quote to reproduce is a price…

```
futures_data = [
    ((11, 2021, ql.Monthly), 99.934),
    ((12, 2021, ql.Monthly), 99.922),
    ((1, 2022, ql.Monthly), 99.914),
    ((2, 2022, ql.Monthly), 99.919),
    ((3, 2022, ql.Quarterly), 99.876),
    ((6, 2022, ql.Quarterly), 99.799),
    ((9, 2022, ql.Quarterly), 99.626),
    ((12, 2022, ql.Quarterly), 99.443),
]
```

…while for OIS, the quote is their fixed rate. In the list below, that shows quotes as they might come from some provider, the rates are in percent unit (that is, the 10-years rate is 1.359%); the library will require them in decimal units (i.e., 0.01359 for the same rate.)

```
ois_data = [
    ("1Y", 0.137),
    ("2Y", 0.409),
    ("3Y", 0.674),
    ("5Y", 1.004),
    ("8Y", 1.258),
    ("10Y", 1.359),
    ("12Y", 1.420),
    ("15Y", 1.509),
```

```
    ("20Y", 1.574),
    ("25Y", 1.586),
    ("30Y", 1.579),
    ("35Y", 1.559),
    ("40Y", 1.514),
    ("45Y", 1.446),
    ("50Y", 1.425),
]
```

Quotes are usually wrapped in `SimpleQuote` instances, so that they can be changed later; when this happens, the curve will then be notified and recalculate when needed. Once wrapped, they are passed to the helpers.

```
futures_helpers = []

for (month, year, frequency), price in futures_data:
    q = ql.SimpleQuote(price)
    futures_helpers.append(
        ql.SofrFutureRateHelper(ql.QuoteHandle(q), month, year, frequency)
    )
```

The above uses a specific helper, `SofrFutureRateHelper`, which knows how to calculate the relevant dates of SOFR futures given their month, year and frequency. It also knows that SOFR fixings are compounded in quarterly futures and averaged in monthly ones. For other indexes, for which no specific helper is available, it's possible to use the `OvernightIndexFutureRateHelper` class instead; its constructor requires you to do a bit more work and pass it the start and end date of the underlying fixing period, as well as the averaging method to use.

The helper to use for OIS quotes is `OISRateHelper`; note that, as mentioned, the OIS rate is passed in decimal units, that is, 0.05 if the rate is 5%. In this case, I'm also saving the quotes in a map besides passing them to the helpers, so that I can access them more easily later on.

```
settlement_days = 2

ois_quotes = {}
ois_helpers = []

for tenor, quote in ois_data:
    q = ql.SimpleQuote(quote / 100.0)
    ois_quotes[tenor] = q
    ois_helpers.append(
        ql.OISRateHelper(
```

```
            settlement_days,
            ql.Period(tenor),
            ql.QuoteHandle(q),
            index,
            paymentFrequency=ql.Annual,
        )
    )
```

Another thing to note is that the ranges of futures and OIS can overlap; for instance, in this case, the maturity of the last futures is later than the maturity of the first OIS.

```
futures_helpers[-1].latestDate()
```

```
Date(15,3,2023)
```

```
ois_helpers[0].latestDate()
```

```
Date(27,10,2022)
```

In a real set of data, you would also have OIS quotes for maturities below the year. Using all the available helpers might cause collisions between maturities (which, in turn, would cause the bootstrap to fail); and even if that doesn't happen, having nodes too close might result in unrealistic forwards between them. You'll have to decide what to do: one possibility is to use only futures in the short range (since they're usually more liquid) and start using swaps after the maturity of the last futures, possibly with a bit of buffer space, as in the cell below:

```
cutoff_date = futures_helpers[-1].latestDate() + ql.Period(1, ql.Months)
ois_helpers = [h for h in ois_helpers if h.latestDate() > cutoff_date]
```

Once the helpers are built and chosen, creating the curve is straightforward. There are a number of Python classes available (such as `PiecewiseLogCubicDiscount`, `PiecewiseFlatForward`, `PiecewiseLinearZero` and others), all wrapping the same underlying C++ class template with different arguments.

Creating the class doesn't trigger bootstrapping, so the fact the the constructor succeeds doesn't guarantee that the curve will work. The boostrapping code runs as soon as we call one of the methods of the curve to retrieve its results.

```
sofr_curve = ql.PiecewiseLogCubicDiscount(
    0,
    ql.UnitedStates(ql.UnitedStates.GovernmentBond),
    futures_helpers + ois_helpers,
    ql.Actual360(),
)
```

Depending on the chosen class, the nodes between which the curve interpolates will contain

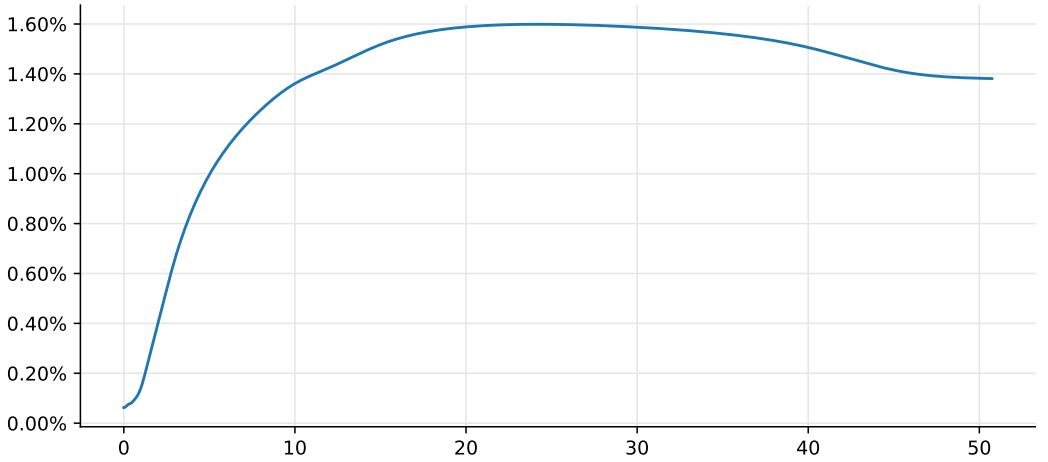
discount factors (as in this case), zero rates, or instantaneous forward rates.

```
sofr_curve.nodes()
```

```
((Date(25,10,2021), 1.0),
 (Date(1,12,2021), 0.9999328324822494),
 (Date(1,1,2022), 0.9998656835848434),
 (Date(1,2,2022), 0.9997916300517078),
 (Date(1,3,2022), 0.9997286453100818),
 (Date(15,6,2022), 0.9993777845072249),
 (Date(21,9,2022), 0.998831257337585),
 (Date(21,12,2022), 0.9978878652374324),
 (Date(15,3,2023), 0.9965926270198491),
 (Date(27,10,2023), 0.9917433028872656),
 (Date(28,10,2024), 0.9796645292460902),
 (Date(27,10,2026), 0.9502361425832937),
 (Date(29,10,2029), 0.9024046595432171),
 (Date(27,10,2031), 0.8704312474038445),
 (Date(27,10,2033), 0.8401724001713865),
 (Date(27,10,2036), 0.7930022596406051),
 (Date(28,10,2041), 0.724132683846159),
 (Date(29,10,2046), 0.666582385788118),
 (Date(27,10,2051), 0.6171451236230561),
 (Date(27,10,2056), 0.5752114363865589),
 (Date(27,10,2061), 0.5447980541075726),
 (Date(27,10,2066), 0.5260997308046376),
 (Date(27,10,2071), 0.4961968121847855))
```

And just for kicks, we can visualize the curve we just created:

```
from matplotlib import pyplot as plt
from matplotlib.ticker import FuncFormatter
import numpy as np

ax = plt.figure(figsize=(9, 4)).add_subplot(1, 1, 1)
ax.yaxis.set_major_formatter(FuncFormatter(lambda r, pos: f"{r:.2%}"))
times = np.linspace(0.0, sofr_curve.maxTime(), 2500)
rates = [sofr_curve.zeroRate(t, ql.Continuous).rate() for t in times]
ax.plot(times, rates);
```

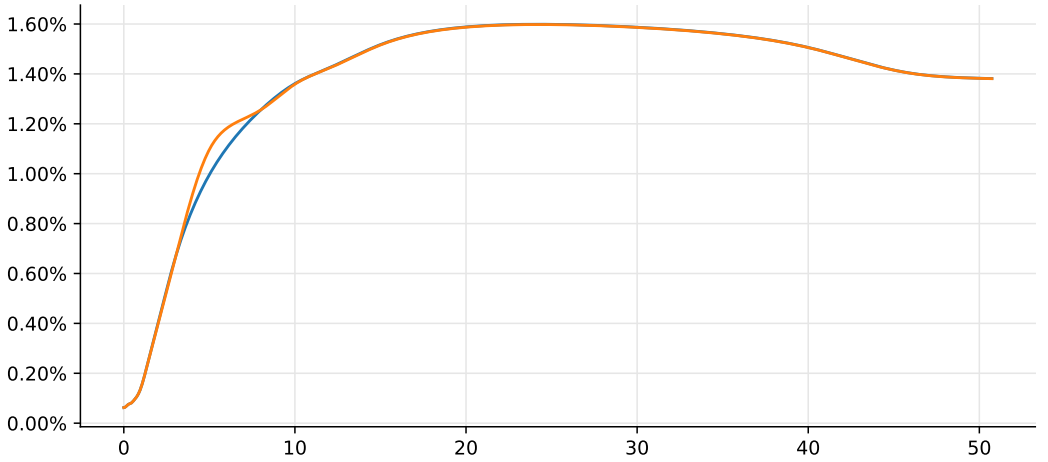


If you kept hold of the quotes like I did above, it's also possible to modify any or all of them when market quotes change, or when you want to see the effect of a given change (more on this in another notebook.)

```
current_5y_value = ois_quotes["5Y"].value()
ois_quotes["5Y"].setValue(current_5y_value + 0.0010)

new_rates = [sofr_curve.zeroRate(t, ql.Continuous).rate() for t in times]

ois_quotes["5Y"].setValue(current_5y_value)
```

Here is the modified curve superimposed to the old one:

```
ax = plt.figure(figsize=(9, 4)).add_subplot(1, 1, 1)
ax.yaxis.set_major_formatter(FuncFormatter(lambda r, pos: f"{r:.2%}"))
ax.plot(times, rates, times, new_rates);
```



LIBOR curves

Actual LIBOR indexes were discountinued a while ago, but similar indexes such as Euribor remains; IBOR indexes, in library lingo. IBOR curves can be bootstrapped using quoted instruments based on the corresponding index; for instance, vanilla fixed-vs-floater swap.

```
swap_data = [
    (ql.Period("1y"), 0.417),
    (ql.Period("18m"), 0.662),
    (ql.Period("2y"), 0.863),
    (ql.Period("3y"), 1.185),
    (ql.Period("5y"), 1.443),
    (ql.Period("7y"), 1.606),
    (ql.Period("10y"), 1.711),
    (ql.Period("12y"), 1.742),
    (ql.Period("15y"), 1.804),
    (ql.Period("20y"), 1.856),
    (ql.Period("25y"), 1.896),
    (ql.Period("30y"), 1.932),
    (ql.Period("50y"), 1.924),
]
```

The vanilla-swap helpers use the curve being bootstrapped to forecast IBOR fixings, but according to current practice they take an external discount curve; here, in the interest of brevity, we'll pretend that the SOFR curve we built above was really an ESTR curve. Of course, we should instead create a new one with the correct index, calendar and whatnot.

```
estr_curve = sofr_curve
```

Onwards with the helpers. Vanilla swaps use the 6-months Euribor, so that's the forecast curve we will bootstrap.

```
index = ql.Euribor(ql.Period(6, ql.Months))
settlement_days = 2
calendar = ql.TARGET()
fixed_frequency = ql.Annual
fixed_convention = ql.Unadjusted
fixed_day_count = ql.Thirty360(ql.Thirty360.BondBasis)

discount_handle = ql.YieldTermStructureHandle(estr_curve)

helpers = [
    ql.SwapRateHelper(
        ql.QuoteHandle(ql.SimpleQuote(quote / 100.0)),
        tenor,
        calendar,
        fixed_frequency,
        fixed_convention,
        fixed_day_count,
        index,
        ql.QuoteHandle(),
        ql.Period(0, ql.Days),
        discount_handle,
    )
    for tenor, quote in swap_data
]
```

The curve is built in the same way as the previous one:

```
euribor_6m_curve = ql.PiecewiseLinearZero(
    0, ql.TARGET(), helpers, ql.Actual360()
)
```

Again, we can look at its nodes (which, this time, contain zero rates…)

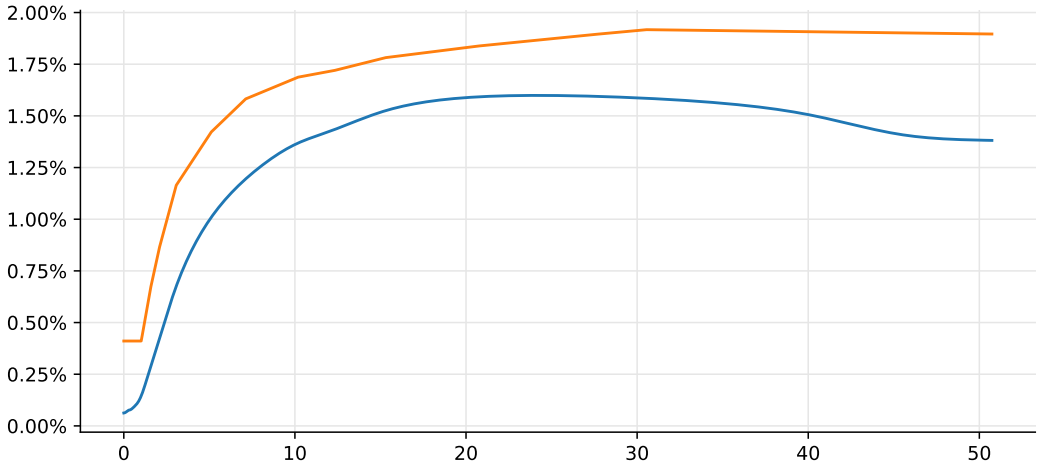
```
euribor_6m_curve.nodes()
```

```
((Date(25,10,2021), 0.004106452138928857),
 (Date(27,10,2022), 0.004106452138928857),
 (Date(27,4,2023), 0.006512076444139096),
 (Date(27,10,2023), 0.00847857348305559),
```

```
 (Date(28,10,2024), 0.01162596400683961),
 (Date(27,10,2026), 0.014192245098651954),
 (Date(27,10,2028), 0.01581512271598231),
 (Date(27,10,2031), 0.016867161885317523),
 (Date(27,10,2033), 0.017169256900638388),
 (Date(27,10,2036), 0.017809443151813677),
 (Date(28,10,2041), 0.01834390600999102),
 (Date(29,10,2046), 0.018769304479864753),
 (Date(27,10,2051), 0.01917190117115746),
 (Date(27,10,2071), 0.018960704667728895))
```

…and plot it, together with the ESTR curve:

```
ax = plt.figure(figsize=(9, 4)).add_subplot(1, 1, 1)
ax.yaxis.set_major_formatter(FuncFormatter(lambda r, pos: f"{r:.2%}"))
euribor_rates = [
    euribor_6m_curve.zeroRate(t, ql.Continuous).rate() for t in times
]
ax.plot(times, rates, times, euribor_rates);
```



Quoted basis swaps

It is also possible to bootstrap a curve over quoted instruments such as OIS-vs-LIBOR swaps. For instance, we might have a set of quotes like this, in which we're given the fair basis between the ESTR and Euribor legs in swaps with different maturities:

```
basis_data = [
    (ql.Period("1y"), 0.223),
    (ql.Period("18m"), 0.237),
    (ql.Period("2y"), 0.249),
    (ql.Period("3y"), 0.266),
    (ql.Period("5y"), 0.271),
    (ql.Period("7y"), 0.290),
    (ql.Period("10y"), 0.308),
    (ql.Period("12y"), 0.309),
    (ql.Period("15y"), 0.315),
    (ql.Period("20y"), 0.321),
    (ql.Period("25y"), 0.334),
    (ql.Period("30y"), 0.341),
    (ql.Period("50y"), 0.355),
]
```

In this case, we need to pass to the helpers (among other parameters) the ESTR index for the first leg, which needs to contain a forecast handle set to the ESTR curve…

```
estr = ql.Estr(ql.YieldTermStructureHandle(estr_curve))
```

…and the Euribor index whose curve we want to bootstrap:

```
euribor3m = ql.Euribor(ql.Period(3, ql.Months))
```

As before, we'll build a helper for each quote. Some of the parameters are inferred from the passed indexes; for instance, the frequency of the payments (taken to be the same as the tenor of the LIBOR) and the day-count conventions, also taken for each coupon from the corresponding index. It is also possible to pass a discount curve, but if not, it is assumed to be the curve set to the overnight index.

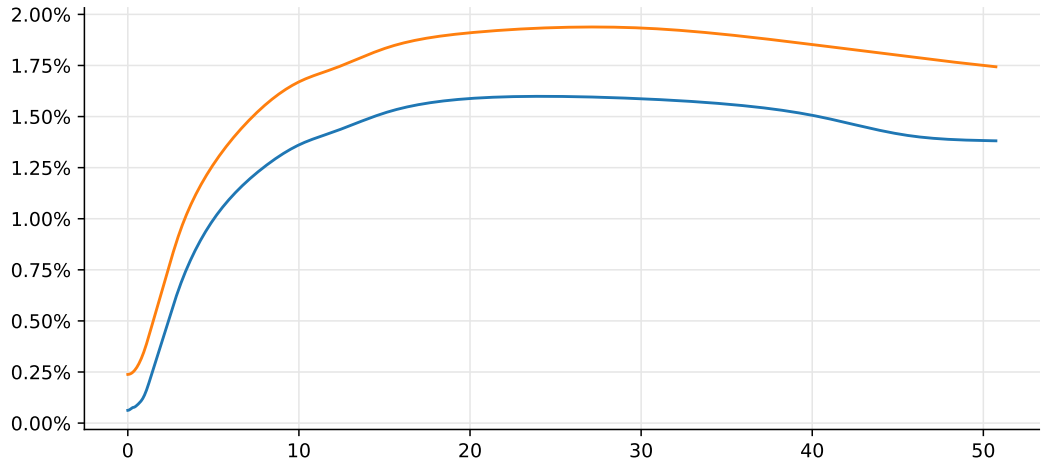
```
settlement_days = 2
calendar = ql.TARGET()
convention = ql.ModifiedFollowing
end_of_month = False

helpers = [
    ql.OvernightIborBasisSwapRateHelper(
        ql.QuoteHandle(ql.SimpleQuote(quote / 100.0)),
        tenor,
        settlement_days,
        calendar,
        convention,
        end_of_month,
```

```
        estr,
        euribor3m,
    )
    for tenor, quote in basis_data
]
```

The curve is built as before:

```
euribor_3m_curve = ql.PiecewiseLogCubicDiscount(
    0, ql.TARGET(), helpers, ql.Actual360()
)
```

```
ax = plt.figure(figsize=(9, 4)).add_subplot(1, 1, 1)
ax.yaxis.set_major_formatter(FuncFormatter(lambda r, pos: f"{r:.2%}"))
euribor_rates = [
    euribor_3m_curve.zeroRate(t, ql.Continuous).rate() for t in times
]
ax.plot(times, rates, times, euribor_rates);
```



It is also possible to bootstrap a curve over a mix of vanilla and basis swaps, provided that they're based on the same IBOR index.

Swap pricing

Once the relevant curves are built, it's possible to price swaps; for instance, a fixed-vs-Euribor swap. First, we build the Euribor index and pass it the corresponding forecast curve:

```
forecast_handle = ql.YieldTermStructureHandle(euribor_6m_curve)

euribor6m = ql.Euribor(ql.Period(3, ql.Months), forecast_handle)
```

Then we build the swap, passing the conventions and terms it needs:

```
start_date = ql.Date(18, ql.November, 2021)
end_date = start_date + ql.Period(30, ql.Years)
fixed_coupon_tenor = ql.Period(ql.Annual)
calendar = ql.TARGET()
rule = ql.DateGeneration.Forward
float_convention = ql.ModifiedFollowing
end_of_month = False
fixed_rate = 150 * bps
floater_spread = 10 * bps

fixed_schedule = ql.Schedule(
    start_date,
    end_date,
    fixed_coupon_tenor,
    calendar,
    fixed_convention,
    fixed_convention,
    rule,
    end_of_month,
)
float_schedule = ql.Schedule(
    start_date,
    end_date,
    euribor6m.tenor(),
    calendar,
    float_convention,
    float_convention,
    rule,
    end_of_month,
)
swap = ql.VanillaSwap(
    ql.Swap.Payer,
    1_000_000,
    fixed_schedule,
    fixed_rate,
    fixed_day_count,
```

```
    float_schedule,
    euribor6m,
    floater_spread,
    euribor6m.dayCounter(),
)
```

Finally, we tell its pricing engine to use the ESTR curve for discounting:

```
swap.setPricingEngine(ql.DiscountingSwapEngine(discount_handle))
```

We can now ask the swap for its value, or other figures:

```
swap.NPV()
```

```
128295.50732521049
```

```
swap.fairRate()
```

```
0.02037361224452942
```

Chapter 13

The effect of today's fixing on bootstrapped interest rates

(Based on a question by Steve Hsieh on the QuantLib mailing list and an issue by Marcin Rybacki on GitHub. Thanks!)

The purpose of this notebook is to highlight an effect that might not be obvious.

```
import QuantLib as ql
import pandas as pd

today = ql.Date(23, ql.January, 2024)
ql.Settings.instance().evaluationDate = today
ql.IborCoupon.createIndexedCoupons()
```

Setting up

Let's say we have the usual dual-curve setup for pricing fixed vs floater swaps. I'll gloss over the mechanics of creating the discount and forecast curves; it's described elsewhere.

For brevity, I'll use just a handful of OIS to create a sample discount curve.

```
helpers = [
    ql.OISRateHelper(
        2,
        tenor,
        ql.QuoteHandle(ql.SimpleQuote(quote / 100.0)),
```

```
        ql.Estr(),
        paymentFrequency=ql.Annual,
    )
    for tenor, quote in [
        (ql.Period("1y"), 3.995),
        (ql.Period("5y"), 4.135),
        (ql.Period("10y"), 4.372),
        (ql.Period("20y"), 4.798),
    ]
]

discount_curve = ql.PiecewiseLogCubicDiscount(
    0, ql.TARGET(), helpers, ql.Actual360()
)

discount_handle = ql.YieldTermStructureHandle(discount_curve)
```

Next, we create the forecast curve for the floating index; in this case, 6-months Euribor.

```
quoted_swap_data = [
    (ql.Period("1y"), 3.96),
    (ql.Period("2y"), 4.001),
    (ql.Period("3y"), 4.055),
    (ql.Period("5y"), 4.175),
    (ql.Period("7y"), 4.304),
    (ql.Period("10y"), 4.499),
    (ql.Period("12y"), 4.611),
    (ql.Period("15y"), 4.741),
    (ql.Period("20y"), 4.846),
]
```

```
index = ql.Euribor(ql.Period(6, ql.Months))
settlement_days = 2
calendar = ql.TARGET()
fixed_frequency = ql.Annual
fixed_convention = ql.Unadjusted
fixed_day_count = ql.Thirty360(ql.Thirty360.BondBasis)

helpers = [
    ql.SwapRateHelper(
        ql.QuoteHandle(ql.SimpleQuote(quote / 100.0)),
        tenor,
```

```
        calendar,
        fixed_frequency,
        fixed_convention,
        fixed_day_count,
        index,
        ql.QuoteHandle(),
        ql.Period(0, ql.Days),
        discount_handle,
    )
    for tenor, quote in quoted_swap_data
]

euribor_curve = ql.PiecewiseFlatForward(
    0, ql.TARGET(), helpers, ql.Actual360()
)
```

Here are the resulting forward rates:

```
df = pd.DataFrame(euribor_curve.nodes(), columns=["date", "rate"])
df.style.format({"rate": "{:.6%}"})
```

	date	rate
0	January 23rd, 2024	3.797969%
1	January 27th, 2025	3.797969%
2	January 26th, 2026	3.919519%
3	January 27th, 2027	4.039856%
4	January 25th, 2029	4.218121%
5	January 27th, 2031	4.499489%
6	January 25th, 2034	4.884324%
7	January 25th, 2036	5.149876%
8	January 26th, 2039	5.275125%
9	January 27th, 2044	5.163086%

We're now able to create an instance of `Euribor6M` that can forecast future fixings—or today's fixing, if we don't have already stored it in the library.

```
euribor_handle = ql.YieldTermStructureHandle(euribor_curve)

euribor = ql.Euribor6M(euribor_handle)
```

```
euribor.fixing(today)
```

```
0.03834665129363748
```

Pricing a sample swap

Now I'll create a sample swap and price it using the discount and forecast curves I created. I'll have it start in the past, so I'll have to store the fixing for the current coupon (which was in the past and can't be read off the forecast curve.)

```
start_date = today - ql.Period(21, ql.Months)
end_date = start_date + ql.Period(15, ql.Years)

fixed_schedule = ql.Schedule(
    start_date,
    end_date,
    ql.Period(fixed_frequency),
    calendar,
    fixed_convention,
    fixed_convention,
    ql.DateGeneration.Forward,
    False,
)
float_schedule = ql.Schedule(
    start_date,
    end_date,
    euribor.tenor(),
    calendar,
    euribor.businessDayConvention(),
    euribor.businessDayConvention(),
    ql.DateGeneration.Forward,
    False,
)
swap = ql.VanillaSwap(
    ql.Swap.Payer,
    1_000_000,
    fixed_schedule,
    0.04,
    fixed_day_count,
    float_schedule,
    euribor,
```

```
    0.0,
    euribor.dayCounter(),
)
swap.setPricingEngine(ql.DiscountingSwapEngine(discount_handle))

euribor.addFixing(ql.Date(19, 10, 2023), 0.0413)

swap.NPV()
```

```
47643.03343425237
```

A surprising effect

Now, if we're pricing a number of swaps with different schedules, it's not very convenient to figure out what past fixings we need to store. It's easier to store the whole history of the index for the past year or so—and if we already have it in our systems, we'll probably add today's fixing as well.

```
euribor.addFixing(today, 0.0394)
```

At this point, if we ask the index for today's fixing, it will return the stored value.

```
euribor.fixing(today)
```

```
0.0394
```

(A note: the full signature of the `fixing` method is

```
Real fixing(const Date& fixingDate, bool forecastTodaysFixing = false)
```

so we can still read the corresponding rate off the curve, if we need it for comparison.)

```
euribor.fixing(today, True)
```

```
0.03729528572216041
```

Our sample swap, though, doesn't use today's fixing to determine its coupons, so its price shouldn't change—right?

```
swap.NPV()
```

```
46857.318298378086
```

What happened?

True, the swap doesn't use today's fixing directly. But storing it causes the forecast curve to recalculate, because it is used by the quoted swaps over which we're bootstrapping it. *Their* first coupon is now determined, and the rest of the curve has to change so that their fair rate still corresponds to the quoted one.

To check this, we can ask the curve for its nodes again and compare the result with what we have already stored in the data frame:

```
df["new rate"] = [r for n, r in euribor_curve.nodes()]
df.style.format({"rate": "{:.6%}", "new rate": "{:.6%}"})
```

	date	rate	new rate
0	January 23rd, 2024	3.797969%	3.694805%
1	January 27th, 2025	3.797969%	3.694805%
2	January 26th, 2026	3.919519%	3.919519%
3	January 27th, 2027	4.039856%	4.039856%
4	January 25th, 2029	4.218121%	4.218121%
5	January 27th, 2031	4.499489%	4.499489%
6	January 25th, 2034	4.884324%	4.884324%
7	January 25th, 2036	5.149876%	5.149876%
8	January 26th, 2039	5.275125%	5.275125%
9	January 27th, 2044	5.163086%	5.163086%

We can see the difference in the first year of the curve. The rates from the second year onwards are not modified; intuitively, that's because both the old and the new curve, by construction, give the same value to the first two floating coupons (those corresponding to a 1-year swap) so the rest of the curve doesn't need to change.

However, this change is enough to modify our forecast of the next coupon of our sample swap, and therefore its total NPV.

Is this desirable?

Well, it might be unexpected or confusing at first, but I don't see a use case for not including the fixing effect. On the one hand, once it's available, we can assume that the quoted swap rates make use of the information, and therefore we need it as well; and on the other hand, ignoring the fixing during bootstrapping and including it when pricing would cause quoted swaps to no longer price at 0—which is obviously undesirable. All in all, I think including the effect is the right thing to do.

Chapter 14

Dangerous day-count conventions

This notebook (based on a question by Min Gao on the QuantLib mailing list. Thanks!) shows one possible pitfall in the instantiation of term structures.

```
import QuantLib as ql
from matplotlib import pyplot as plt
```

```
today = ql.Date(22, 1, 2018)
ql.Settings.instance().evaluationDate = today
```

The problem

Talking about term structures in *Implementing QuantLib*, I mention that they need to be passed a day-count convention and I suggest to use simple ones such as Actual/360 or Actual/365. 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 all day counters. The property holds for most dates…

```
dc = ql.Thirty360(ql.Thirty360.USA)
T = dc.yearFraction
```

```
d1 = ql.Date(1, ql.January, 2018)
d2 = ql.Date(15, ql.January, 2018)
```

```
d3 = ql.Date(31, ql.January, 2018)
```

```
print(T(d1, d2) + T(d2, d3))
print(T(d1, d3))
```

```
0.08333333333333334
0.08333333333333333
```

…but doesn't for some.

```
d1 = ql.Date(1, ql.January, 2018)
d2 = ql.Date(30, ql.January, 2018)
d3 = ql.Date(31, ql.January, 2018)
```

```
print(T(d1, d2) + T(d2, d3))
print(T(d1, d3))
```

```
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:

```
d1 = ql.Date(30, ql.January, 2018)
d2 = ql.Date(31, ql.January, 2018)

print(T(d1, d2))
```

```
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. In fact, it seemed like a natural thing to do. However, this convention needs to be given a reference period for its calculations, as well as the two dates whose distance one needs to measure; failing to do so will result in the wrong results…

```
d1 = ql.Date(1, ql.January, 2018)
d2 = ql.Date(15, ql.January, 2018)

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

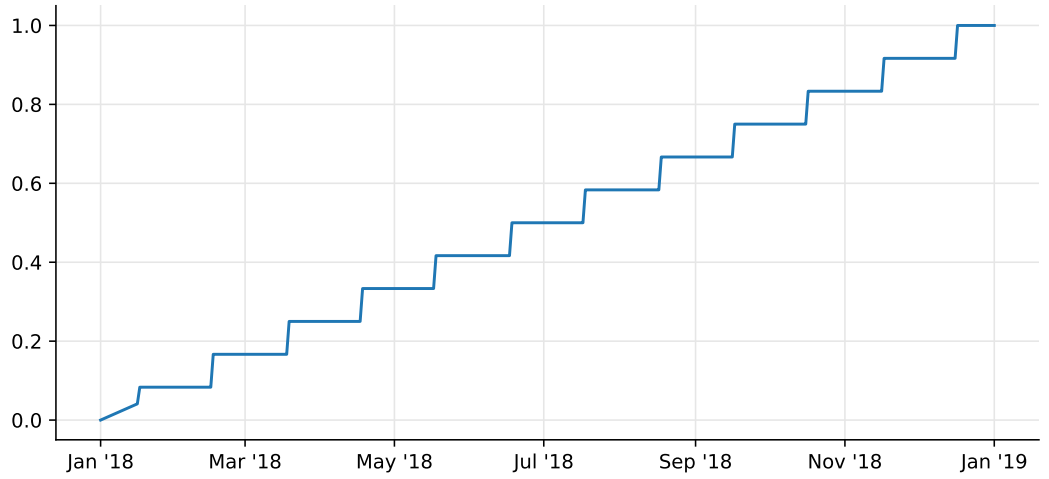
```
dc = ql.ActualActual(ql.ActualActual.ISMA)

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

```
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.

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

```
ax = plt.figure(figsize=(9, 4)).add_subplot(1, 1, 1)
ax.plot([d.to_date() 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.

```
curve = ql.FlatForward(today, 0.01, ql.ActualActual(ql.ActualActual.ISMA))
```

```
dates = [(today + i) for i in range(366)]
discounts = [curve.discount(d) for d in dates]
ax = plt.figure(figsize=(9, 4)).add_subplot(1, 1, 1)
ax.plot([d.to_date() for d in dates], discounts, "-");
```

Why did we allow to use this convention without passing the reference period? Well, we couldn't make it non optional in the call to `yearFraction`, because that would have made its interface different from what is declared in the base `DayCounter` class. Making it non-optional in the base-class method would have been inconvenient for all other convention that don't need it. At the time, we probably didn't think (memories of the early times of the library are quickly fading) of keeping the common interface and raising an error for this convention if the reference dates weren't passed. I'm not sure we can safely make that change now.

Any solutions?

Not really, at least not in a general way. For a specific bond, it is possible to store a schedule inside an ISMA actual/actual day counter and use it to retrieve the correct reference period for dates within a coupon:

```
schedule = ql.MakeSchedule(
    effectiveDate=ql.Date(1, ql.January, 2018),
    terminationDate=ql.Date(1, ql.January, 2025),
    frequency=ql.Annual,
)
dc = ql.ActualActual(ql.ActualActual.ISMA, schedule)
```

This way, the calculations seem to be consistent…

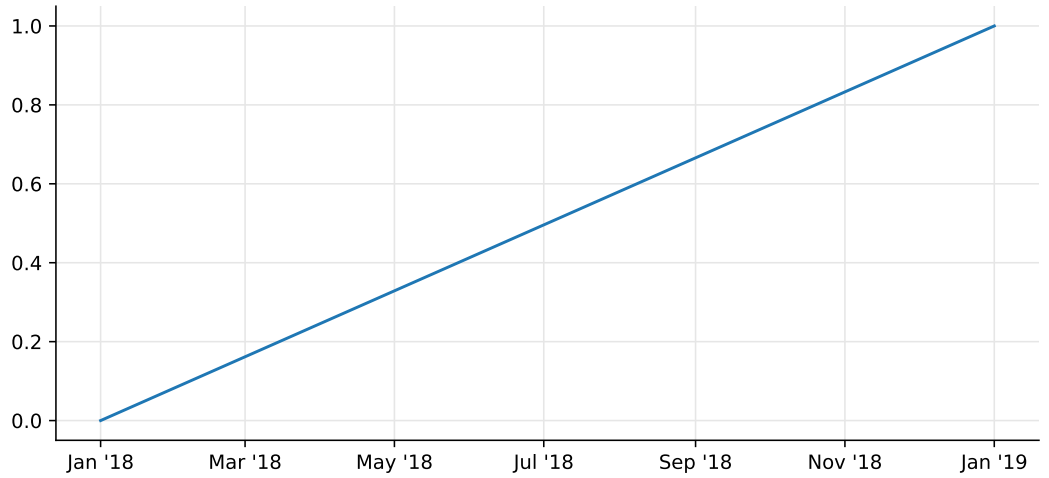
```
d1 = ql.Date(1, ql.January, 2018)
d2 = ql.Date(15, ql.January, 2018)
```

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

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

```
0.038356164383561646
0.038356164383561646
```

…and conversion from dates to times seems to be reasonable.

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

ax = plt.figure(figsize=(9, 4)).add_subplot(1, 1, 1)
ax.plot([d.to_date() for d in dates], times, "-");
```



However, that's probably not the best choice for a curve. The calculation is bound to a specific bond, which is not great if you want to use the curve as a generic one for multiple instruments. Also, the calculation only works within the range of the schedule, and will give you an error if you try to go outside them:

```
try:
    dc.yearFraction(d1, schedule[-1] + ql.Period(10, ql.Days))
```

```
except Exception as e:
    print(f"Error: {e}")
```

```
Error: Dates out of range of schedule: date 1: January 1st, 2018, date 2: January 11th, 202
```

Therefore, I stand by my suggestion. When creating a bond, of course, you must use its specified day-count convention; but, unless something prevents it, use a simple day-count convention such as actual/360 or actual/365 for term structures. You can use a day-count convention bound to a particular schedule in some specific cases; for instance, calculating the yield of a specific bond.

Part IV

More interest-rate instruments

After having seen how we can build interest-rate term structures, we can expand our catalog of interest-rate instruments beyond simple bonds.

For brevity, though, I might still use flat or otherwise made-up interest-rate curves. Please don't hold this against me.

Chapter 15

Vanilla bonds

```
import QuantLib as ql
import pandas as pd

today = ql.Date(2, ql.May, 2024)
ql.Settings.instance().evaluationDate = today
```

We've already seen in a previous notebook how to instantiate a fixed-rate bond. Here we expand on that: we'll see a number of possible calculations, as well as other kinds of bonds that can be created and priced. Let's start again from a sample fixed-rate bond and see what we can do.

```
schedule = ql.Schedule(
    ql.Date(8, ql.February, 2023),
    ql.Date(8, ql.February, 2028),
    ql.Period(6, ql.Months),
    ql.TARGET(),
    ql.Following,
    ql.Following,
    ql.DateGeneration.Backward,
    False,
)
settlementDays = 3
faceAmount = 10_000
coupons = [0.03]
paymentDayCounter = ql.Thirty360(ql.Thirty360.BondBasis)
```

```
bond = ql.FixedRateBond(
    settlementDays, faceAmount, schedule, coupons, paymentDayCounter
)
```

Static information

First of all, even without using an engine or other market data, the bond is able to return some static information. We can ask it for its current settlement date, or for the settlement date corresponding to another evaluation date:

```
print(bond.settlementDate())
```

```
May 7th, 2024
```

```
print(bond.settlementDate(ql.Date(31, ql.May, 2027)))
```

```
June 3rd, 2027
```

The same goes for the amount accrued so far: we can call the corresponding method without a date, returning the amount accrued up to the current settlement date, or pass another date. Note that the accrued amount, like prices, is scaled based on a notional of 100, not the actual face amount of the bond.

```
print(bond.accruedAmount())
```

```
0.7416666666666627
```

```
print(bond.accruedAmount(ql.Date(9, ql.May, 2027)))
```

```
0.7583333333333275
```

In this case, the passed date is treated as a settlement date; we can verify that this is the case by passing the current settlement date and checking that the returned amount equals the one returned by the call without date.

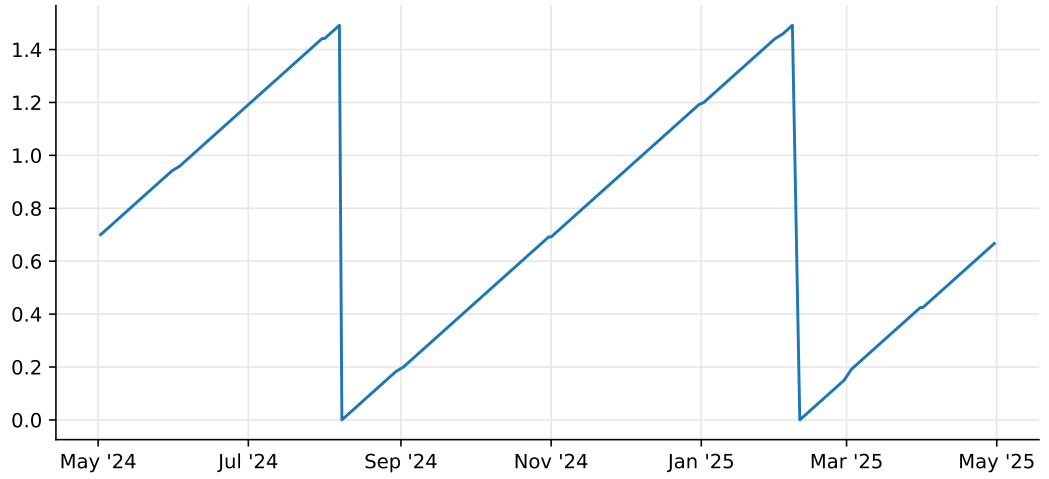
```
print(bond.accruedAmount(bond.settlementDate()))
```

```
0.7416666666666627
```

And for a further check, we can cycle over the next year and visualize the accrued amount at each date, verifying that it makes sense; that is, that it increases up to the full coupon amount and then resets to zero semiannually (the frequency of the bond we created.)

```
dates = [
    today + i for i in range(365) if ql.TARGET().isBusinessDay(today + i)
]
```

```
accruals = [bond.accruedAmount(d) for d in dates]

from matplotlib import pyplot as plt

ax = plt.figure(figsize=(9, 4)).add_subplot(1, 1, 1)
ax.plot([d.to_date() for d in dates], accruals, "-");
```



Cash flows

Again, we've already seen that we can perform cash-flow analysis on the bond. As a quick recap, the `cashflows` method returns a sequence of cash flows, each one providing basic information. In C++, the return type would be `std::vector<shared_ptr<CashFlow>>`, exported to Python as a list or tuple of `CashFlow` instances; the latter is the base class for different kinds of coupons and other cash flows. Its interface contains little more than the amount of the cash flow and the date when it occurs; other information (such as, for instance, the rate paid by the bond coupons) is available from derived classes. The amounts are returned based on the actual face amount of the bond.

```
pd.DataFrame(
    [(c.date(), c.amount()) for c in bond.cashflows()],
    columns=("date", "amount"),
    index=range(1, len(bond.cashflows()) + 1),
).style.format({"amount": "{:.2f}"})
```

	date	amount
1	August 8th, 2023	150.00
2	February 8th, 2024	150.00
3	August 8th, 2024	150.00
4	February 10th, 2025	151.67
5	August 8th, 2025	148.33
6	February 9th, 2026	150.83
7	August 10th, 2026	150.83
8	February 8th, 2027	148.33
9	August 9th, 2027	150.83
10	February 8th, 2028	149.17
11	February 8th, 2028	10000.00

As you can see, the returned cash flows include both the coupon and the final redemption (distinct from the final coupon paid on the same date.)

Access to more detailed information requires to downcast the `CashFlow` instances. In C++, this would be done as

```
auto cfs = bond->cashflows();
auto coupon = ext::dynamic_pointer_cast<FixedRateCoupon>(cfs[0]);
```

but Python doesn't have the concept of casting from one type to another. Therefore, the QuantLib bindings provide functions such as `as_fixed_rate_coupon` or `as_floating_rate_coupon` that perform the same operation. If the cast doesn't succeed, the returned value is a null pointer in C++ or `None` in Python. This gives us access to the methods defined in derived classes: we can use it to perform a more detailed cash-flow analysis, returning coupon-specific information when the cast succeeds and only basic information when it fails, as in the case of the bond redemptiom:

```
def coupon_info(cf):
    c = ql.as_coupon(cf)
    if not c:
        return (cf.date(), None, None, cf.amount())
    else:
        return (c.date(), c.rate(), c.accrualPeriod(), c.amount())


pd.DataFrame(
    [coupon_info(c) for c in bond.cashflows()],
    columns=("date", "rate", "accrual time", "amount"),
    index=range(1, len(bond.cashflows()) + 1),
```

```
).style.format({"amount": "{:.2f}", "rate": "{:.2%}"})
```

	date	rate	accrual time	amount
1	August 8th, 2023	3.00%	0.500000	150.00
2	February 8th, 2024	3.00%	0.500000	150.00
3	August 8th, 2024	3.00%	0.500000	150.00
4	February 10th, 2025	3.00%	0.505556	151.67
5	August 8th, 2025	3.00%	0.494444	148.33
6	February 9th, 2026	3.00%	0.502778	150.83
7	August 10th, 2026	3.00%	0.502778	150.83
8	February 8th, 2027	3.00%	0.494444	148.33
9	August 9th, 2027	3.00%	0.502778	150.83
10	February 8th, 2028	3.00%	0.497222	149.17
11	February 8th, 2028	nan%	nan	10000.00

Pricing

Calculating the price of the bond requires some market data, since we need to discount its cash flows. One way to do this is to use quoted yield information:

```
bond.cleanPrice(
    0.035,
    ql.Thirty360(ql.Thirty360.BondBasis),
    ql.Compounded,
    ql.Semiannual,
)
```

```
98.25259209893974
```

As I mentioned, the price is returned in base 100. At this time, there's no support for bonds that are quoted in other units, e.g., around 1000 as some markets do.

Another way to calculate the price is to use a discount curve. Here I'll use a simple flat rate for brevity, but any interest-rate term structure can be used; it could be, for instance, a Nelson-Siegel curve fitted on market prices. In this case, we're not passing the discount curve to a bond method, as we did for the yield in the previous cell; instead, we pass it to a pricing engine and set the engine to the bond. This way, the bond will react if the curve changes. Additionally, we could keep hold of the term structure handle and relink it if necessary.

The engine we're using is `DiscountingBondEngine`, which collects the dates and amounts from the cash flows, discounts them accordingly and accumulates them. (For more complex bonds, such as callables or convertibles, we'd need different engines.)

```
discount_curve = ql.FlatForward(0, ql.TARGET(), 0.04, ql.Actual360())
engine = ql.DiscountingBondEngine(
    ql.YieldTermStructureHandle(discount_curve)
)
bond.setPricingEngine(engine)
```

Once the pricing engine is set, we can ask the bond for its price and accrued amount without having to pass other arguments.

```
print(bond.cleanPrice())
print(bond.dirtyPrice())
print(bond.accruedAmount())
```

```
96.19296369916307
96.93463036582973
0.7416666666666627
```

The usual `NPV` method is also implemented, but for bonds it can be confusing. Like `dirtyPrice`, it returns the cumulative discounted value of the cashflows; but it differs from `dirtyPrice` in a couple of points. First, its result is based on the face value of the bond; and second, it discounts the cash flows to the reference date of the discount curve (usually today) instead of the settlement date of the bond, as in the case of the price.

```
print(bond.NPV())
```

```
9688.079274968233
```

However, there is one case in which the `NPV` method is useful. When it's too close to maturity, the bond can no longer price: for the sample bond we're using, we're still getting results one week before its maturity…

```
ql.Settings.instance().evaluationDate = ql.Date(1, 2, 2028)
```

```
bond.cleanPrice()
```

```
99.9882359483094
```

…but a few days later, it returns a null price, even if the maturity has not been reached:

```
ql.Settings.instance().evaluationDate = ql.Date(4, 2, 2028)
```

```
bond.cleanPrice()
```

```
0.0
```

This is because the settlement date is now at the maturity of the bond, and therefore the latter is no longer tradable.

```
print(bond.maturityDate())
print(bond.settlementDate())
```

```
February 8th, 2028
February 9th, 2028
```

However, it's still possible to calculate the NPV, which gives us the value of the cash flows still to receive.

```
bond.NPV()
```

```
10144.656928164271
```

Finally, and returning to the original evaluation date, it is also possible to obtain a yield from a price:

```
ql.Settings.instance().evaluationDate = today

print(
    bond.bondYield(
        98.08,
        ql.Thirty360(ql.Thirty360.BondBasis),
        ql.SimpleThenCompounded,
        ql.Semiannual,
    )
)
```

```
0.03548806204795839
```

(In C++ the method is called `yield`, but that's a reserved keyword in Python.)

Duration and other figures

Some other calculations are not implemented as bond methods, but as static methods of the `BondFunctions` class; for instance, the various kinds of duration and the convexity.

```
print(
    ql.BondFunctions.duration(
        bond,
        0.035,
        ql.Actual360(),
        ql.Compounded,
        ql.Semiannual,
        ql.Duration.Macaulay,
```

```
    )
)
```

```
3.6046062651997746
```

```
print(
    ql.BondFunctions.duration(
        bond,
        0.035,
        ql.Actual360(),
        ql.Compounded,
        ql.Semiannual,
        ql.Duration.Modified,
    )
)
```

```
3.542610580048918
```

```
print(
    ql.BondFunctions.convexity(
        bond, 0.035, ql.Actual360(), ql.Compounded, ql.Semiannual
    )
)
```

```
14.76085604114641
```

Other kinds of fixed-rate bonds

The same `FixedRateBond` class we used so far can also be instantiated with a sequence of several different coupon rates, making it possible to model step-up bonds:

```
schedule = ql.Schedule(
    ql.Date(8, ql.February, 2023),
    ql.Date(8, ql.February, 2028),
    ql.Period(6, ql.Months),
    ql.TARGET(),
    ql.Following,
    ql.Following,
    ql.DateGeneration.Backward,
    False,
)
settlementDays = 3
faceAmount = 100
```

```
coupons = [0.03, 0.03, 0.04, 0.04, 0.05]
paymentDayCounter = ql.Thirty360(ql.Thirty360.BondBasis)

bond = ql.FixedRateBond(
    settlementDays, faceAmount, schedule, coupons, paymentDayCounter
)
```

Here, as usual, we can see the corresponding cash flows.

```
pd.DataFrame(
    [coupon_info(c) for c in bond.cashflows()],
    columns=("date", "rate", "accrual time", "amount"),
    index=range(1, len(bond.cashflows()) + 1),
).style.format({"amount": "{:.2f}", "rate": "{:.2%}"})
```

	date	rate	accrual time	amount
1	August 8th, 2023	3.00%	0.500000	1.50
2	February 8th, 2024	3.00%	0.500000	1.50
3	August 8th, 2024	4.00%	0.500000	2.00
4	February 10th, 2025	4.00%	0.505556	2.02
5	August 8th, 2025	5.00%	0.494444	2.47
6	February 9th, 2026	5.00%	0.502778	2.51
7	August 10th, 2026	5.00%	0.502778	2.51
8	February 8th, 2027	5.00%	0.494444	2.47
9	August 9th, 2027	5.00%	0.502778	2.51
10	February 8th, 2028	5.00%	0.497222	2.49
11	February 8th, 2028	nan%	nan	100.00

To specify varying notionals, instead, we need to use the `AmortizingFixedRateBond` class:

```
schedule = ql.Schedule(
    ql.Date(8, ql.February, 2018),
    ql.Date(8, ql.February, 2028),
    ql.Period(1, ql.Years),
    ql.TARGET(),
    ql.Following,
    ql.Following,
    ql.DateGeneration.Backward,
    False,
)
settlementDays = 3
```

```
notionals = [10000, 10000, 8000, 8000, 6000, 6000, 4000, 4000, 2000, 2000]
coupons = [0.03]
paymentDayCounter = ql.Thirty360(ql.Thirty360.BondBasis)

bond = ql.AmortizingFixedRateBond(
    settlementDays, notionals, schedule, coupons, paymentDayCounter
)
bond.setPricingEngine(engine)
```

In this case, the cashflows include amortization payments:

```
pd.DataFrame(
    [coupon_info(c) for c in bond.cashflows()],
    columns=("date", "rate", "accrual time", "amount"),
    index=range(1, len(bond.cashflows()) + 1),
).style.format({"amount": "{:.2f}", "rate": "{:.2%}"})
```

	date	rate	accrual time	amount
1	February 8th, 2019	3.00%	1.000000	300.00
2	February 10th, 2020	3.00%	1.005556	301.67
3	February 10th, 2020	nan%	nan	2000.00
4	February 8th, 2021	3.00%	0.994444	238.67
5	February 8th, 2022	3.00%	1.000000	240.00
6	February 8th, 2022	nan%	nan	2000.00
7	February 8th, 2023	3.00%	1.000000	180.00
8	February 8th, 2024	3.00%	1.000000	180.00
9	February 8th, 2024	nan%	nan	2000.00
10	February 10th, 2025	3.00%	1.005556	120.67
11	February 9th, 2026	3.00%	0.997222	119.67
12	February 9th, 2026	nan%	nan	2000.00
13	February 8th, 2027	3.00%	0.997222	59.83
14	February 8th, 2028	3.00%	1.000000	60.00
15	February 8th, 2028	nan%	nan	2000.00

It's also possible to ask for the current notional at a given date…

```
bond.notional(bond.settlementDate(ql.Date(25, 10, 2027)))
```

```
2000.0
```

…defaulting, as usual, to the current settlement date:

```
bond.notional()
```

```
4000.0
```

Following common market practice, the price returned by the class is scaled so that it's in base 100 relative to the current notional; the NPV, as usual, returns the cash value. For instance:

```
bond.NPV()
```

```
3910.55084058516
```

```
bond.cleanPrice()
```

```
97.07643264387757
```

Zero-coupon bonds

There's not a lot to say about these bonds; they can be instantiated by means of a dedicated class…

```
bond = ql.ZeroCouponBond(
    settlementDays=3,
    calendar=ql.TARGET(),
    faceAmount=10_000,
    maturityDate=ql.Date(8, ql.February, 2030),
)
bond.setPricingEngine(engine)
```

…and offer the same features of fixed-rate bonds.

```
print(bond.cleanPrice())
```

```
79.162564713698
```

```
print(
    bond.bondYield(
        79.50,
        ql.Thirty360(ql.Thirty360.BondBasis),
        ql.Compounded,
        ql.Annual,
    )
)
```

```
0.040684510707855226
```

```
print(
    ql.BondFunctions.duration(
        bond,
        0.035,
        ql.Actual360(),
        ql.Compounded,
        ql.Semiannual,
        ql.Duration.Macaulay,
    )
)
```

```
5.841666666666667
```

Floating-rate bonds

Not surprisingly, floating-rate bonds require an interest-rate index to determine the coupon rates; in turn, the index requires a forecasting curve. I'm using another flat curve for brevity, but in a real-world case it would probably be a curve bootstrapped from market data.

```
forecast_curve = ql.FlatForward(0, ql.TARGET(), 0.025, ql.Actual360())
euribor = ql.Euribor6M(ql.YieldTermStructureHandle(forecast_curve))
```

```
schedule = ql.Schedule(
    ql.Date(8, ql.February, 2023),
    ql.Date(8, ql.February, 2028),
    ql.Period(6, ql.Months),
    ql.TARGET(),
    ql.Following,
    ql.Following,
    ql.DateGeneration.Backward,
    False,
)

bond = ql.FloatingRateBond(
    settlementDays=3,
    faceAmount=10_000,
    schedule=schedule,
    index=euribor,
    spreads=[0.001],
    paymentDayCounter=ql.Actual360(),
    paymentConvention=ql.Following,
)
```

```
bond.setPricingEngine(engine)
```

Once instantiated, they provide the same facilities as fixed-rate bonds; however, they might also require a past index fixing to calculate the amount of the current coupon.

```
try:
    print(bond.cleanPrice())
except Exception as e:
    print(f"Error: {e}")
```

```
Error: Missing Euribor6M Actual/360 fixing for February 6th, 2024
```

The missing fixings can be added through the index:

```
euribor.addFixing(ql.Date(6, ql.February, 2023), 0.028)
euribor.addFixing(ql.Date(4, ql.August, 2023), 0.0283)
euribor.addFixing(ql.Date(6, ql.February, 2024), 0.0279)

print(bond.cleanPrice())
```

```
95.08004252605194
```

Coupon and accrual information work the same (with future coupon rates and accruals being forecast via the curve, rather than determined). Once downcast, the coupons can provide a bit more information about the index fixings.

```
def coupon_info(cf):
    c = ql.as_floating_rate_coupon(cf)
    if not c:
        return (cf.date(), None, None, None, None, cf.amount())
    else:
        return (
            c.date(),
            c.fixingDate(),
            c.indexFixing(),
            c.rate(),
            c.accrualPeriod(),
            c.amount(),
        )


pd.DataFrame(
    [coupon_info(c) for c in bond.cashflows()],
    columns=(
```

```
        "payment date",
        "fixing date",
        "index fixing",
        "rate",
        "accrual time",
        "amount",
    ),
    index=range(1, len(bond.cashflows()) + 1),
).style.format(
    {"amount": "{:.2f}", "rate": "{:.2%}", "index fixing": "{:.2%}"}
)
```

	payment date	fixing date	index fixing	rate	accrual time	amount
1	August 8th, 2023	February 6th, 2023	2.80%	2.90%	0.502778	145.81
2	February 8th, 2024	August 4th, 2023	2.83%	2.93%	0.511111	149.76
3	August 8th, 2024	February 6th, 2024	2.79%	2.89%	0.505556	146.11
4	February 10th, 2025	August 6th, 2024	2.52%	2.62%	0.516667	135.17
5	August 8th, 2025	February 6th, 2025	2.52%	2.62%	0.497222	130.05
6	February 9th, 2026	August 6th, 2025	2.52%	2.62%	0.513889	134.44
7	August 10th, 2026	February 5th, 2026	2.52%	2.62%	0.505556	132.25
8	February 8th, 2027	August 6th, 2026	2.52%	2.62%	0.505556	132.25
9	August 9th, 2027	February 4th, 2027	2.52%	2.62%	0.505556	132.25
10	February 8th, 2028	August 5th, 2027	2.52%	2.62%	0.508333	132.98
11	February 8th, 2028	None	nan%	nan%	nan	10000.00

Also, floaters have a number of optional features; for instance, caps and floors. However, they require additional market information:

```
bond = ql.FloatingRateBond(
    settlementDays=3,
    faceAmount=10_000,
    schedule=schedule,
    index=euribor,
    paymentDayCounter=ql.Actual360(),
    paymentConvention=ql.Following,
    floors=[0.0],
    caps=[0.05],
)

bond.setPricingEngine(
```

```
    ql.DiscountingBondEngine(ql.YieldTermStructureHandle(discount_curve))
)
```

```
try:
    print(bond.cleanPrice())
except Exception as e:
    print(f"Error: {e}")
```

```
Error: pricer not set
```

Why is the error worded in this way? If you read the chapter on cash flows of *Implementing QuantLib*, you'll see that floating-rate coupons (such as those created by the `FloatingRateBond` constructor) can be set a pricer, more or less like instruments can be set a pricing engine. We didn't see it so far because, if caps and floors are not enabled, coupons based on an IBOR index are set a default pricer than calculates their fixing from the forecast curve they hold.

Caps and floors, however, introduce optionality and thus require volatility data. That information is missing from the default pricer, so the coupons are not given one during construction; we have to create the pricer ourselves and pass it to the coupons.

```
volatility = ql.ConstantOptionletVolatility(
    0, ql.TARGET(), ql.Following, 0.25, ql.Actual360()
)
pricer = ql.BlackIborCouponPricer(
    ql.OptionletVolatilityStructureHandle(volatility)
)
```

```
ql.setCouponPricer(bond.cashflows(), pricer)
print(bond.cleanPrice())
```

```
94.68340203601055
```

Looking at the cash flow information shows, for dates far enough in the future, that the effective rate paid by the coupon no longer equals the index fixing. This is due to the change in value coming from the caps and floors.

```
pd.DataFrame(
    [coupon_info(c) for c in bond.cashflows()],
    columns=(
        "payment date",
        "fixing date",
        "index fixing",
        "rate",
        "accrual time",
        "amount",
```

```
    ),
    index=range(1, len(bond.cashflows()) + 1),
).style.format(
    {"amount": "{:.2f}", "rate": "{:.4%}", "index fixing": "{:.4%}"}
)
```

	payment date	fixing date	index fixing	rate	accrual time	amount
1	August 8th, 2023	February 6th, 2023	2.8000%	2.8000%	0.502778	140.78
2	February 8th, 2024	August 4th, 2023	2.8300%	2.8300%	0.511111	144.64
3	August 8th, 2024	February 6th, 2024	2.7900%	2.7900%	0.505556	141.05
4	February 10th, 2025	August 6th, 2024	2.5162%	2.5162%	0.516667	130.00
5	August 8th, 2025	February 6th, 2025	2.5159%	2.5154%	0.497222	125.07
6	February 9th, 2026	August 6th, 2025	2.5161%	2.5136%	0.513889	129.17
7	August 10th, 2026	February 5th, 2026	2.5159%	2.5073%	0.505556	126.76
8	February 8th, 2027	August 6th, 2026	2.5160%	2.4976%	0.505556	126.27
9	August 9th, 2027	February 4th, 2027	2.5159%	2.4851%	0.505556	125.64
10	February 8th, 2028	August 5th, 2027	2.5160%	2.4707%	0.508333	125.60
11	February 8th, 2028	None	nan%	nan%	nan	10000.00

Floating-rate bonds on risk-free rates

There is no specific class yet for floaters that pay compounded overnight rates such as SOFR, ESTR or SONIA. However, it's possible to create the cashflows and use them to build an instance of the base `Bond` class. The bond will figure out its redemption on its own.

```
forecast_curve = ql.FlatForward(0, ql.TARGET(), 0.02, ql.Actual360())
estr = ql.Estr(ql.YieldTermStructureHandle(forecast_curve))
```

As usual, the index will need past fixings. In the real world, you'd read them from a database or an API. Here, I'll feed it constant ones for brevity:

```
d = ql.Date(8, ql.February, 2024)
while d < today:
    if estr.isValidFixingDate(d):
        estr.addFixing(d, 0.02)
    d += 1
```

Onwards with the bond construction:

```
schedule = ql.Schedule(
    ql.Date(8, ql.February, 2024),
    ql.Date(8, ql.February, 2029),
```

```
    ql.Period(1, ql.Years),
    ql.TARGET(),
    ql.Following,
    ql.Following,
    ql.DateGeneration.Backward,
    False,
)

coupons = ql.OvernightLeg(
    schedule=schedule,
    index=estr,
    nominals=[10_000],
    spreads=[0.0023],
    paymentLag=2,
)
```

In C++, the call to `OvernightLeg` above would be written as

```
Leg coupons =
    OvernightLeg(schedule, estr)
    .withNotionals(10000)
    .withSpreads(0.0023)
    .withPaymentLeg(2);
```

and, as you might guess, several other parameters are available. Finally, we can create the bond as

```
bond = ql.Bond(
    settlementDays,
    ql.TARGET(),
    schedule[0],
    coupons,
)
bond.setPricingEngine(engine)
```

```
print(bond.cleanPrice())
```

```
92.07577763235304
```

This kind of bonds can also be used as a proxy for old-style floaters that need fall-back calculation after the cessation of LIBOR fixings.

A warning: duration of floating-rate bonds

If we try to pass a floating-rate bond we instantiated to the function provided for calculating bond durations, we run into a problem. For, say, a 1.5% semiannual yield, the result we get is:

```
forecast_curve = ql.FlatForward(0, ql.TARGET(), 0.015, ql.Actual360())
forecast_handle = ql.RelinkableYieldTermStructureHandle(forecast_curve)
euribor = ql.Euribor6M(forecast_handle)

schedule = ql.Schedule(
    ql.Date(8, ql.February, 2024),
    ql.Date(8, ql.February, 2028),
    ql.Period(6, ql.Months),
    ql.TARGET(),
    ql.Following,
    ql.Following,
    ql.DateGeneration.Backward,
    False,
)

bond = ql.FloatingRateBond(
    settlementDays=3,
    faceAmount=100,
    schedule=schedule,
    index=euribor,
    spreads=[0.001],
    paymentDayCounter=ql.Actual360(),
    paymentConvention=ql.Following,
)
```

```
y0 = 0.015
y = ql.InterestRate(y0, ql.Actual360(), ql.Compounded, ql.Semiannual)
print(ql.BondFunctions.duration(bond, y, ql.Duration.Modified))
```

```
3.6491647768932602
```

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

The problem is that 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:

```
y = ql.SimpleQuote(y0)
yield_curve = ql.FlatForward(
    bond.settlementDate(),
    ql.QuoteHandle(y),
    ql.Actual360(),
    ql.Compounded,
    ql.Semiannual,
)

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

print(P)
```

```
101.43285359570949
```

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:

```
dy = 1e-5

y.setValue(y0 + 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(y0 - 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(y0)
```

```
101.42915222219966
101.43655512613346
```

```
print(-(1 / P) * (P_p - P_m) / (2 * dy))
```

```
3.649164778159966
```

which is, within accuracy, the same figure returned by `BondFunctions.duration`.

The reason 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 if it were 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:

```
pd.DataFrame(
    list(zip(dates, cfs_p, discounts_p, cfs_m, discounts_m)),
    columns=(
        "date",
        "amount (+)",
        "discounts (+)",
        "amount (-)",
        "discounts (-)",
    ),
    index=range(1, len(dates) + 1),
)
```

	date	amount (+)	discounts (+)	amount (-)	discounts (-)
1	August 8th, 2024	1.461056	0.996144	1.461056	0.996149
2	February 10th, 2025	0.829678	0.988478	0.829678	0.988493
3	August 8th, 2025	0.798344	0.981155	0.798344	0.981180
4	February 9th, 2026	0.825201	0.973644	0.825201	0.973679
5	August 10th, 2026	0.811772	0.966311	0.811772	0.966355
6	February 8th, 2027	0.811772	0.959033	0.811772	0.959087
7	August 9th, 2027	0.811772	0.951810	0.811772	0.951873
8	February 8th, 2028	0.816248	0.944602	0.816248	0.944674
9	February 8th, 2028	100.000000	0.944602	100.000000	0.944674

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

Unfortunately, there's no easy way to modify the `BondFunctions.duration` method so that it does what we expect. 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; this might imply setting up a z-spread between forecast and discount curve, but I'll gloss over this now. I'll also set a pricing engine to the bond so it does the dirty price calculation for us.

```
forecast_handle.linkTo(yield_curve)
bond.setPricingEngine(
    ql.DiscountingBondEngine(ql.YieldTermStructureHandle(yield_curve))
)
```

```
y.setValue(y0 + 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(y0 - 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(y0)
```

```
101.41322141911154
101.41375522190961
```

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

```
pd.DataFrame(
    list(zip(dates, cfs_p, discounts_p, cfs_m, discounts_m)),
    columns=(
        "date",
        "amount (+)",
        "discounts (+)",
        "amount (-)",
        "discounts (-)",
    ),
    index=range(1, len(dates) + 1),
)
```

	date	amount (+)	discounts (+)	amount (-)	discounts (-)
1	August 8th, 2024	1.461056	0.996144	1.461056	0.996149
2	February 10th, 2025	0.827280	0.988478	0.826247	0.988493
3	August 8th, 2025	0.796037	0.981155	0.795043	0.981180
4	February 9th, 2026	0.822816	0.973644	0.821788	0.973679

	date	amount (+)	discounts (+)	amount (-)	discounts (-)
5	August 10th, 2026	0.809426	0.966311	0.808415	0.966355
6	February 8th, 2027	0.809426	0.959033	0.808415	0.959087
7	August 9th, 2027	0.809426	0.951810	0.808415	0.951873
8	February 8th, 2028	0.813889	0.944602	0.812872	0.944674
9	February 8th, 2028	100.000000	0.944602	100.000000	0.944674

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

```
print(-(1 / P) * (P_p - P_m) / (2 * dy))
```

```
0.2631311153887885
```

Chapter 16

Different kinds of swaps

```
import QuantLib as ql
import pandas as pd

today = ql.Date(21, ql.September, 2023)
ql.Settings.instance().evaluationDate = today

bps = 1e-4
```

Overnight-indexed swaps

Overnight-indexed swaps (OIS) need a single interest-rate curve that will be used both for forecasting the values of the underlying index and for discounting the value of the resulting cashflows. The bootstrapping process used to create the curve is the subject of another notebook; for brevity, here I'll use a mock curve with zero rates increasing linearly over time.

```
sofr_curve = ql.ZeroCurve(
    [today, today + ql.Period(50, ql.Years)],
    [0.02, 0.04],
    ql.Actual365Fixed(),
)
```

Given the curve, we can instantiate index objects able to forecast their future fixings.

```
sofr_handle = ql.YieldTermStructureHandle(sofr_curve)
```

```
sofr = ql.Sofr(sofr_handle)
```

```
sofr.fixing(ql.Date(7, ql.February, 2025))
```

```
0.020821983903180907
```

In turn, we can use the index to build an OIS:

```
start_date = today
end_date = start_date + ql.Period(10, ql.Years)
coupon_tenor = ql.Period(1, ql.Years)
calendar = ql.UnitedStates(ql.UnitedStates.GovernmentBond)
convention = ql.Following
rule = ql.DateGeneration.Forward
end_of_month = False

fixed_rate = 40 * bps
fixed_day_counter = sofr.dayCounter()

schedule = ql.Schedule(
    start_date,
    end_date,
    coupon_tenor,
    calendar,
    convention,
    convention,
    rule,
    end_of_month,
)
swap = ql.OvernightIndexedSwap(
    ql.Swap.Payer, 1_000_000, schedule, fixed_rate, fixed_day_counter, sofr
)
```

We'll also use the SOFR curve to discount the cashflows…

```
swap.setPricingEngine(ql.DiscountingSwapEngine(sofr_handle))
```

…and thus get the value of the swap.

```
swap.NPV()
```

```
177641.18261457735
```

For more details, it's possible to extract the cashflows from the swap and call their methods.

```
data = []
for cf in swap.fixedLeg():
    coupon = ql.as_fixed_rate_coupon(cf)
    data.append(
        (
            coupon.date(),
            coupon.rate(),
            coupon.accrualPeriod(),
            coupon.amount(),
        )
    )
```

```
pd.DataFrame(
    data, columns=["date", "rate", "tenor", "amount"]
).style.format({"amount": "{:.2f}", "rate": "{:.2%}"})
```

	date	rate	tenor	amount
0	September 23rd, 2024	0.40%	1.022222	4088.89
1	September 22nd, 2025	0.40%	1.011111	4044.44
2	September 21st, 2026	0.40%	1.011111	4044.44
3	September 21st, 2027	0.40%	1.013889	4055.56
4	September 21st, 2028	0.40%	1.016667	4066.67
5	September 21st, 2029	0.40%	1.013889	4055.56
6	September 23rd, 2030	0.40%	1.019444	4077.78
7	September 22nd, 2031	0.40%	1.011111	4044.44
8	September 21st, 2032	0.40%	1.013889	4055.56
9	September 21st, 2033	0.40%	1.013889	4055.56

```
data = []
for cf in swap.overnightLeg():
    coupon = ql.as_floating_rate_coupon(cf)
    data.append(
        (
            coupon.date(),
            coupon.rate(),
            coupon.accrualPeriod(),
            coupon.amount(),
        )
    )
```

```
pd.DataFrame(
    data, columns=["date", "rate", "tenor", "amount"]
).style.format({"amount": "{:.2f}", "rate": "{:.2%}"})
```

	date	rate	tenor	amount
0	September 23rd, 2024	2.03%	1.022222	20783.73
1	September 22nd, 2025	2.11%	1.011111	21371.70
2	September 21st, 2026	2.19%	1.011111	22184.07
3	September 21st, 2027	2.27%	1.013889	23062.11
4	September 21st, 2028	2.36%	1.016667	23947.63
5	September 21st, 2029	2.44%	1.013889	24701.40
6	September 23rd, 2030	2.52%	1.019444	25664.78
7	September 22nd, 2031	2.60%	1.011111	26271.32
8	September 21st, 2032	2.68%	1.013889	27164.13
9	September 21st, 2033	2.76%	1.013889	27985.60

Other results

Besides its present value, the OIS can return other figures, such as the fixed rate that would make the swap fair:

```
swap.fairRate()
```

```
0.02379864737943738
```

We can test it by building a second swap, identical to the first but paying the fair rate:

```
test_swap = ql.OvernightIndexedSwap(
    ql.Swap.Payer,
    1_000_000,
    schedule,
    swap.fairRate(),
    fixed_day_counter,
    sofr,
)
test_swap.setPricingEngine(ql.DiscountingSwapEngine(sofr_handle))
test_swap.NPV()
```

```
-2.9103830456733704e-11
```

As expected, the NPV of the fair swap is zero within numerical accuracy.

Other results include the NPV of each leg and their BPS, that is, the change in their value if their rate increases by 1 bp:

```
swap.fixedLegNPV()
```

```
-35889.55936435795
```

```
swap.overnightLegNPV()
```

```
213530.7419789353
```

```
swap.fixedLegBPS()
```

```
-897.2389841089508
```

Again, we can test it by comparing the expected value of a swap whose fixed leg pays 1 bps more…

```
swap.fixedLegNPV() + swap.fixedLegBPS()
```

```
-36786.7983484669
```

…with the value of an actual swap paying that modified rate:

```
test_swap = ql.OvernightIndexedSwap(
    ql.Swap.Payer,
    1_000_000,
    schedule,
    fixed_rate + 1 * bps,
    fixed_day_counter,
    sofr,
)
test_swap.setPricingEngine(ql.DiscountingSwapEngine(sofr_handle))
test_swap.fixedLegNPV()
```

```
-36786.79834846717
```

Known fixings

An added twist: if the swap already started, it needs fixings in the past that the curve can't forecast.

```
start_date = today - ql.Period(3, ql.Months)
end_date = start_date + ql.Period(10, ql.Years)

schedule = ql.Schedule(
    start_date,
    end_date,
    coupon_tenor,
```

```
    calendar,
    convention,
    convention,
    rule,
    end_of_month,
)
swap = ql.OvernightIndexedSwap(
    ql.Swap.Payer, 1_000_000, schedule, fixed_rate, fixed_day_counter, sofr
)
swap.setPricingEngine(ql.DiscountingSwapEngine(sofr_handle))
```

```
try:
    swap.NPV()
except Exception as e:
    print(f"{type(e).__name__}: {e}")
```

```
RuntimeError: 2nd leg: Missing SOFRON Actual/360 fixing for June 21st, 2023
```

The information can be stored through the index (and will be shared by all instances of that index). If it's already available and stored, today's fixing will be used, too; if not, it will be forecast from the curve.

```
d = ql.Date(21, ql.June, 2023)
while d <= today:
    if sofr.isValidFixingDate(d):
        sofr.addFixing(d, 0.02)
    d += 1
```

```
swap.NPV()
```

```
176996.59653466725
```

More features

Overnight-indexed swaps make it possible to specify other aspects of the contract; for instance, the notionals of the coupons can vary, as in the following swap:

```
notionals = [
    1_000_000,
    900_000,
    800_000,
    700_000,
    600_000,
    500_000,
```

```
    400_000,
    300_000,
    200_000,
    100_000,
]

swap = ql.OvernightIndexedSwap(
    ql.Swap.Payer, notionals, schedule, fixed_rate, fixed_day_counter, sofr
)
swap.setPricingEngine(ql.DiscountingSwapEngine(sofr_handle))
```

```
swap.NPV()
```

```
94959.4310675247
```

Here are the corresponding floating-rate coupons:

```
data = []
for cf in swap.overnightLeg():
    coupon = ql.as_floating_rate_coupon(cf)
    data.append(
        (
            coupon.date(),
            coupon.nominal(),
            coupon.rate(),
            coupon.accrualPeriod(),
            coupon.amount(),
        )
    )
```

```
pd.DataFrame(
    data, columns=["date", "nominal", "rate", "tenor", "amount"]
).style.format({"amount": "{:.2f}", "rate": "{:.2%}", "nominal": "{:.0f}"})
```

	date	nominal	rate	tenor	amount
0	June 21st, 2024	1000000	2.02%	1.016667	20559.03
1	June 23rd, 2025	900000	2.09%	1.019444	19207.48
2	June 22nd, 2026	800000	2.17%	1.011111	17584.73
3	June 21st, 2027	700000	2.25%	1.011111	15955.64
4	June 21st, 2028	600000	2.34%	1.016667	14244.46
5	June 21st, 2029	500000	2.42%	1.013889	12247.47
6	June 21st, 2030	400000	2.50%	1.013889	10125.71
7	June 23rd, 2031	300000	2.58%	1.019444	7884.48

	date	nominal	rate	tenor	amount
8	June 21st, 2032	200000	2.66%	1.011111	5376.69
9	June 21st, 2033	100000	2.74%	1.013889	2777.85

Other features make it possible for the floating-rate leg to pay an added spread on top of the SOFR fixings, or for the coupons to have a payment lag of a few days:

```
spread = 10 * bps
payment_lag = 2

swap = ql.OvernightIndexedSwap(
    ql.Swap.Payer,
    1_000_000,
    schedule,
    fixed_rate,
    fixed_day_counter,
    sofr,
    spread,
    payment_lag,
)
swap.setPricingEngine(ql.DiscountingSwapEngine(sofr_handle))
```

```
swap.NPV()
```

```
185991.83097733647
```

Again, we can check the coupons and compare the payment dates and the rates with the previous cases:

```
data = []
for cf in swap.overnightLeg():
    coupon = ql.as_floating_rate_coupon(cf)
    data.append(
        (
            coupon.date(),
            coupon.rate(),
            coupon.accrualPeriod(),
            coupon.amount(),
        )
    )
```

```
pd.DataFrame(
    data, columns=["date", "rate", "tenor", "amount"]
```

```
).style.format({"amount": "{:.2f}", "rate": "{:.2%}"})
```

	date	rate	tenor	amount
0	June 25th, 2024	2.12%	1.016667	21575.69
1	June 25th, 2025	2.19%	1.019444	22361.09
2	June 24th, 2026	2.27%	1.011111	22992.03
3	June 23rd, 2027	2.35%	1.011111	23804.88
4	June 23rd, 2028	2.44%	1.016667	24757.43
5	June 25th, 2029	2.52%	1.013889	25508.83
6	June 25th, 2030	2.60%	1.013889	26328.17
7	June 25th, 2031	2.68%	1.019444	27301.05
8	June 23rd, 2032	2.76%	1.011111	27894.57
9	June 23rd, 2033	2.84%	1.013889	28792.37

It's also possible for the swap to calculate the spread that would make it fair:

```
swap.fairSpread()
```

```
-0.01960712998027658
```

And again, we can check this by creating a swap with the fair spread:

```
test_swap = ql.OvernightIndexedSwap(
    ql.Swap.Payer,
    1_000_000,
    schedule,
    fixed_rate,
    fixed_day_counter,
    sofr,
    swap.fairSpread(),
    payment_lag,
)
test_swap.setPricingEngine(ql.DiscountingSwapEngine(sofr_handle))
test_swap.NPV()
```

```
3.637978807091713e-11
```

As before, the NPV is null within numerical accuracy.

At the time of this writing, features like lookback and lockout days are being worked on; they should be available starting from release 1.35, scheduled for July 2024.

Fixed-vs-floater swaps

With some differences, the same ideas apply to vanilla fixed-vs-floater swaps in the markets where they're still relevant. For instance, a swap paying a fixed rate vs 6-months Euribor will need a discount curve (probably calculated from ESTR rates) and a forecast curve for Euribor. As before, I'll use mocks.

```
estr_curve = ql.ZeroCurve(
    [today, today + ql.Period(50, ql.Years)],
    [0.02, 0.04],
    ql.Actual365Fixed(),
)

euribor6m_curve = ql.ZeroCurve(
    [today, today + ql.Period(50, ql.Years)],
    [0.03, 0.05],
    ql.Actual365Fixed(),
)
```

```
estr_handle = ql.YieldTermStructureHandle(estr_curve)
```

```
euribor6m_handle = ql.YieldTermStructureHandle(euribor6m_curve)
```

```
euribor6m = ql.Euribor(ql.Period(6, ql.Months), euribor6m_handle)
```

```
euribor6m.fixing(ql.Date(8, ql.February, 2024))
```

```
0.03032682683069796
```

Vanilla swaps are built using a different class, but work in the same way.

```
start_date = today + 2
end_date = start_date + ql.Period(10, ql.Years)
calendar = ql.TARGET()
rule = ql.DateGeneration.Forward
fixed_frequency = ql.Annual
fixed_convention = ql.Unadjusted
fixed_day_count = ql.Thirty360(ql.Thirty360.BondBasis)
float_convention = euribor6m.businessDayConvention()
end_of_month = False
fixed_rate = 50 * bps

fixed_schedule = ql.Schedule(
    start_date,
    end_date,
```

```
    ql.Period(fixed_frequency),
    calendar,
    fixed_convention,
    fixed_convention,
    rule,
    end_of_month,
)
float_schedule = ql.Schedule(
    start_date,
    end_date,
    euribor6m.tenor(),
    calendar,
    float_convention,
    float_convention,
    rule,
    end_of_month,
)
swap = ql.VanillaSwap(
    ql.Swap.Payer,
    1_000_000,
    fixed_schedule,
    fixed_rate,
    fixed_day_count,
    float_schedule,
    euribor6m,
    0.0,
    euribor6m.dayCounter(),
)
```

This time, though, we'll take care to use the correct discount curve:

```
swap.setPricingEngine(ql.DiscountingSwapEngine(estr_handle))
```

Once the swap is set up, it can return a number of results:

```
swap.NPV()
```

```
259485.7403164844
```

```
swap.fairRate()
```

```
0.0343510181748013
```

```
swap.fairSpread()
```

```
-0.02876783996070111
```

Again, seasoned swaps need past-fixing information.

```
start_date = today - ql.Period(3, ql.Months)
end_date = start_date + ql.Period(10, ql.Years)

fixed_schedule = ql.Schedule(
    start_date,
    end_date,
    ql.Period(fixed_frequency),
    calendar,
    fixed_convention,
    fixed_convention,
    rule,
    end_of_month,
)
float_schedule = ql.Schedule(
    start_date,
    end_date,
    euribor6m.tenor(),
    calendar,
    float_convention,
    float_convention,
    rule,
    end_of_month,
)
swap = ql.VanillaSwap(
    ql.Swap.Payer,
    1_000_000,
    fixed_schedule,
    fixed_rate,
    fixed_day_count,
    float_schedule,
    euribor6m,
    0.0,
    euribor6m.dayCounter(),
)
swap.setPricingEngine(ql.DiscountingSwapEngine(estr_handle))
```

```
try:
    swap.NPV()
except Exception as e:
```

```
    print(f"{type(e).__name__}: {e}")
```

```
RuntimeError: 2nd leg: Missing Euribor6M Actual/360 fixing for June 19th, 2023
```

```
euribor6m.addFixing(ql.Date(19, 6, 2023), 0.03)
```

```
swap.NPV()
```

```
259487.36632473825
```

And again, we can dive into the cashflows:

```
data = []
for cf in swap.fixedLeg():
    coupon = ql.as_fixed_rate_coupon(cf)
    data.append(
        (
            coupon.date(),
            coupon.rate(),
            coupon.accrualPeriod(),
            coupon.amount(),
        )
    )
```

```
pd.DataFrame(
    data, columns=["date", "rate", "tenor", "amount"]
).style.format({"amount": "{:.2f}", "rate": "{:.2%}"})
```

	date	rate	tenor	amount
0	June 21st, 2024	0.50%	1.000000	5000.00
1	June 23rd, 2025	0.50%	1.000000	5000.00
2	June 22nd, 2026	0.50%	1.000000	5000.00
3	June 21st, 2027	0.50%	1.000000	5000.00
4	June 21st, 2028	0.50%	1.000000	5000.00
5	June 21st, 2029	0.50%	1.000000	5000.00
6	June 21st, 2030	0.50%	1.000000	5000.00
7	June 23rd, 2031	0.50%	1.000000	5000.00
8	June 21st, 2032	0.50%	1.000000	5000.00
9	June 21st, 2033	0.50%	1.000000	5000.00

```
data = []
for cf in swap.floatingLeg():
    coupon = ql.as_floating_rate_coupon(cf)
```

```
    data.append(
        (
            coupon.date(),
            coupon.rate(),
            coupon.accrualPeriod(),
            coupon.amount(),
        )
    )
```

```
pd.DataFrame(
    data, columns=["date", "rate", "tenor", "amount"]
).style.format({"amount": "{:.2f}", "rate": "{:.2%}"})
```

	date	rate	tenor	amount
0	December 21st, 2023	3.00%	0.508333	15250.00
1	June 21st, 2024	3.02%	0.508333	15358.25
2	December 23rd, 2024	3.06%	0.513889	15734.84
3	June 23rd, 2025	3.10%	0.505556	15681.24
4	December 22nd, 2025	3.14%	0.505556	15883.15
5	June 22nd, 2026	3.18%	0.505556	16085.09
6	December 21st, 2026	3.22%	0.505556	16287.07
7	June 21st, 2027	3.26%	0.505556	16489.09
8	December 21st, 2027	3.30%	0.508333	16784.18
9	June 21st, 2028	3.34%	0.508333	16988.53
10	December 21st, 2028	3.38%	0.508333	17192.92
11	June 21st, 2029	3.42%	0.505556	17300.91
12	December 21st, 2029	3.46%	0.508333	17600.70
13	June 21st, 2030	3.50%	0.505556	17706.51
14	December 23rd, 2030	3.54%	0.513889	18208.38
15	June 23rd, 2031	3.58%	0.505556	18114.49
16	December 22nd, 2031	3.62%	0.505556	18316.87
17	June 21st, 2032	3.66%	0.505556	18519.30
18	December 21st, 2032	3.70%	0.508333	18826.15
19	June 21st, 2033	3.74%	0.505556	18925.38

More generic swaps

To build fixed-vs-floating swaps with less common features (such as decreasing notionals or floating-rate gearings) we can build the two legs separately and put them together in an instance of the `Swap` class.

```
fixed_leg = ql.FixedRateLeg(
    schedule=fixed_schedule,
    dayCount=ql.Thirty360(ql.Thirty360.BondBasis),
    nominals=[10000, 8000, 6000, 4000, 2000],
    couponRates=[0.01],
)
floating_leg = ql.IborLeg(
    schedule=float_schedule,
    index=euribor6m,
    nominals=[
        10000,
        10000,
        8000,
        8000,
        6000,
        6000,
        4000,
        4000,
        2000,
        2000,
    ],
    gearings=[0.8],
)
swap = ql.Swap(fixed_leg, floating_leg)
swap.setPricingEngine(ql.DiscountingSwapEngine(estr_handle))
swap.NPV()
```

```
601.4025020590461
```

Some results are still available, and can be addressed by the index of the leg.

```
print(swap.legNPV(0))
print(swap.legNPV(1))
```

```
-370.756802307719
972.1593043667651
```

Some others are currently available through other classes.

```
print(ql.CashFlows.bps(swap.leg(0), estr_curve, False))
print(ql.CashFlows.bps(swap.leg(1), estr_curve, False))
```

```
3.707568023077187
3.7857461356076967
```

Other calculations require a bit more logic.

Basis swaps

We don't necessarily need one fixed-rate leg and one floating-rate leg. By combining two floating-rate legs with different indexes, we can build a basis swap.

```
estr = ql.Estr(estr_handle)

d = ql.Date(21, ql.June, 2023)
while d < today:
    if estr.isValidFixingDate(d):
        estr.addFixing(d, -0.0035)
    d += 1
```

```
euribor_leg = ql.IborLeg([10000], float_schedule, euribor6m)
estr_leg = ql.OvernightLeg([10000], float_schedule, estr)

swap = ql.Swap(estr_leg, euribor_leg)
swap.setPricingEngine(ql.DiscountingSwapEngine(estr_handle))
swap.NPV()
```

```
968.7991051046724
```

```
print(swap.legNPV(0))
print(swap.legNPV(1))
```

```
-2070.8646132177128
3039.663718322385
```

Cross-currency swaps

Finally, like for basis swaps, there is no specific class modeling cross-currency swaps; and it's not always possible to use the `Swap` class, either. We can price them by creating the two legs explicitly (including the final notional exchange) and using library functions to get their NPV. This is the subject of another notebook.

Chapter 17

Asset swaps

```
import QuantLib as ql
import pandas as pd
```

```
today = ql.Date(20, ql.May, 2021)
ql.Settings.instance().evaluationDate = today
```

The `AssetSwap` class builds a swap that exchanges the coupons from a given bond for floating-rate coupons.

Let's take a fixed-rate bond as an example:

```
schedule = ql.Schedule(
    ql.Date(8, ql.February, 2020),
    ql.Date(8, ql.February, 2025),
    ql.Period(6, ql.Months),
    ql.TARGET(),
    ql.Following,
    ql.Following,
    ql.DateGeneration.Backward,
    False,
)
settlementDays = 3
faceAmount = 100
coupons = [0.03]
paymentDayCounter = ql.Thirty360(ql.Thirty360.BondBasis)

bond = ql.FixedRateBond(
```

```
    settlementDays, faceAmount, schedule, coupons, paymentDayCounter
)
```

Besides the bond, the `AssetSwap` constructor takes the floating-rate index used to fix the exchanged coupons…

```
forecast_curve = ql.RelinkableYieldTermStructureHandle(
    ql.FlatForward(today, 0.01, ql.Actual360())
)
index = ql.Euribor6M(forecast_curve)
```

…and other parameters: a spread over the floating rate, the bond price, the schedule for the floating-rate coupons (which is optional: if we pass an empty one, the swap will use the same as the bond), the day-count convention for the floating-rate coupons, and a couple of flags specifying the kind of swap we're creating.

```
spread = 0.0050
bond_price = 103.0

pay_fixed = True
par_asset_swap = False

swap = ql.AssetSwap(
    pay_fixed,
    bond,
    bond_price,
    index,
    spread,
    ql.Schedule(),
    index.dayCounter(),
    par_asset_swap,
)
```

When `par_asset_swap = False`, the swap creates floating-rate coupons paid on a notional equal to the bond price. As for bonds, it's possible to extract the coupons and retrieve information on each one:

```
def print_coupon_info(cashflows):
    data = []
    for cf in cashflows:
        c = ql.as_coupon(cf)
        if c is not None:
            data.append((c.date(), c.rate(), c.nominal(), c.amount()))
        else:
```

```
            data.append((cf.date(), None, None, cf.amount()))

    return pd.DataFrame(
        data, columns=["date", "rate", "notional", "amount"]
    ).style.format(
        {"amount": "{:.2f}", "notional": "{:.2f}", "rate": "{:.2%}"}
    )
```

```
print_coupon_info(bond.cashflows())
```

	date	rate	notional	amount
0	August 10th, 2020	3.00%	100.00	1.50
1	February 8th, 2021	3.00%	100.00	1.48
2	August 9th, 2021	3.00%	100.00	1.51
3	February 8th, 2022	3.00%	100.00	1.49
4	August 8th, 2022	3.00%	100.00	1.50
5	February 8th, 2023	3.00%	100.00	1.50
6	August 8th, 2023	3.00%	100.00	1.50
7	February 8th, 2024	3.00%	100.00	1.50
8	August 8th, 2024	3.00%	100.00	1.50
9	February 10th, 2025	3.00%	100.00	1.52
10	February 10th, 2025	nan%	nan	100.00

```
print_coupon_info(swap.leg(0))
```

	date	rate	notional	amount
0	August 9th, 2021	3.00%	100.00	1.51
1	February 8th, 2022	3.00%	100.00	1.49
2	August 8th, 2022	3.00%	100.00	1.50
3	February 8th, 2023	3.00%	100.00	1.50
4	August 8th, 2023	3.00%	100.00	1.50
5	February 8th, 2024	3.00%	100.00	1.50
6	August 8th, 2024	3.00%	100.00	1.50
7	February 10th, 2025	3.00%	100.00	1.52
8	February 10th, 2025	nan%	nan	100.00

```
print_coupon_info(swap.leg(1))
```

	date	rate	notional	amount
0	August 10th, 2021	1.50%	103.89	0.33
1	February 10th, 2022	1.50%	103.89	0.80
2	August 10th, 2022	1.50%	103.89	0.78
3	February 10th, 2023	1.50%	103.89	0.80
4	August 10th, 2023	1.50%	103.89	0.78
5	February 12th, 2024	1.50%	103.89	0.81
6	August 12th, 2024	1.50%	103.89	0.79
7	February 10th, 2025	1.50%	103.89	0.79
8	February 10th, 2025	nan%	nan	103.89

When `par_asset_swap = True`, the floating-rate coupons are paid on a notional equal to 100 and the swap includes an upfront payment:

```
par_asset_swap = True

swap = ql.AssetSwap(
    pay_fixed,
    bond,
    bond_price,
    index,
    spread,
    ql.Schedule(),
    index.dayCounter(),
    par_asset_swap,
)
```

```
print_coupon_info(swap.leg(0))
```

	date	rate	notional	amount
0	August 9th, 2021	3.00%	100.00	1.51
1	February 8th, 2022	3.00%	100.00	1.49
2	August 8th, 2022	3.00%	100.00	1.50
3	February 8th, 2023	3.00%	100.00	1.50
4	August 8th, 2023	3.00%	100.00	1.50
5	February 8th, 2024	3.00%	100.00	1.50
6	August 8th, 2024	3.00%	100.00	1.50
7	February 10th, 2025	3.00%	100.00	1.52
8	February 10th, 2025	nan%	nan	100.00

```
print_coupon_info(swap.leg(1))
```

	date	rate	notional	amount
0	May 25th, 2021	nan%	nan	3.89
1	August 10th, 2021	1.50%	100.00	0.32
2	February 10th, 2022	1.50%	100.00	0.77
3	August 10th, 2022	1.50%	100.00	0.76
4	February 10th, 2023	1.50%	100.00	0.77
5	August 10th, 2023	1.50%	100.00	0.76
6	February 12th, 2024	1.50%	100.00	0.78
7	August 12th, 2024	1.50%	100.00	0.76
8	February 10th, 2025	1.50%	100.00	0.76
9	February 10th, 2025	nan%	nan	100.00

In both cases, once we give it an discounting engine, the swap can return more information.

```
discount_curve = ql.YieldTermStructureHandle(
    ql.FlatForward(today, 0.02, ql.Actual360())
)
swap.setPricingEngine(ql.DiscountingSwapEngine(discount_curve))
```

The `NPV` and `legNPV` methods return the value of the swap or of either leg. In this case we're paying the bond coupons, therefore the corresponding leg has a negative value.

```
print(swap.NPV())
print(swap.legNPV(0))
print(swap.legNPV(1))
```

```
-2.230481904331157
-104.26057030905751
102.03008840472636
```

It's also possible to retrieve the spread over the floating index that would make the swap fair:

```
fair_spread = swap.fairSpread()
print(fair_spread)
```

```
0.01117507740466585
```

We can test it by re-building the swap with this spread and asking for the NPV again:

```
swap = ql.AssetSwap(
    pay_fixed,
    bond,
```

```
    bond_price,
    index,
    fair_spread,
    ql.Schedule(),
    index.dayCounter(),
    par_asset_swap,
)
swap.setPricingEngine(ql.DiscountingSwapEngine(discount_curve))
```

```
print(swap.NPV())
print(swap.legNPV(0))
print(swap.legNPV(1))
```

```
0.0
-104.26057030905751
104.26057030905751
```

```
print_coupon_info(swap.leg(0))
```

	date	rate	notional	amount
0	August 9th, 2021	3.00%	100.00	1.51
1	February 8th, 2022	3.00%	100.00	1.49
2	August 8th, 2022	3.00%	100.00	1.50
3	February 8th, 2023	3.00%	100.00	1.50
4	August 8th, 2023	3.00%	100.00	1.50
5	February 8th, 2024	3.00%	100.00	1.50
6	August 8th, 2024	3.00%	100.00	1.50
7	February 10th, 2025	3.00%	100.00	1.52
8	February 10th, 2025	nan%	nan	100.00

```
print_coupon_info(swap.leg(1))
```

	date	rate	notional	amount
0	May 25th, 2021	nan%	nan	3.89
1	August 10th, 2021	2.12%	100.00	0.45
2	February 10th, 2022	2.12%	100.00	1.08
3	August 10th, 2022	2.12%	100.00	1.07
4	February 10th, 2023	2.12%	100.00	1.08
5	August 10th, 2023	2.12%	100.00	1.07
6	February 12th, 2024	2.12%	100.00	1.10

	date	rate	notional	amount
7	August 12th, 2024	2.12%	100.00	1.07
8	February 10th, 2025	2.12%	100.00	1.07
9	February 10th, 2025	nan%	nan	100.00

Asset swaps can be built based on other kinds of bonds besides fixed-rate ones; the resulting instances work the same way.

Chapter 18

Cross-currency swaps

At this time, there's no instrument class in the library modeling cross-currency swaps. However, it's possible to calculate their value by working with cashflows. Here is a short example.

```
import QuantLib as ql
import pandas as pd

today = ql.Date(27, ql.October, 2021)
ql.Settings.instance().evaluationDate = today

bps = 1e-4
```

Sample data

For the purposes of this notebook, we'll use mock rates. The data frame below holds made-up zero rates from a few different curves at a number of nodes…

```
sample_rates = pd.DataFrame(
    [
        (ql.Date(27, 10, 2021), 0.0229, 0.0893, -0.5490, -0.4869),
        (ql.Date(27, 1, 2022), 0.0645, 0.1059, -0.5584, -0.5057),
        (ql.Date(27, 4, 2022), 0.0414, 0.1602, -0.5480, -0.5236),
        (ql.Date(27, 10, 2022), 0.1630, 0.2601, -0.5656, -0.5030),
        (ql.Date(27, 10, 2023), 0.4639, 0.6281, -0.4365, -0.3468),
        (ql.Date(27, 10, 2024), 0.7187, 0.9270, -0.3500, -0.2490),
        (ql.Date(27, 10, 2025), 0.9056, 1.1257, -0.3041, -0.1590),
```

```
        (ql.Date(27, 10, 2026), 1.0673, 1.2821, -0.2340, -0.0732),
        (ql.Date(27, 10, 2027), 1.1615, 1.3978, -0.1690, -0.0331),
        (ql.Date(27, 10, 2028), 1.2326, 1.4643, -0.1041, 0.0346),
        (ql.Date(27, 10, 2029), 1.3050, 1.5589, -0.0070, 0.1263),
        (ql.Date(27, 10, 2030), 1.3584, 1.5986, 0.0272, 0.1832),
        (ql.Date(27, 10, 2031), 1.4023, 1.6488, 0.0744, 0.2599),
        (ql.Date(27, 10, 2036), 1.5657, 1.8136, 0.3011, 0.4406),
        (ql.Date(27, 10, 2041), 1.6191, 1.8749, 0.3882, 0.5331),
        (ql.Date(27, 10, 2046), 1.6199, 1.8701, 0.3762, 0.5225),
        (ql.Date(27, 10, 2051), 1.6208, 1.8496, 0.3401, 0.4926),
    ],
    columns=[
        "date",
        "SOFR",
        "USDLibor3M",
        "EUR-USD-discount",
        "Euribor3M",
    ],
)
```

…and this helper function uses them to create an interest-rate curve. Depending on the context, curves will be used for either forecasting or discounting.

```
def sample_curve(tag):
    curve = ql.ZeroCurve(
        sample_rates["date"], sample_rates[tag] / 100, ql.Actual365Fixed()
    )
    return ql.YieldTermStructureHandle(curve)
```

Const-notional cross-currency swaps

The first kind of cross-currency swaps we'll model is less common but simpler. The notionals are exchanged at the beginning (where they have the same value, given the exchange rate at that time) and again at the end. There is no rebalancing during the life of the swap. All the coupons in either leg have the same notional in the leg's payment currency.

In this example we'll model a 5-years swap paying quarterly coupons, based on 3M Euribor on one leg and 3M USD Libor on the other. We start by creating the indexes and their forecasting curves.

```
euribor3M_curve = sample_curve("Euribor3M")
euribor3M = ql.Euribor(ql.Period(3, ql.Months), euribor3M_curve)
```

```
usdlibor3M_curve = sample_curve("USDLibor3M")
usdlibor3M = ql.USDLibor(ql.Period(3, ql.Months), usdlibor3M_curve)
```

For each of the currencies, we create a corresponding sequence of cashflows, including the notional exchanges (which, unlike for vanilla swaps, don't cancel out—at least at maturity). The reference notional will be in dollars, converted in EUR at the present exchange rate (also made up). Just for kicks, we'll also add a spread to the USD leg.

```
notional = 1_000_000
fx_0 = 0.85
spread = 50.0 * bps
```

As I mentioned, the swaps make quarterly payments. The corresponding schedule starts spot and ends in five years.

```
calendar = ql.UnitedStates(ql.UnitedStates.FederalReserve)
start_date = calendar.advance(today, ql.Period(2, ql.Days))
end_date = calendar.advance(start_date, ql.Period(5, ql.Years))
tenor = ql.Period(3, ql.Months)
rule = ql.DateGeneration.Forward
convention = ql.Following
end_of_month = False

schedule = ql.Schedule(
    start_date,
    end_date,
    tenor,
    calendar,
    convention,
    convention,
    rule,
    end_of_month,
)
```

Since the notionals don't change, the two legs are similar: an initial lending of notional, the interest payments received according to the schedule and the index fixings, and the final payment of the notional.

```
usd_leg = (
    (ql.SimpleCashFlow(-notional, schedule[0]),)
    + ql.IborLeg(
        nominals=[notional],
        schedule=schedule,
        index=usdlibor3M,
```

```
        spreads=[spread],
    )
    + (ql.SimpleCashFlow(notional, schedule[-1]),)
)
```

For the EUR leg, of course, we'll have to convert the notional in the proper currency.

```
eur_leg = (
    (ql.SimpleCashFlow(-notional * fx_0, schedule[0]),)
    + ql.IborLeg(
        nominals=[notional * fx_0], schedule=schedule, index=euribor3M
    )
    + (ql.SimpleCashFlow(notional * fx_0, schedule[-1]),)
)
```

Now, we can get the NPV of each leg in its own currency by discounting them with the corresponding curve. For the USD leg, that would be the SOFR curve. For the EUR leg, ideally, a discount curve bootstrapped on cross-currency instruments so that it can capture the basis.

```
sofr_curve = sample_curve("SOFR")
usd_npv = ql.CashFlows.npv(usd_leg, sofr_curve, True)
usd_npv
```

```
35427.6553257405
```

```
eurusd_curve = sample_curve("EUR-USD-discount")
eur_npv = ql.CashFlows.npv(eur_leg, eurusd_curve, True)
eur_npv
```

```
6908.723028828779
```

And of course, we can convert the NPV of the EUR leg in USD:

```
ql.CashFlows.npv(eur_leg, eurusd_curve, True) / fx_0
```

```
8127.909445680917
```

We can also look at each cashflow by means of a short(ish) helper function:

```
def cashflow_data(leg):
    data = []
    for cf in sorted(leg, key=lambda c: c.date()):
        coupon = ql.as_floating_rate_coupon(cf)
        if coupon is None:
            data.append((cf.date(), None, None, cf.amount()))
```

```
        else:
            data.append(
                (
                    coupon.date(),
                    coupon.nominal(),
                    coupon.rate(),
                    coupon.amount(),
                )
            )
    return pd.DataFrame(
        data, columns=["date", "nominal", "rate", "amount"]
    ).style.format(
        {"amount": "{:.2f}", "nominal": "{:.2f}", "rate": "{:.2%}"}
    )
```

Here are the cashflows in USD…

```
cashflow_data(usd_leg)
```

	date	nominal	rate	amount
0	October 29th, 2021	nan	nan%	-1000000.00
1	January 31st, 2022	1000000.00	0.61%	1585.56
2	April 29th, 2022	1000000.00	0.72%	1750.57
3	July 29th, 2022	1000000.00	0.81%	2040.59
4	October 31st, 2022	1000000.00	0.91%	2386.92
5	January 30th, 2023	1000000.00	1.22%	3080.34
6	May 1st, 2023	1000000.00	1.40%	3541.30
7	July 31st, 2023	1000000.00	1.58%	3999.86
8	October 30th, 2023	1000000.00	1.76%	4444.64
9	January 29th, 2024	1000000.00	1.79%	4518.96
10	April 29th, 2024	1000000.00	1.93%	4890.80
11	July 29th, 2024	1000000.00	2.08%	5262.77
12	October 29th, 2024	1000000.00	2.22%	5682.53
13	January 29th, 2025	1000000.00	2.06%	5257.82
14	April 29th, 2025	1000000.00	2.16%	5388.64
15	July 29th, 2025	1000000.00	2.25%	5695.32
16	October 29th, 2025	1000000.00	2.35%	6000.94
17	January 29th, 2026	1000000.00	2.27%	5807.00
18	April 29th, 2026	1000000.00	2.35%	5873.71
19	July 29th, 2026	1000000.00	2.43%	6133.38
20	October 29th, 2026	1000000.00	2.50%	6388.32

	date	nominal	rate	amount
21	October 29th, 2026	nan	nan%	1000000.00

…and here are those in EUR.

```
cashflow_data(eur_leg)
```

	date	nominal	rate	amount
0	October 29th, 2021	nan	nan%	-850000.00
1	January 31st, 2022	850000.00	-0.50%	-1108.91
2	April 29th, 2022	850000.00	-0.53%	-1109.57
3	July 29th, 2022	850000.00	-0.49%	-1042.88
4	October 31st, 2022	850000.00	-0.46%	-1020.88
5	January 30th, 2023	850000.00	-0.30%	-644.90
6	May 1st, 2023	850000.00	-0.22%	-479.05
7	July 31st, 2023	850000.00	-0.15%	-314.08
8	October 30th, 2023	850000.00	-0.07%	-158.20
9	January 29th, 2024	850000.00	-0.12%	-266.58
10	April 29th, 2024	850000.00	-0.08%	-163.55
11	July 29th, 2024	850000.00	-0.03%	-60.50
12	October 29th, 2024	850000.00	0.02%	42.55
13	January 29th, 2025	850000.00	0.04%	96.24
14	April 29th, 2025	850000.00	0.09%	188.22
15	July 29th, 2025	850000.00	0.13%	284.92
16	October 29th, 2025	850000.00	0.18%	383.98
17	January 29th, 2026	850000.00	0.20%	443.61
18	April 29th, 2026	850000.00	0.25%	523.68
19	July 29th, 2026	850000.00	0.29%	619.73
20	October 29th, 2026	850000.00	0.33%	708.11
21	October 29th, 2026	nan	nan%	850000.00

To get the NPV of the swap, we add those of the two legs after converting the EUR leg back to dollars:

```
NPV = usd_npv - eur_npv / fx_0
NPV
```

```
27299.745880059585
```

Mark-to-market cross-currency swaps

In this more common kind of cross-currency swaps, the notionals are rebalanced at each coupon date so that their value remain the same (according, of course, to the value of the FX rate at each coupon start).

In order to model this rebalancing feature we will need to estimate the FX rates in the future, the corresponding notionals in Euro for the floating cashflows, and the amounts exchanged due to rebalancing.

The future FX rates can be forecast from the discount curves, since they model the cost of money. We'll write a convenience function to extract it at any given date:

```
def FX(date):
    return fx_0 * sofr_curve.discount(date) / eurusd_curve.discount(date)
```

The notionals at the start of each coupon can now be calculated from the FX rates:

```
start_dates = list(schedule)[:-1]

notionals = [notional * FX(d) for d in start_dates]
```

Given the notionals, we can also calculate the rebalancing cashflows:

```
rebalancing_cashflows = []
for i in range(len(notionals) - 1):
    rebalancing_cashflows.append(
        ql.SimpleCashFlow(notionals[i] - notionals[i + 1], schedule[i + 1])
    )
```

Finally, we can create the two legs and price them. The USD leg is as before, since its notional doesn't change:

```
usd_leg = (
    (ql.SimpleCashFlow(-notional, schedule[0]),)
    + ql.IborLeg(
        nominals=[notional],
        schedule=schedule,
        index=usdlibor3M,
        spreads=[spread],
    )
    + (ql.SimpleCashFlow(notional, schedule[-1]),)
)
```

```
cashflow_data(usd_leg)
```

	date	nominal	rate	amount
0	October 29th, 2021	nan	nan%	-1000000.00
1	January 31st, 2022	1000000.00	0.61%	1585.56
2	April 29th, 2022	1000000.00	0.72%	1750.57
3	July 29th, 2022	1000000.00	0.81%	2040.59
4	October 31st, 2022	1000000.00	0.91%	2386.92
5	January 30th, 2023	1000000.00	1.22%	3080.34
6	May 1st, 2023	1000000.00	1.40%	3541.30
7	July 31st, 2023	1000000.00	1.58%	3999.86
8	October 30th, 2023	1000000.00	1.76%	4444.64
9	January 29th, 2024	1000000.00	1.79%	4518.96
10	April 29th, 2024	1000000.00	1.93%	4890.80
11	July 29th, 2024	1000000.00	2.08%	5262.77
12	October 29th, 2024	1000000.00	2.22%	5682.53
13	January 29th, 2025	1000000.00	2.06%	5257.82
14	April 29th, 2025	1000000.00	2.16%	5388.64
15	July 29th, 2025	1000000.00	2.25%	5695.32
16	October 29th, 2025	1000000.00	2.35%	6000.94
17	January 29th, 2026	1000000.00	2.27%	5807.00
18	April 29th, 2026	1000000.00	2.35%	5873.71
19	July 29th, 2026	1000000.00	2.43%	6133.38
20	October 29th, 2026	1000000.00	2.50%	6388.32
21	October 29th, 2026	nan	nan%	1000000.00

The EUR leg, instead, changes notionals and thus it includes the rebalancing payments:

```
eur_leg = (
    [ql.SimpleCashFlow(-notionals[0], schedule[0])]
    + list(
        ql.IborLeg(nominals=notionals, schedule=schedule, index=euribor3M)
    )
    + rebalancing_cashflows
    + [ql.SimpleCashFlow(notionals[-1], schedule[-1])]
)
```

```
cashflow_data(eur_leg)
```

	date	nominal	rate	amount
0	October 29th, 2021	nan	nan%	-849973.31
1	January 31st, 2022	849973.31	-0.50%	-1108.87

	date	nominal	rate	amount
2	January 31st, 2022	nan	nan%	1361.41
3	April 29th, 2022	848611.90	-0.53%	-1107.76
4	April 29th, 2022	nan	nan%	1140.19
5	July 29th, 2022	847471.71	-0.49%	-1039.78
6	July 29th, 2022	nan	nan%	1688.83
7	October 31st, 2022	845782.88	-0.46%	-1015.82
8	October 31st, 2022	nan	nan%	2036.91
9	January 30th, 2023	843745.97	-0.30%	-640.16
10	January 30th, 2023	nan	nan%	1989.74
11	May 1st, 2023	841756.23	-0.22%	-474.40
12	May 1st, 2023	nan	nan%	2164.38
13	July 31st, 2023	839591.85	-0.15%	-310.23
14	July 31st, 2023	nan	nan%	2337.65
15	October 30th, 2023	837254.19	-0.07%	-155.83
16	October 30th, 2023	nan	nan%	2508.90
17	January 29th, 2024	834745.29	-0.12%	-261.80
18	January 29th, 2024	nan	nan%	2661.04
19	April 29th, 2024	832084.25	-0.08%	-160.10
20	April 29th, 2024	nan	nan%	2825.60
21	July 29th, 2024	829258.65	-0.03%	-59.02
22	July 29th, 2024	nan	nan%	2988.43
23	October 29th, 2024	826270.22	0.02%	41.36
24	October 29th, 2024	nan	nan%	3181.28
25	January 29th, 2025	823088.95	0.04%	93.19
26	January 29th, 2025	nan	nan%	3166.37
27	April 29th, 2025	819922.58	0.09%	181.56
28	April 29th, 2025	nan	nan%	3227.34
29	July 29th, 2025	816695.24	0.13%	273.75
30	July 29th, 2025	nan	nan%	3392.07
31	October 29th, 2025	813303.17	0.18%	367.40
32	October 29th, 2025	nan	nan%	3550.54
33	January 29th, 2026	809752.63	0.20%	422.60
34	January 29th, 2026	nan	nan%	3259.96
35	April 29th, 2026	806492.67	0.25%	496.88
36	April 29th, 2026	nan	nan%	3266.85
37	July 29th, 2026	803225.82	0.29%	585.63
38	July 29th, 2026	nan	nan%	3380.28
39	October 29th, 2026	799845.53	0.33%	666.33
40	October 29th, 2026	nan	nan%	799845.53

Finally, as in the previous case, we can calculate the NPV of each leg and of the swap:

```
usd_npv = ql.CashFlows.npv(usd_leg, sofr_curve, True)
usd_npv
```

```
35427.6553257405
```

```
eur_npv = ql.CashFlows.npv(eur_leg, eurusd_curve, True)
eur_npv
```

```
6678.756064412276
```

```
NPV = usd_npv - eur_npv / fx_0
NPV
```

```
27570.295249961353
```

Future developments

The above calculations work but are not very well integrated with the rest of the library; if you're familiar with QuantLib, you know that usually instruments can monitor market quotes and recalculate automatically when they change. This is not possible with the above; it will be up to you to recreate the cashflows when the FX rate or the curves are updated. A cross-currency swap modeled as an instrument would be way more convenient.

The reason we don't have such an instrument yet is that, unfortunately, the calculations above are a bit awkward to encapsulate in an instrument class. The constant-notional case is easy enough, but the mark-to-market case requires changing the notionals of the coupons when market quotes change, which is something our current classes were not prepared for. We'll have to think about it and try to add the required functionality in future releases.

Chapter 19

Payment-in-kind bonds

```
import QuantLib as ql
import pandas as pd

today = ql.Date(28, ql.April, 2023)
ql.Settings.instance().evaluationDate = today
```

In this kind of bonds, the interest matured during a coupon period is not paid off but added as additional notional. This doesn't fit very well the currently available coupon types, whose notional is supposed to be known upon construction. However, for fixed-rate PIK bonds we can work around the limitation.

A workaround

The idea is to build the coupons one by one, each time feeding into a coupon information from the previous one. Let's say we have the usual information for a bond:

```
start_date = ql.Date(8, ql.February, 2021)
maturity_date = ql.Date(8, ql.February, 2026)
frequency = ql.Semiannual
calendar = ql.TARGET()
convention = ql.Following
settlement_days = 3
coupon_rate = 0.03
day_counter = ql.Thirty360(ql.Thirty360.BondBasis)
face_amount = 10000
```

We first build the schedule, as we would do for a vanilla fixed-rate bond.

```
schedule = ql.Schedule(
    start_date,
    maturity_date,
    ql.Period(frequency),
    calendar,
    convention,
    convention,
    ql.DateGeneration.Backward,
    False,
)
```

Instead of using the usual classes or functions for building a bond or a whole fixed-rate leg, though, we'll start by building only the first coupon:

```
coupons = []
coupons.append(
    ql.FixedRateCoupon(
        calendar.adjust(schedule[1]),
        face_amount,
        coupon_rate,
        day_counter,
        schedule[0],
        schedule[1],
    )
)
```

The rest of the coupons can now be built one by one, each time taking the amount from the previous coupon and adding it to the notional:

```
for i in range(2, len(schedule)):
    previous = coupons[-1]
    coupons.append(
        ql.FixedRateCoupon(
            calendar.adjust(schedule[i]),
            previous.nominal() + previous.amount(),
            coupon_rate,
            day_counter,
            schedule[i - 1],
            schedule[i],
        )
    )
```

Finally, we can build a `Bond` instance by passing the list of coupons:

```
bond = ql.Bond(settlement_days, calendar, start_date, coupons)
```

We can check the resulting cash flows:

```
def coupon_info(cf):
    c = ql.as_coupon(cf)
    if not c:
        return (cf.date(), None, None, cf.amount())
    else:
        return (c.date(), c.nominal(), c.rate(), c.amount())


(
    pd.DataFrame(
        [coupon_info(c) for c in bond.cashflows()],
        columns=("date", "nominal", "rate", "amount"),
        index=range(1, len(bond.cashflows()) + 1),
    ).style.format(
        {"amount": "{:.2f}", "nominal": "{:.2f}", "rate": "{:.2%}"}
    )
)
```

	date	nominal	rate	amount
1	August 9th, 2021	10000.00	3.00%	150.83
2	August 9th, 2021	nan	nan%	-150.83
3	February 8th, 2022	10150.83	3.00%	151.42
4	February 8th, 2022	nan	nan%	-151.42
5	August 8th, 2022	10302.25	3.00%	154.53
6	August 8th, 2022	nan	nan%	-154.53
7	February 8th, 2023	10456.78	3.00%	156.85
8	February 8th, 2023	nan	nan%	-156.85
9	August 8th, 2023	10613.64	3.00%	159.20
10	August 8th, 2023	nan	nan%	-159.20
11	February 8th, 2024	10772.84	3.00%	161.59
12	February 8th, 2024	nan	nan%	-161.59
13	August 8th, 2024	10934.43	3.00%	164.02
14	August 8th, 2024	nan	nan%	-164.02
15	February 10th, 2025	11098.45	3.00%	168.33
16	February 10th, 2025	nan	nan%	-168.33
17	August 8th, 2025	11266.78	3.00%	167.12

	date	nominal	rate	amount
18	August 8th, 2025	nan	nan%	-167.12
19	February 9th, 2026	11433.90	3.00%	172.46
20	February 9th, 2026	nan	nan%	11433.90

For each date before maturity, we see two opposite payments: the 3% interest payment (with a positive sign since it's made to the holder of the bond) as well as a negative payment that models the same amount being immediately put by the holder into the bond.

We didn't create those payments explicitly; the `Bond` constructor created them automatically to account for the change in face amount between consecutive coupons. This feature was coded in order to generate amortizing payments in case of a decreasing face amount, but it works in the opposite direction just as well.

At maturity, we also have two payments; however, this time they are the 3% interest payment and the final reimbursement. Together, they give the final payment:

```
final_payment = (
    bond.cashflows()[-2].amount() + bond.cashflows()[-1].amount()
)
final_payment
```

```
11606.360684055619
```

One thing to note, though, is that asking the bond for its price might not give the expected result. Let's use a null rate for discounting, so we can spot the issue immediately:

```
discount_curve = ql.FlatForward(today, 0.0, ql.Actual365Fixed())

bond.setPricingEngine(
    ql.DiscountingBondEngine(ql.YieldTermStructureHandle(discount_curve))
)
bond.dirtyPrice()
```

```
109.35330081248894
```

Apart for the scaling to base 100, you might have expected the (undiscounted) bond value to equal the final amount. However, according to the convention for amortizing bonds, the `dirtyPrice` method also scales the price with respect to the current notional of the bond:

```
current_notional = bond.notional(bond.settlementDate())
current_notional
```

```
10613.635434706595
```

```
final_payment * 100 / current_notional
```

```
109.35330081248891
```

To avoid rescaling, we can use another method:

```
bond.settlementValue()
```

```
11606.36068405562
```

The `NPV` method would also work; the difference is that `NPV` discounts to the reference date of the discount curve (today's date, in this case) while `settlementValue` discounts to the settlement date of the bond.

Limitations

Of course, this works without problems if the coupon rate is fixed. For floating-rate bonds, we can use the same workaround; but the resulting cashflows and the bond will need to be thrown away and rebuilt when the forecasting curve changes, because their face amount will also change. This means that we can calculate one-off prices, but we won't be able to keep the bond and have it react when the curve changes, as vanilla floaters do.

Part V

Generic calculations

The infrastructure of the library allows us to perform some calculations in a generic way; that is, independent of the particular instruments we're pricing.

The idea is usually to bump the input data, or modifying them, or changing the evaluation date, and then to reprice our instruments. In the next few chapters, we'll see how to do it in an idiomatic way.

Chapter 20

Numerical Greeks

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 a given engine doesn't provide them.

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

```
import QuantLib as ql
```

```
today = ql.Date(8, ql.October, 2014)
ql.Settings.instance().evaluationDate = today
```

A somewhat exotic option

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

```
strike = 100.0
barrier = 120.0
rebate = 30.0

option = ql.BarrierOption(
    ql.Barrier.UpIn,
    barrier,
    rebate,
    ql.PlainVanillaPayoff(ql.Option.Call, strike),
    ql.EuropeanExercise(ql.Date(8, ql.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…

```
u = ql.SimpleQuote(100.0)
r = ql.SimpleQuote(0.01)
σ = ql.SimpleQuote(0.20)
```

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

```
riskFreeCurve = ql.FlatForward(
    0, ql.TARGET(), ql.QuoteHandle(r), ql.Actual360()
)
volatility = ql.BlackConstantVol(
    0, ql.TARGET(), ql.QuoteHandle(σ), ql.Actual360()
)
```

```
process = ql.BlackScholesProcess(
    ql.QuoteHandle(u),
    ql.YieldTermStructureHandle(riskFreeCurve),
    ql.BlackVolTermStructureHandle(volatility),
)
```

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

```
option.setPricingEngine(ql.AnalyticBarrierEngine(process))
```

Now we can ask the option for its value…

```
print(option.NPV())
```

```
29.24999528163375
```

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

```
try:
    print(option.delta())
except Exception as e:
    print(f"{type(e).__name__}: {e}")
```

```
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: 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, u_0 is the current value of the underlying and h is a tiny increment.

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

```
u0 = u.value()
h = 0.01
```

```
P0 = option.NPV()
print(P0)
```

```
29.24999528163375
```

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

```
u.setValue(u0 + h)
P_plus = option.NPV()
print(P_plus)
```

```
29.24851583696518
```

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

```
u.setValue(u0 - h)
P_minus = option.NPV()
print(P_minus)
```

```
29.25147220877793
```

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

```
u.setValue(u0)
```

Applying the formulas above give us the desired Greeks:

```
Δ = (P_plus - P_minus) / (2 * h)
Γ = (P_plus - 2 * P0 + P_minus) / (h * h)
print(Δ)
print(Γ)
```

```
-0.14781859063752734
-0.025175243933972524
```

The approach works for almost any Greeks. 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:

```
r0 = r.value()
h = 0.0001

r.setValue(r0 + h)
P_plus = option.NPV()
r.setValue(r0)

ρ = (P_plus - P0) / h
print(ρ)
```

```
-9.984440333461464
```

```
σ0 = σ.value()
h = 0.0001

σ.setValue(σ0 + h)
P_plus = option.NPV()
σ.setValue(σ0)

Vega = (P_plus - P0) / h
print(Vega)
```

```
-12.997818541258255
```

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 difference comes from the fact 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

can set it to tomorrow's date to get the corresponding option value. In this case, the value of the time increment h should be equivalent to one day:

```
ql.Settings.instance().evaluationDate = today + 1
P1 = option.NPV()
ql.Settings.instance().evaluationDate = today

h = 1.0 / 365
Θ = (P1 - P0) / h
print(Θ)
```

```
5.548998890566246
```

Why not in the engine?

We just saw that it's possible to calculate the missing ones numerically. So why don't engines use this method when Greeks are not available otherwise?

It's a good question; there are a few layers to it, and it can lead us to some insights into the architecture of the library.

The first answer is that numerical Greeks are costly: each one needs to reprice the instrument once or twice, depending on the accuracy we want. Therefore, we don't want them to be calculated by default. You should incur the cost only if you need them.

That doesn't need to stop us, though. We could configure the engine by passing argument to its constructor that specify which Greeks to calculate. It would add a bit of housekeeping, but it's doable, so let's assume we went ahead with the plan. (I won't bother showing you the code keeping track of the configuration. That's boring stuff, and moreover, I don't have it.)

How would we write the calculation of the Greeks, then? As an example, let's take the `MCEuropeanEngine` class. It takes as its first input a `shared_ptr` to a `GeneralizedBlackScholesProcess` (I'm talking of the C++ class, of course, since that's where we would implement the change); in turn, this contains a `Handle<Quote>` containing the value of the underlying, two `Handle<YieldTermStructure>` holding the risk-free rate and dividend yield curves, and a `Handle<BlackVolTermStructure>` holding the volatility. Those are the inputs we want to bump before repricing the option.

It turns out that it's fairly difficult to modify them. Even in the simplest case, that is, the `Handle<Quote>` containing the underlying, we have no guarantee that the contained `Quote` can be cast to a `SimpleQuote` and set a new value; it couls be another user-defined class. The same holds for the other handles; without knowledge of the actual derived class of the contained `YieldTermStructure` or `BlackVolTermStructure` instances, there's no way we

can modify them.

We don't actually need to do it, though, right? Let's take, e.g., the risk-free curve; we can wrap the current instance in, say, a `ZeroSpreadedTermStructure` instance to shift it upwards or backwards a given increment. Something like:

```
Handle<Quote> dr(make_shared<SimpleQuote>(1e-4)); // 1 bp

auto new_curve =
    make_shared<ZeroSpreadedTermStructure>(process_->riskFreeRate(), dr);
```

The same goes for the underlying quote: we can ask for the current value, perturb it, and store it in a new `SimpleQuote` instance, as in:

```
Real du = 0.001;

auto new_u =
    make_shared<SimpleQuote>(process_->x0() + du);
```

However, the problem now is that we can't put the new inputs into the process. It contains `Handle` instances, and those can't be relinked so that they store the new curve and quote; only `RelinkableHandle` instances can. Hmm. It seems like we went out of our way to make this operation difficult, doesn't it?

Well, in fact, we did—we wrote the code explicitly so that the inputs couldn't be changed. We had a reason for that, though: the instrument might not be the only one currently using them. If we had a bunch of options on the same underlying, for instance, they could be sharing the same process instance; and instruments in the same currency would probably share the same risk-free curve. There's no point creating and maintaining an instance for each instrument.

This means that any one instrument isn't free to modify its inputs: it could affect other, unrelated instruments. That's why inputs are only ever stored and made available through handles. (As a historical side note, we were once bitten by this before we started coding this way. In fact, we were bitten so badly we had to retire a release because it was too dangerous to use: version 0.3.5 was removed from the download page and version 0.3.6 replaced it as soon as we found out. You can check the fix here if you're curious.)

If you're determined enough and *really* want to add numerical Greeks to your engine, you can still find a way. Instead of modifying the process, you'll have to clone it. For instance, if you want to shift the risk-free rate, you can create a shifted curve as above and write:

```
auto new_process =
    make_shared<GeneralizedBlackScholesProcess>(
        process_->stateVariable(),
        process_->dividendYield(),
```

```
        Handle<YieldTermStructure>(new_curve),
        process_->blackVolatility());
```

after which the engine can perform the calculations using the cloned process.

You might also want to think what should happen if the Greeks calculations were to raise an exception. Do you throw away the calculated price, or keep it and just bail on the Greeks, and maybe try the next one? Granted, this requires some thought even for ordinary Greeks; but those are usually calculated together with the price, and it's more likely that they succeed or fail together.

In the end, though, we preferred to leave it to users to calculate missing Greeks. On the one hand, they can do it much more easily by perturbing the inputs that they set up and probably still hold on to; and on the other hand, unlike us, they're also able to optimize the calculation when multiple instruments depend on the same process. There's no need for each of them to create a new process; users can modify the relevant input once, read the new prices from all the instruments, and restore the input to its former value.

Chapter 21

Pricing over a range of days

Based on questions on *Stack Exchange* from Charles, bob.jonst, MCM and lcheng.

```
import QuantLib as ql
import pandas as pd
from matplotlib import pyplot as plt
```

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 rate information, 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.

```
prices = {}
```

Producing a single price

To price the bond on a single date, we set the evaluation date…

```
today = ql.Date(3, ql.May, 2021)
ql.Settings.instance().evaluationDate = today
```

…and we create the instrument itself…

```
start_date = ql.Date(8, ql.February, 2019)
maturity_date = start_date + ql.Period(5, ql.Years)
schedule = ql.Schedule(
```

```
    start_date,
    maturity_date,
    ql.Period(ql.Semiannual),
    ql.TARGET(),
    ql.Following,
    ql.Following,
    ql.DateGeneration.Backward,
    False,
)
coupons = [0.01]
bond = ql.FixedRateBond(
    3, 100, schedule, coupons, ql.Thirty360(ql.Thirty360.BondBasis)
)
```

…and the required discount curve. Here, I'm interpolating it over a set of tabulated rates, such as those I might have available from an external curve service.

```
data = [
    (ql.Period(0, ql.Months), 0.04),
    (ql.Period(6, ql.Months), 0.04),
    (ql.Period(1, ql.Years), 0.06),
    (ql.Period(2, ql.Years), 0.16),
    (ql.Period(3, ql.Years), 0.33),
    (ql.Period(5, ql.Years), 0.84),
    (ql.Period(7, ql.Years), 1.29),
    (ql.Period(10, ql.Years), 1.63),
    (ql.Period(20, ql.Years), 2.18),
    (ql.Period(30, ql.Years), 2.30),
]
```

```
dates = [today + p for p, _ in data]
rates = [r / 100 for _, r in data]

discount_curve = ql.ZeroCurve(dates, rates, ql.Actual365Fixed())
```

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

```
discount_handle = ql.RelinkableYieldTermStructureHandle(discount_curve)
bond.setPricingEngine(ql.DiscountingBondEngine(discount_handle))
```

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

```
101.94609091386468
```

Pricing on multiple days

Now, let's say for instance that we have rates for each day from January to April and we want to get the corresponding prices. For this notebook, they're stored in a file so I don't need to show you pages of numbers in order to define them. In the real world, they will probably be stored in a DB or some other resource.

```
df = pd.read_csv(
    "./data/multiple-yields.txt", delimiter="\t", parse_dates=["Date"]
)
df
```

	Date	0M	6M	1Y	2Y	3Y	5Y	7Y	10Y	20Y	30Y
0	2021-01-04	0.09	0.09	0.10	0.11	0.16	0.36	0.64	0.93	1.46	1.66
1	2021-01-05	0.09	0.09	0.10	0.13	0.17	0.38	0.66	0.96	1.49	1.70
2	2021-01-06	0.09	0.09	0.11	0.14	0.20	0.43	0.74	1.04	1.60	1.81
3	2021-01-07	0.09	0.09	0.11	0.14	0.22	0.46	0.78	1.08	1.64	1.85
4	2021-01-08	0.09	0.09	0.10	0.14	0.24	0.49	0.81	1.13	1.67	1.87
...	...	...	...	...	...	...	...	...	...	...	...
78	2021-04-26	0.04	0.04	0.06	0.18	0.35	0.85	1.27	1.58	2.13	2.24
79	2021-04-27	0.04	0.04	0.06	0.17	0.36	0.88	1.32	1.63	2.18	2.29
80	2021-04-28	0.04	0.04	0.05	0.17	0.35	0.86	1.31	1.63	2.19	2.29
81	2021-04-29	0.04	0.04	0.05	0.16	0.35	0.86	1.32	1.65	2.20	2.31
82	2021-04-30	0.03	0.03	0.05	0.16	0.35	0.86	1.32	1.65	2.19	2.30

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

For instance, here I'll calculate the price for the first row of data. As I mentioned, I need to set the new evaluation date:

```
d = ql.Date.from_date(df["Date"][0])
ql.Settings.instance().evaluationDate = d
```

Then I'll pick the corresponding rates and rebuild a discount curve:

```
tenors = list(df.columns[1:])
```

```
dates = []
rates = []
```

```
for tenor in tenors:
    dates.append(ql.Settings.instance().evaluationDate + ql.Period(tenor))
    rates.append(df[tenor][0] / 100)

discount_curve = ql.ZeroCurve(dates, rates, ql.Actual365Fixed())
```

Now I can link the handle in the engine to the new discount curve…

```
discount_handle.linkTo(discount_curve)
```

…after which the bond returns the updated price.

```
print(bond.cleanPrice())
```

```
102.55592512367953
```

By repeating the process, I can generate prices for the whole data set.

```
for _, row in df.iterrows():
    d = ql.Date.from_date(row["Date"])
    ql.Settings.instance().evaluationDate = d

    dates = []
    rates = []
    for tenor in tenors:
        dates.append(d + ql.Period(tenor))
        rates.append(row[tenor] / 100)
    discount_curve = ql.ZeroCurve(dates, rates, ql.Actual365Fixed())

    discount_handle.linkTo(discount_curve)

    prices[d] = bond.cleanPrice()
```

Here are the results.

```
dates, values = zip(*sorted(prices.items()))
```

```
ax = plt.figure(figsize=(9, 4)).add_subplot(1, 1, 1)
ax.plot([d.to_date() 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.

```
index = ql.Estr()
tenors = [ql.Period(i, ql.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 = ql.SimpleQuote(rate)
    h = ql.OISRateHelper(2, tenor, ql.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 always equal the evaluation date, whatever it happens to be at any given moment; while passing 2, for instance, would cause it to equal the corresponding spot date.

```
risk_free_curve = ql.PiecewiseFlatForward(
    0, ql.TARGET(), helpers, ql.Actual360()
)
```

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

```
spread = ql.SimpleQuote(0.01)
discount_curve = ql.ZeroSpreadedTermStructure(
    ql.YieldTermStructureHandle(risk_free_curve), ql.QuoteHandle(spread)
)
```

Now we can recalculate today's price:

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

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

```
96.48182194290018
```

Multiple days again

As before, I'll now assume to have quotes stored for the past four months; and again, I'll read the quotes from a file.

```
df = pd.read_csv(
    "./data/multiple-quotes.txt", delimiter="\t", parse_dates=["date"]
)
df
```

	date	1Y	2Y	3Y	4Y	5Y	6Y	7Y	8Y	9Y	10Y	spread
0	2021-01-04	0.60	0.77	0.90	0.97	1.25	1.26	1.42	1.62	1.70	1.76	0.31
1	2021-01-05	0.59	0.77	0.91	0.98	1.12	1.31	1.35	1.55	1.72	1.77	0.35
2	2021-01-06	0.59	0.80	0.87	0.98	1.18	1.26	1.39	1.58	1.77	1.74	0.33
3	2021-01-07	0.58	0.81	0.88	1.00	1.20	1.28	1.36	1.59	1.77	1.74	0.40
4	2021-01-08	0.60	0.77	0.92	1.00	1.15	1.28	1.42	1.58	1.68	1.80	0.36
...	...	...	...	...	...	...	...	...	...	...	...	...
78	2021-04-26	0.94	1.20	1.32	1.36	1.53	1.57	1.73	1.97	2.16	2.21	1.12
79	2021-04-27	0.96	1.15	1.29	1.34	1.65	1.63	1.79	2.06	2.11	2.19	0.99
80	2021-04-28	0.98	1.18	1.30	1.39	1.57	1.67	1.74	2.03	2.11	2.24	1.08
81	2021-04-29	0.96	1.20	1.28	1.33	1.61	1.63	1.80	2.00	2.14	2.29	0.97
82	2021-04-30	1.00	1.17	1.30	1.41	1.60	1.71	1.81	1.91	2.13	2.16	1.01

Now that we created the curve based on quotes, we don't need to build a new one at each step. Instead, we can set new values to the quotes and they will trigger the necessary recalculations in the curve and the instrument.

```
prices = {}

for _, row in df.iterrows():
    date = ql.Date.from_date(row["date"])

    for q, tenor in zip(
        quotes,
        ["1Y", "2Y", "3Y", "4Y", "5Y", "6Y", "7Y", "8Y", "9Y", "10Y"],
    ):
        q.setValue(row[tenor] / 100)

    spread.setValue(row["spread"] / 100)

    ql.Settings.instance().evaluationDate = date

    prices[date] = bond.cleanPrice()
```

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:

```
dates, values = zip(*sorted(prices.items()))

ax = plt.figure(figsize=(9, 4)).add_subplot(1, 1, 1)
```

```
ax.plot([d.to_date() 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…

```
forecast_handle = ql.YieldTermStructureHandle(risk_free_curve)
index = ql.Euribor6M(forecast_handle)

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

```
ql.Settings.instance().evaluationDate = today
for q, r in zip(quotes, rates):
    q.setValue(r)
spread.setValue(0.01)
```

…and try to price it. No joy.

```
try:
    print(bond.cleanPrice())
except Exception as e:
```

```
    print(f"{type(e).__name__}: {e}")
```

```
RuntimeError: Missing Euribor6M Actual/360 fixing for February 4th, 2021
```

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:

```
index.addFixing(ql.Date(4, ql.February, 2021), 0.005)

print(bond.cleanPrice())
```

```
97.09258058529258
```

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…

```
ql.Settings.instance().evaluationDate = ql.Date(1, ql.March, 2021)
print(bond.cleanPrice())

ql.Settings.instance().evaluationDate = ql.Date(15, ql.February, 2021)
print(bond.cleanPrice())
```

```
96.84054989393012
96.78778930005275
```

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

```
ql.Settings.instance().evaluationDate = ql.Date(1, ql.February, 2021)
try:
    print(bond.cleanPrice())
except Exception as e:
    print(f"{type(e).__name__}: {e}")
```

```
RuntimeError: Missing Euribor6M Actual/360 fixing for August 6th, 2020
```

Once we add it, the calculation works again.

```
index.addFixing(ql.Date(6, ql.August, 2020), 0.004)
print(bond.cleanPrice())
```

```
96.97859587521839
```

(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 past fixings

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.

```
ql.Settings.instance().evaluationDate = ql.Date(1, ql.June, 2021)

print(bond.cleanPrice())
```

```
97.2097212826188
```

However, there's another problem, as pointed out by Mariano Zeron 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.

```
ql.Settings.instance().evaluationDate = ql.Date(1, ql.September, 2021)
try:
    print(bond.cleanPrice())
except Exception as e:
    print(f"{type(e).__name__}: {e}")
```

```
RuntimeError: Missing Euribor6M Actual/360 fixing for August 5th, 2021
```

Here the curve starts on September 1st 2021, 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:

```
ql.Settings.instance().evaluationDate = today

fixing_date = ql.Date(5, ql.August, 2021)
future_fixing = index.fixing(fixing_date)
print(future_fixing)
index.addFixing(fixing_date, future_fixing)
```

```
0.009974987403789588
```

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

```
ql.Settings.instance().evaluationDate = ql.Date(1, ql.September, 2021)

print(bond.cleanPrice())
```

```
97.55258650460641
```

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 2022 will be retrieved if the evaluation date is later…

```
index.addFixing(ql.Date(3, ql.February, 2022), 0.02)

ql.Settings.instance().evaluationDate = ql.Date(1, ql.June, 2022)
print(index.fixing(ql.Date(3, ql.February, 2022)))
```

```
0.02
```

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

```
ql.Settings.instance().evaluationDate = today
print(index.fixing(ql.Date(3, ql.February, 2022)))
```

```
0.012063742194402307
```

Chapter 22

Assessing duration risk

(Originally published as an article in Wilmott Magazine, July 2023.)

As I write this, the fall of Silicon Valley Bank or Credit Suisse is still pretty fresh; which prompted me to write about the several ways we can estimate interest-rate risk using QuantLib.

```
import QuantLib as ql
import pandas as pd
import numpy as np
```

Setting up

Let's imagine for a bit that we're back to that magic time where interest rates were low and deposits were plentiful; for instance, March 17th, 2022. After setting QuantLib's global evaluation date accordingly, I'll define a list holding the tenors and par rates conveniently made available on the Treasury web site for our evaluation date.

```
today = ql.Date(17, 3, 2022)
ql.Settings.instance().evaluationDate = today
```

```
tenors = [
    "1M",
    "2M",
    "3M",
    "6M",
    "1Y",
```

```
    "2Y",
    "3Y",
    "5Y",
    "7Y",
    "10Y",
    "20Y",
    "30Y",
]
```

```
yields = [
    0.20,
    0.30,
    0.40,
    0.81,
    1.30,
    1.94,
    2.14,
    2.17,
    2.22,
    2.20,
    2.60,
    2.50,
]
```

Today's discount curve

Let's now use those quotes to bootstrap a discount curve. I'll keep it simple, but I'll be very interested in any improvements you might suggest to the procedure, so please, do reach out to me if you have comments. Here, I'm going to create an instance of `FixedRateBondHelper` for each of the quotes; it's an object that holds the terms and conditions of a fixed-rate bond, its quoted price, and has methods for recalculating said price off a discount curve and reporting the discrepancy with the quote.

```
calendar = ql.UnitedStates(ql.UnitedStates.GovernmentBond)
settlement_days = 1
face_amount = 100.0
day_counter = ql.Thirty360(ql.Thirty360.BondBasis)
```

```
helpers = []
for t, y in zip(tenors, yields):
    schedule = ql.Schedule(
        today,
```

```
            today + ql.Period(t),
            ql.Period(ql.Semiannual),
            calendar,
            ql.Unadjusted,
            ql.Unadjusted,
            ql.DateGeneration.Backward,
            False,
        )
        helpers.append(
            ql.FixedRateBondHelper(
                ql.QuoteHandle(ql.SimpleQuote(100)),
                settlement_days,
                face_amount,
                schedule,
                [y / 100.0],
                day_counter,
            )
        )
```

I have a confession to make, though: this helper wasn't exactly written for this case. You might have noticed that I mentioned the quoted price of a bond in the previous paragraph; however, what's actually quoted in this case is the par yield. For lack of a more specialized helper class, we'll make do by passing a price of 100 as the quote and by setting the yield as the coupon rate. However, that's getting the logic rather backwards; and, when the yields change, it forces one to create new helpers and a new curve, instead of just setting a new value to the quote.

Of course, it would be possible to create a par-rate helper that models this particular case, takes the yield as a quote, and matches it with the one recalculated off the curve being bootstrapped. Its disadvantage? It would be slower: obtaining a bond price given a curve is a direct calculation, but obtaining its yield involves a root-solving process. We might implement that in the future and let users decide about the trade-off; for the time being, such a helper is not available.

Anyway: for each of the quotes, creating the helper involves first building a coupon schedule based on the quoted tenor, and then passing it to the helper constructor together with the quote for the price and other bond parameters.

Once we have the helpers, we can use them to create our discount curve; namely, an instance of `PiecewiseYieldCurve` which will run the bootstrapping process. In the interest of keeping my word count within reasonable limits, I won't describe it here; you can have a look at my *Implementing QuantLib* book for all the (many) details.

```
curve = ql.PiecewiseLinearZero(today, helpers, ql.Actual360())
```

```
curve.enableExtrapolation()
```

A sample bond portfolio

Now, whose risk should we assess? In the next part of the code, I'll create a mock portfolio; that is, a list of bonds with different coupon rates and maturities. For brevity, I'll assume that all bonds have the same frequency and conventions, as well as a common face amount of 100.

```
data = [
    (ql.Date(15, 9, 2016), ql.Date(15, 9, 2022), 1.45),
    (ql.Date(1, 11, 2017), ql.Date(1, 11, 2022), 5.5),
    (ql.Date(1, 8, 2021), ql.Date(1, 8, 2023), 4.75),
    (ql.Date(1, 12, 2008), ql.Date(1, 12, 2024), 2.5),
    (ql.Date(1, 6, 2020), ql.Date(1, 6, 2027), 2.2),
    (ql.Date(1, 11, 2009), ql.Date(1, 11, 2029), 5.25),
    (ql.Date(1, 4, 2016), ql.Date(1, 4, 2030), 1.35),
    (ql.Date(15, 9, 2012), ql.Date(15, 9, 2032), 1.33),
    (ql.Date(1, 3, 2012), ql.Date(1, 3, 2035), 3.35),
    (ql.Date(1, 2, 2017), ql.Date(1, 2, 2037), 4.0),
    (ql.Date(1, 8, 2007), ql.Date(1, 8, 2039), 3.0),
    (ql.Date(15, 9, 2006), ql.Date(15, 9, 2041), 2.97),
    (ql.Date(1, 9, 2018), ql.Date(1, 9, 2044), 3.75),
    (ql.Date(1, 3, 2013), ql.Date(1, 3, 2048), 3.45),
    (ql.Date(1, 9, 2012), ql.Date(1, 9, 2049), 3.85),
    (ql.Date(1, 9, 2007), ql.Date(1, 9, 2051), 1.7),
    (ql.Date(1, 3, 2016), ql.Date(1, 3, 2067), 2.8),
    (ql.Date(1, 3, 2020), ql.Date(1, 3, 2072), 2.15),
]
```

Since we're going to price all these bonds off the same curve, the code first creates a relinkable handle to hold the curve and then uses the handle to create a pricing engine. During the calculation, the `DiscountingBondEngine` class does what you would expect: it adds the discounted values of the bond cash flows.

```
discount_handle = ql.RelinkableYieldTermStructureHandle()
engine = ql.DiscountingBondEngine(discount_handle)
```

The next step is to create the actual bonds. The code is pretty similar to the loop that created the curve helpers earlier, except that it's creating instances of the `Bond` class now. The same engine can be set to every bond; there's no need for separate engines as long as we don't

think about multithreading, and that's a bad idea in QuantLib anyway.

```
bonds = []
for start, end, coupon in data:
    schedule = ql.Schedule(
        start,
        end,
        ql.Period(ql.Semiannual),
        calendar,
        ql.Unadjusted,
        ql.Unadjusted,
        ql.DateGeneration.Backward,
        False,
    )
    bond = ql.FixedRateBond(
        settlement_days,
        face_amount,
        schedule,
        [coupon / 100.0],
        day_counter,
    )
    bond.setPricingEngine(engine)
    bonds.append(bond)
```

Reference prices

Once the bonds are created and the engine is set, we can finally calculate their prices. By linking our treasury curve to the discount handle, we make it available to the pricing engine. All that remains is to loop over the bonds and ask each of them for its price. The results are stored in a Pandas data frame, in the "price" column.

```
discount_handle.linkTo(curve)
```

```
results = []
for b in bonds:
    m = b.maturityDate()
    p = b.cleanPrice()
    results.append((m, p))
results = pd.DataFrame(results, columns=["maturity", "price"])
results
```

	maturity	price
0	September 15th, 2022	100.325987
1	November 1st, 2022	102.821079
2	August 1st, 2023	104.341431
3	December 1st, 2024	101.086154
4	June 1st, 2027	100.118562
5	November 1st, 2029	121.230541
6	April 1st, 2030	93.664362
7	September 15th, 2032	91.673963
8	March 1st, 2035	111.512720
9	February 1st, 2037	120.086115
10	August 1st, 2039	107.116189
11	September 15th, 2041	106.031770
12	September 1st, 2044	120.219158
13	March 1st, 2048	117.343050
14	September 1st, 2049	126.277776
15	September 1st, 2051	83.218728
16	March 1st, 2067	111.000187
17	March 1st, 2072	93.543566

Price sensitivity

And now, we can start talking about assessing risk. How can we calculate, for instance, the change in price corresponding to a change of one basis point in yield? We ask each bond for its yield, and then for its price given the perturbed yield. (We could also average the changes for a positive and negative yield movement, but I'll keep it simple here.) The difference between the perturbed prices and the reference prices gives us the "sensitivity" column in table 1.

```
prices = []
for b in bonds:
    y = b.bondYield(day_counter, ql.Compounded, ql.Semiannual)
    prices.append(
        b.cleanPrice(y + 0.0001, day_counter, ql.Compounded, ql.Semiannual)
    )
results["sensitivity"] = np.array(prices) - results["price"]
results
```

	maturity	price	sensitivity
0	September 15th, 2022	100.325987	-0.004914
1	November 1st, 2022	102.821079	-0.006332
2	August 1st, 2023	104.341431	-0.013913
3	December 1st, 2024	101.086154	-0.026352
4	June 1st, 2027	100.118562	-0.048967
5	November 1st, 2029	121.230541	-0.078068
6	April 1st, 2030	93.664362	-0.070618
7	September 15th, 2032	91.673963	-0.088716
8	March 1st, 2035	111.512720	-0.118587
9	February 1st, 2037	120.086115	-0.138884
10	August 1st, 2039	107.116189	-0.146022
11	September 15th, 2041	106.031770	-0.157851
12	September 1st, 2044	120.219158	-0.190570
13	March 1st, 2048	117.343050	-0.209395
14	September 1st, 2049	126.277776	-0.229316
15	September 1st, 2051	83.218728	-0.185209
16	March 1st, 2067	111.000187	-0.294056
17	March 1st, 2072	93.543566	-0.278230

An alternative method

Another method for calculating the same sensitivity is to modify the discount curve and then recalculate bonds prices; this can be useful, say, for instruments such as swaps that are not priced in terms of a yield. One way is to use the `ZeroSpreadedTermStructure` class, which takes a reference term structure and adds a parallel spread to its zero rates (the default, as in this case, is to work on continuous rates; other conventions can be specified).

Here, we're passing the reference curve and a shift of one basis point, resulting in a new shifted curve that we can link to the discount handle; after doing that, asking once again each bond for its price will return updated results based on the new curve. The changes with respect to the original prices give us the alternative sensitivity of our bonds.

```
discount_handle.linkTo(
    ql.ZeroSpreadedTermStructure(
        ql.YieldTermStructureHandle(curve),
        ql.QuoteHandle(ql.SimpleQuote(0.0001)),
    )
)
```

```
results["sensitivity (alt.)"] = (
    np.array([b.cleanPrice() for b in bonds]) - results["price"]
)
results
```

	maturity	price	sensitivity	sensitivity (alt.)
0	September 15th, 2022	100.325987	-0.004914	-0.005045
1	November 1st, 2022	102.821079	-0.006332	-0.006505
2	August 1st, 2023	104.341431	-0.013913	-0.014248
3	December 1st, 2024	101.086154	-0.026352	-0.027058
4	June 1st, 2027	100.118562	-0.048967	-0.050229
5	November 1st, 2029	121.230541	-0.078068	-0.080078
6	April 1st, 2030	93.664362	-0.070618	-0.072447
7	September 15th, 2032	91.673963	-0.088716	-0.091026
8	March 1st, 2035	111.512720	-0.118587	-0.121531
9	February 1st, 2037	120.086115	-0.138884	-0.142264
10	August 1st, 2039	107.116189	-0.146022	-0.149421
11	September 15th, 2041	106.031770	-0.157851	-0.161291
12	September 1st, 2044	120.219158	-0.190570	-0.194415
13	March 1st, 2048	117.343050	-0.209395	-0.213709
14	September 1st, 2049	126.277776	-0.229316	-0.234061
15	September 1st, 2051	83.218728	-0.185209	-0.189618
16	March 1st, 2067	111.000187	-0.294056	-0.302897
17	March 1st, 2072	93.543566	-0.278230	-0.287335

A flattening scenario

The spread that we add to the reference curve doesn't need to be constant; the `PiecewiseZeroSpreadedTermStructure` class allows us to specify a set of spreads corresponding to different dates, and then interpolates between them.

In this case, we'll create a fairly simple set of spreads: a short-term positive spread of 0.2% at the value date and a long-term negative spread of 0.2% at the 60-years point. Added to our reference curve, the interpolated time-dependent spread creates a flattened curve, with higher short-term rates and lower long-term rates.

```
short_term_spread = ql.SimpleQuote(0.002)
long_term_spread = ql.SimpleQuote(-0.002)

dates = [today, today + ql.Period(60, ql.Years)]
spreads = [
```

```
    ql.QuoteHandle(short_term_spread),
    ql.QuoteHandle(long_term_spread),
]

tilted_curve = ql.PiecewiseZeroSpreadedTermStructure(
    ql.YieldTermStructureHandle(curve), spreads, dates
)
tilted_curve.enableExtrapolation()
```

Once we have the new curve, the process is the same: we link it to the discount handle, extract the new prices, and take the differences.

```
results = results[["maturity", "price"]].copy()

discount_handle.linkTo(tilted_curve)

results["flattening"] = (
    np.array([b.cleanPrice() for b in bonds]) - results["price"]
)
results
```

	maturity	price	flattening
0	September 15th, 2022	100.325987	-0.099161
1	November 1st, 2022	102.821079	-0.127298
2	August 1st, 2023	104.341431	-0.271664
3	December 1st, 2024	101.086154	-0.491554
4	June 1st, 2027	100.118562	-0.830087
5	November 1st, 2029	121.230541	-1.212212
6	April 1st, 2030	93.664362	-1.061867
7	September 15th, 2032	91.673963	-1.191191
8	March 1st, 2035	111.512720	-1.442707
9	February 1st, 2037	120.086115	-1.549767
10	August 1st, 2039	107.116189	-1.394847
11	September 15th, 2041	106.031770	-1.319961
12	September 1st, 2044	120.219158	-1.339476
13	March 1st, 2048	117.343050	-1.072036
14	September 1st, 2049	126.277776	-1.041110
15	September 1st, 2051	83.218728	-0.411631
16	March 1st, 2067	111.000187	1.375404
17	March 1st, 2072	93.543566	2.220772

A steepening scenario

Changing the values of the short-term and long-term spreads to -0.2% and 0.2%, respectively, gives us a steeper curve where the short-term rates decrease and the long-term rates increase. The changes are calculated as usual.

```
short_term_spread.setValue(-0.002)
long_term_spread.setValue(0.002)

results["steepening"] = (
    np.array([b.cleanPrice() for b in bonds]) - results["price"]
)
results
```

	maturity	price	flattening	steepening
0	September 15th, 2022	100.325987	-0.099161	0.099260
1	November 1st, 2022	102.821079	-0.127298	0.127456
2	August 1st, 2023	104.341431	-0.271664	0.272379
3	December 1st, 2024	101.086154	-0.491554	0.493991
4	June 1st, 2027	100.118562	-0.830087	0.837225
5	November 1st, 2029	121.230541	-1.212212	1.225468
6	April 1st, 2030	93.664362	-1.061867	1.074386
7	September 15th, 2032	91.673963	-1.191191	1.207413
8	March 1st, 2035	111.512720	-1.442707	1.463129
9	February 1st, 2037	120.086115	-1.549767	1.571870
10	August 1st, 2039	107.116189	-1.394847	1.414609
11	September 15th, 2041	106.031770	-1.319961	1.337761
12	September 1st, 2044	120.219158	-1.339476	1.355756
13	March 1st, 2048	117.343050	-1.072036	1.083442
14	September 1st, 2049	126.277776	-1.041110	1.052151
15	September 1st, 2051	83.218728	-0.411631	0.416085
16	March 1st, 2067	111.000187	1.375404	-1.286171
17	March 1st, 2072	93.543566	2.220772	-2.052258

A clairvoyant stress scenario

Finally, instead of modifying the current curve, we can forget about it and replace it with an entirely different one. For instance, back in March 2022 (when we're still pretending to be, and when our current evaluation date is set), you might have said: "What if the Fed increased the rates by 5% in one year and reversed the curve?" to which some colleague might have answered: "Not gonna happen, but sure, let's check it out—I'm curious."

In this case, you would have needed to guess a new set of par rates; for instance, the ones shown in the code and corresponding to the quotes for March 17th, 2023. As I mentioned, in this setup we can't simply set new values to the existing helpers, so we have to create new helpers and a new curve, in the same way in which we created the reference curve. In real-world code, to avoid repetition, we would of course abstract those lines out in a function and call it with our different sets of quotes.

```
new_yields = [
    4.31,
    4.51,
    4.52,
    4.71,
    4.26,
    3.81,
    3.68,
    3.44,
    3.45,
    3.39,
    3.76,
    3.60,
]
```

```
helpers = []
for t, y in zip(tenors, new_yields):
    schedule = ql.Schedule(
        today,
        today + ql.Period(t),
        ql.Period(ql.Semiannual),
        ql.UnitedStates(ql.UnitedStates.GovernmentBond),
        ql.Unadjusted,
        ql.Unadjusted,
        ql.DateGeneration.Backward,
        False,
    )
    helpers.append(
        ql.FixedRateBondHelper(
            ql.QuoteHandle(ql.SimpleQuote(100)),
            1,
            100.0,
            schedule,
            [y / 100.0],
            ql.Thirty360(ql.Thirty360.BondBasis),
        )
```

```
    )
```

```
new_curve = ql.PiecewiseLinearZero(today, helpers, ql.Actual360())
new_curve.enableExtrapolation()
```

In any case, once we have our stressed curve, we can link it to the same old discount handle and ask the bonds for their prices again. "Ouch," you would say to your colleague as the code prints out the results.

```
results = results[["maturity", "price"]].copy()

discount_handle.linkTo(new_curve)

results["precog"] = (
    np.array([b.cleanPrice() for b in bonds]) - results["price"]
)
results
```

	maturity	price	precog
0	September 15th, 2022	100.325987	-1.866123
1	November 1st, 2022	102.821079	-2.264614
2	August 1st, 2023	104.341431	-3.485236
3	December 1st, 2024	101.086154	-4.201212
4	June 1st, 2027	100.118562	-5.962030
5	November 1st, 2029	121.230541	-9.219825
6	April 1st, 2030	93.664362	-8.097490
7	September 15th, 2032	91.673963	-9.836337
8	March 1st, 2035	111.512720	-13.110537
9	February 1st, 2037	120.086115	-15.242371
10	August 1st, 2039	107.116189	-15.746457
11	September 15th, 2041	106.031770	-16.829242
12	September 1st, 2044	120.219158	-19.800276
13	March 1st, 2048	117.343050	-20.953544
14	September 1st, 2049	126.277776	-22.683295
15	September 1st, 2051	83.218728	-17.548003
16	March 1st, 2067	111.000187	-25.273680
17	March 1st, 2072	93.543566	-23.000832

Other possibilities

The classes shown here will enable you to create other kind of scenarios. For instance, `PiecewiseZeroSpreadedTermStructure` could let you generate some kind of historical stress scenario by selecting some interesting or stressful past period and a set of key rates (say, the 1-, 2-, 5-, 10-, 20- and 50-years zero rates, or as many or few as you like). By taking the variations of those key rates over the period and applying them as spreads, you'd obtain a stressed curve to use for discounting. Or again, by adding a predetermined spread only in a specific region of the curve, you could calculate the corresponding key-rate durations.

In short: you can use the building blocks provided by QuantLib to create your own bespoke solution to the problem of monitoring duration risk. Have fun putting the pieces together.

Part VI

Inflation

Unlike interest-rate indexes, inflation indexes have a considerable lag between the period they refer to and the date they're published; and also unlike interest-rate indexes, they're published monthly or quarterly, which often means they are interpolated to obtain a value for a particular date.

This has significant consequences in the interface of both the indexes and their forecast curves; and in some cases, we're still trying to figure out what the best interface should be. In the next chapters, we'll have a look at the current state of the relevant classes.

Chapter 23

Inflation indexes and curves

```
import QuantLib as ql
import pandas as pd
```

```
today = ql.Date(11, ql.May, 2024)
ql.Settings.instance().evaluationDate = today
```

Inflation indexes

The library provides classes to model some predefined inflation indexes, such as `EUHICP` or `UKRPI`; any missing one can be defined using the base class `ZeroInflationIndex`.

```
index = ql.EUHICP()
```

Historical fixings can be saved (for all instances of the same index) by calling the `addFixing` method.

The method is inherited from the base `Index` class, so it requires a specific date even if an inflation fixing corresponds to a whole month. By convention, the date we'll pass together with a fixing must be the first day of the corresponding month (not the day of publishing, which would be in the following month).

```
inflation_fixings = [
    ((2022, ql.January), 110.70),
    ((2022, ql.February), 111.74),
    ((2022, ql.March), 114.46),
    ((2022, ql.April), 115.11),
```

```
    ((2022, ql.May), 116.07),
    ((2022, ql.June), 117.01),
    ((2022, ql.July), 117.14),
    ((2022, ql.August), 117.85),
    ((2022, ql.September), 119.26),
    ((2022, ql.October), 121.03),
    ((2022, ql.November), 120.95),
    ((2022, ql.December), 120.52),
    ((2023, ql.January), 120.27),
    ((2023, ql.February), 121.24),
    ((2023, ql.March), 122.34),
    ((2023, ql.April), 123.12),
    ((2023, ql.May), 123.15),
    ((2023, ql.June), 123.47),
    ((2023, ql.July), 123.36),
    ((2023, ql.August), 124.03),
    ((2023, ql.September), 124.43),
    ((2023, ql.October), 124.54),
    ((2023, ql.November), 123.85),
    ((2023, ql.December), 124.05),
    ((2024, ql.January), 123.60),
    ((2024, ql.February), 124.37),
    ((2024, ql.March), 125.31),
    ((2024, ql.April), 126.05),
]

for (year, month), fixing in inflation_fixings:
    index.addFixing(ql.Date(1, month, year), fixing)
```

Asking for a fixing for any past date will return the fixing for the month:

```
index.fixing(ql.Date(15, ql.March, 2024))
```

```
125.31
```

Of course, some past dates still don't have an inflation fixing available: at the time of this writing, in the middle of May 2024, the index for this month is not published yet. Fixings for these dates, as well as for future dates, will need to be forecast: and for that, we require an inflation term structure.

Another note: in the past, the index could also return interpolated fixings; however, the result was not always right, since the index didn't have the correct information on the number of days to use while interpolating. In version 1.29, the interpolation logic was moved into

inflation coupons, where the information is available; the corresponding logic in the index was deprecated in the same release and removed in version 1.34.

The logic behind the interpolation is also available as a static method in the `CPI` class, used by inflation coupons:

```
observation_lag = ql.Period(3, ql.Months)

ql.CPI.laggedFixing(
    index, ql.Date(15, ql.May, 2024), observation_lag, ql.CPI.Linear
)
```

```
124.79451612903226
```

For instance, the call above interpolates linearly a fixing for May 15th, 2024 with an observation lag of three months. This means that the fixings to be interpolated will be those for February and March 2024, but their weights in the interpolation will be based on the number of days between May 1st and May 15th and between May 15th and June 1st.

Inflation curves

As for other kinds of term structures, it's possible to create inflation curves by bootstrapping over a number of quoted instruments; at this time, the library provides helpers for zero-coupon inflation swaps. The information needed to build them includes, besides more common data such as the calendar and day count convention, an observation lag for the fixing of the underlying inflation index and the interpolation to be used between fixings.

The helpers will also need an external nominal curve of interest rates. This used to be stored into the inflation curve itself, but for consistency with other helpers and term structures it was moved into the helpers in version 1.15.

```
inflation_quotes = [
    (ql.Period(1, ql.Years), 2.93),
    (ql.Period(2, ql.Years), 2.95),
    (ql.Period(3, ql.Years), 2.965),
    (ql.Period(4, ql.Years), 2.98),
    (ql.Period(5, ql.Years), 3.0),
    (ql.Period(7, ql.Years), 3.06),
    (ql.Period(10, ql.Years), 3.175),
    (ql.Period(12, ql.Years), 3.243),
    (ql.Period(15, ql.Years), 3.293),
    (ql.Period(20, ql.Years), 3.338),
    (ql.Period(25, ql.Years), 3.348),
    (ql.Period(30, ql.Years), 3.348),
```

```
    (ql.Period(40, ql.Years), 3.308),
    (ql.Period(50, ql.Years), 3.228),
]

calendar = ql.TARGET()
observation_lag = ql.Period(3, ql.Months)
day_counter = ql.Thirty360(ql.Thirty360.BondBasis)
interpolation = ql.CPI.Linear

nominal_curve = ql.YieldTermStructureHandle(
    ql.FlatForward(today, 0.03, ql.Actual365Fixed())
)

helpers = []

for tenor, quote in inflation_quotes:
    maturity = calendar.advance(today, tenor)
    helpers.append(
        ql.ZeroCouponInflationSwapHelper(
            ql.makeQuoteHandle(quote / 100),
            observation_lag,
            maturity,
            calendar,
            ql.Following,
            day_counter,
            index,
            interpolation,
            nominal_curve,
        )
    )
```

Now, inflation curves are kind of odd compared to other curves. Usually, term structures in QuantLib (e.g. for interest rates or default probabilities) forecast their underlying quantities starting from today; yesterday's rates are assumed to be known, and so is whether an issuer defaulted. As I mentioned, though, inflation curves can also be used to forecast still unpublished fixings corresponding to past dates. Therefore, they have a so-called base date in the past which separates known and unknown fixings.

Up to version 1.33, though, the base date was not specified explicitly. Instead, the constructors of most inflation curves required an observation lag, probably because of some past confusion with the observation lag of the instruments used for bootstrapping; the base date was calculated by starting from today's date, subtracting the lag and taking the first date of

the resulting month.

However, the only base date that makes sense is the date of the last available inflation fixing: the fixings are known up to that point and need to be forecast after it. Therefore, the observation lag could not be a constant attribute of a given curve, but had to be calculated based on the available data: at the beginning of May, when the April fixing was not published yet, we needed a lag of two months to get to March, while mid-May we had to switch to a lag of one month to get an April base date as soon as the corresponding fixing was published. The curve also needed a base rate to be used for $t = 0$; in the current implementation, the base rate is still required when using the old constructor but is ignored by the bootstrapping process.

```
availability_lag = ql.Period(1, ql.Months)
fixing_frequency = ql.Monthly
base_rate = 0.029

inflation_curve = ql.PiecewiseZeroInflation(
    today,
    calendar,
    ql.Actual365Fixed(),
    availability_lag,
    fixing_frequency,
    base_rate,
    helpers,
)
```

```
pd.DataFrame(
    inflation_curve.nodes(), columns=["node", "rate"]
).style.format({"rate": "{:.4%}"})
```

	node	rate
0	April 1st, 2024	2.0985%
1	March 1st, 2025	2.0985%
2	March 1st, 2026	2.5889%
3	March 1st, 2027	2.7210%
4	March 1st, 2028	2.7980%
5	March 1st, 2029	2.8545%
6	March 1st, 2031	2.9586%
7	March 1st, 2034	3.1057%
8	March 1st, 2036	3.1858%
9	March 1st, 2039	3.2467%
10	March 1st, 2044	3.3029%

	node	rate
11	March 1st, 2049	3.3190%
12	March 1st, 2054	3.3235%
13	March 1st, 2064	3.2888%
14	March 1st, 2074	3.2117%

Using the curve, an inflation index can forecast future fixings:

```
hicp = ql.EUHICP(ql.ZeroInflationTermStructureHandle(inflation_curve))
```

```
hicp.fixing(ql.Date(1, ql.February, 2027))
```

```
135.99222083993126
```

Starting with version 1.34, though, it's finally possible to specify the base date explicitly; in order to determine it, we also added a `lastFixingDate` method to inflation indexes for convenience.

```
inflation_curve = ql.PiecewiseZeroInflation(
    today,
    hicp.lastFixingDate(),
    fixing_frequency,
    ql.Actual365Fixed(),
    helpers,
)
```

```
pd.DataFrame(
    inflation_curve.nodes(), columns=["node", "rate"]
).style.format({"rate": "{:.4%}"})
```

	node	rate
0	April 1st, 2024	2.0985%
1	March 1st, 2025	2.0985%
2	March 1st, 2026	2.5889%
3	March 1st, 2027	2.7210%
4	March 1st, 2028	2.7980%
5	March 1st, 2029	2.8545%
6	March 1st, 2031	2.9586%
7	March 1st, 2034	3.1057%
8	March 1st, 2036	3.1858%
9	March 1st, 2039	3.2467%
10	March 1st, 2044	3.3029%

	node	rate
11	March 1st, 2049	3.3190%
12	March 1st, 2054	3.3235%
13	March 1st, 2064	3.2888%
14	March 1st, 2074	3.2117%

As you can see above, the result of the bootstrapping process is the same, and so are the fixings forecast by the index:

```
hicp = ql.EUHICP(ql.ZeroInflationTermStructureHandle(inflation_curve))
```

```
hicp.fixing(ql.Date(1, ql.February, 2027))
```

```
135.99222083993126
```

The old constructors for the various inflation curves available in the library are now deprecated and will be removed in QuantLib 1.39.

Feedback

As I mentioned in the introduction, this is a work in progress. I'll work on adding more chapters in the future. If you want to suggest topics to cover, feel free to contact me; the same goes for corrections and comments on the existing chapters. I can't guarantee I'll follow up on every request, but I'll read them all. Bye for now!