QuantLib-Python Module Reference

David Duarte

Created on : December, 2018

Last updated : September, 2021

More documents are freely available at PythonDSP

Table of contents

Table of contents i

List of figures vii

List of tables ix

1 Basics 3
- 1.1 Settings 3
- 1.2 Array 3
- 1.3 Matrix 4
- 1.4 Observable 4
- 1.5 Quotes 5
  - 1.5.1 SimpleQuote 5
  - 1.5.2 DerivedQuote 5
  - 1.5.3 CompositeQuote 5
  - 1.5.4 DeltaVolQuote 5

2 CashFlows, Legs and Interest Rates 7
- 2.1 Interest Rates 7
- 2.2 CashFlows 8
  - 2.2.1 SimpleCashFlow 8
  - 2.2.2 Redemption 8
  - 2.2.3 AmortizingPayment 8
- 2.3 Coupons 8
  - 2.3.1 FixedRateCoupon 8
  - 2.3.2 IborCoupon 9
  - 2.3.3 OvernightIndexedCoupon 9
  - 2.3.4 CappedFlooredCoupon 9
  - 2.3.5 CappedFlooredIborCoupon 9
  - 2.3.6 CmsCoupon 9
  - 2.3.7 CappedFlooredCmsCoupon 10
  - 2.3.8 CmsSpreadCoupon 10
  - 2.3.9 CappedFlooredCmsSpreadCoupon 10
- 2.4 Legs 11
  - 2.4.1 Leg 11
  - 2.4.2 FixedRateLeg 11
  - 2.4.3 IborLeg 11
  - 2.4.4 OvernightLeg 12
- 2.5 Pricers 12
  - 2.5.1 BlackIborCouponPricer 12
  - 2.5.2 LinearTsrPricer 12
  - 2.5.3 LognormalCmsSpreadPricer 13
  - 2.5.4 NumericHaganPricer 13

2.5.5 AnalyticHaganPricer . . . 13
2.6 Cashflow Analysis Functions . . . 13
2.6.1 Date Inspectors . . . 13
2.6.2 Cashflow Inspectors . . . 13
2.6.3 YieldTermstructure . . . 13
2.6.4 Yield (a.k.a. Internal Rate of Return, i.e. IRR) . . . 14
2.6.5 Z-spread . . . 15

3 Currencies **17**
3.1 Currency . . . 17
3.2 Money . . . 17
3.3 ExhangeRate . . . 18
3.4 ExchangeRateManager . . . 18

4 Dates and Conventions **19**
4.1 Conventions . . . 19
4.1.1 Compounding . . . 19
4.1.2 Frequencies . . . 19
4.1.3 Weekday correction . . . 19
4.1.4 DateGeneration . . . 20
4.2 Date . . . 20
4.3 Period . . . 21
4.4 Calendar . . . 22
4.5 DayCounter . . . 24
4.6 Schedule . . . 25
4.7 MakeSchedule . . . 26
4.8 TimeGrid . . . 26

5 Indexes **27**
5.1 Interest Rate . . . 27
5.1.1 IborIndex . . . 27
5.1.2 OvernightIndex . . . 27
5.1.3 SwapIndex . . . 28
5.1.4 SwapSpreadIndex . . . 28
5.2 Inflation . . . 28
5.2.1 Zero Inflation . . . 28
5.2.2 YoY inflation . . . 28
5.3 Fixings . . . 29
5.4 IndexManager . . . 29

6 Instruments **31**
6.1 Fixed Income . . . 31
6.1.1 Forwards . . . 31
6.1.2 Bonds . . . 32
6.1.3 Swaps . . . 35
6.1.4 Swaptions . . . 39
6.1.5 Caps & Floors . . . 39
6.2 Inflation . . . 40
6.2.1 CPI Bond . . . 40
6.2.2 CPISwap . . . 41
6.2.3 ZeroCouponInflationSwap . . . 41
6.2.4 YearOnYearInflationSwap . . . 42
6.2.5 YoYInflationCap . . . 42
6.2.6 YoYInflationFloor . . . 42
6.2.7 YoYInflationCollar . . . 42
6.3 Credit . . . 42
6.3.1 CreditDefaultSwap . . . 42
6.3.2 CdsOption . . . 42
6.4 Options . . . 43

6.4.1 Vanilla Options . . . 43
6.4.2 Asian Options . . . 43
6.4.3 Barrier Options . . . 44
6.4.4 Basket Options . . . 45
6.4.5 Cliquet Options . . . 45
6.4.6 Forward Options . . . 45
6.4.7 Quanto Options . . . 46

7 Math Tools **47**
7.1 Solvers . . . 47
7.2 Integration . . . 48
7.2.1 Gaussian Quadrature . . . 48
7.3 Interpolation . . . 49
7.3.1 1D interpolation method . . . 49
7.3.2 2D Interpolation Methods . . . 50
7.4 Optimization . . . 50
7.5 Random Number Generators . . . 50
7.5.1 HaltonRsg . . . 52
7.5.2 SobolRsg . . . 52
7.6 Path Generators . . . 52
7.6.1 GaussianMultiPathGenerator . . . 52
7.6.2 GaussianSobolMultiPathGenerator . . . 53
7.7 Statistics . . . 54
7.8 Convention Calculators . . . 54
7.8.1 BlackDeltaCalculator . . . 54

8 Pricing Engines **55**
8.1 Bond Pricing Engines . . . 55
8.1.1 DiscountingBondEngine . . . 55
8.1.2 BlackCallableFixedRateBondEngine . . . 55
8.1.3 TreeCallableFixedRateEngine . . . 55
8.2 Cap Pricing Engines . . . 56
8.2.1 BlackCapFloorEngine . . . 56
8.2.2 BachelierCapFloorEngine . . . 56
8.2.3 AnalyticCapFloorEngine . . . 56
8.2.4 TreeCapFloorEngine . . . 57
8.3 Swap Pricing Engines . . . 57
8.3.1 DiscountingSwapEngine . . . 57
8.4 Swaption Pricing Engines . . . 58
8.4.1 BlackSwaptionEngine . . . 58
8.4.2 BachelierSwaptionEngine . . . 58
8.4.3 FdHullWhiteSwaptionEngine . . . 58
8.4.4 FdG2SwaptionEngine . . . 58
8.4.5 G2SwaptionEngine . . . 58
8.4.6 JamshidianSwaptionEngine . . . 59
8.4.7 TreeSwaptionEngine . . . 59
8.5 Credit Pricing Engines . . . 59
8.5.1 IsdaCdsEngine . . . 59
8.5.2 MidPointCdsEngine . . . 59
8.5.3 IntegralCdsEngine . . . 60
8.5.4 BlackCdsOptionEngine . . . 60
8.6 Option Pricing Engines . . . 60
8.6.1 Vanilla Options . . . 60
8.6.2 Asian Options . . . 64
8.6.3 Barrier Options . . . 67
8.6.4 Basket Options . . . 69
8.6.5 Cliquet Options . . . 70
8.6.6 Forward Options . . . 70

8.6.7 Quanto Options . . . 71

9 Pricing Models **73**
9.1 Equity . . . 73
9.1.1 Heston . . . 73
9.1.2 Bates . . . 74
9.2 Short Rate Models . . . 74
9.2.1 One Factor Models . . . 74
9.2.2 Two Factor Models . . . 75

10 Stochastic Processes **77**
10.1 GeometricBrownianMotionProcess . . . 77
10.2 BlackScholesProcess . . . 77
10.3 BlackScholesMertonProcess . . . 77
10.4 GeneralizedBlackScholesProcess . . . 78
10.5 ExtendedOrnsteinUhlenbeckProcess . . . 78
10.6 ExtOUWithJumpsProcess . . . 78
10.7 BlackProcess . . . 78
10.8 Merton76Process . . . 79
10.9 VarianceGammaProcess . . . 79
10.10GarmanKohlagenProcess . . . 79
10.11HestonProcess . . . 79
10.12HestonSLVProcess . . . 80
10.13BatesProcess . . . 81
10.14HullWhiteProcess . . . 81
10.15HullWhiteForwardProcess . . . 81
10.16GSR Process . . . 81
10.17G2Process . . . 81
10.18G2ForwardProcess . . . 81
10.19Multiple Processes . . . 81

11 Term Structures **83**
11.1 Yield Term Structures . . . 83
11.1.1 FlatForward . . . 83
11.1.2 DiscountCurve . . . 83
11.1.3 ZeroCurve . . . 84
11.1.4 ForwardCurve . . . 84
11.1.5 Piecewise . . . 84
11.1.6 ImpliedTermStructure . . . 86
11.1.7 ForwardSpreadedTermStructure . . . 86
11.1.8 ZeroSpreadedTermStructure . . . 86
11.1.9 SpreadedLinearZeroInterpolatedTermStructure . . . 87
11.1.10 FittedBondCurve . . . 87
11.1.11 FXImpliedCurve . . . 88
11.2 Volatility . . . 88
11.2.1 BlackConstantVol . . . 88
11.2.2 BlackVarianceCurve . . . 89
11.2.3 BlackVarianceSurface . . . 89
11.2.4 HestonBlackVolSurface . . . 90
11.2.5 AndreasenHugeVolatilityAdapter . . . 90
11.2.6 BlackVolTermStructureHandle . . . 91
11.2.7 RelinkableBlackVolTermStructureHandle . . . 91
11.2.8 LocalConstantVol . . . 91
11.2.9 LocalVolSurface . . . 91
11.2.10 NoExceptLocalVolSurface . . . 92
11.2.11 AndreasenHugeLocalVolAdapter . . . 92
11.2.12 LocalVolTermStructureHandle . . . 93
11.3 Cap Volatility . . . 93
11.3.1 ConstantOptionletVolatility . . . 93

11.3.2 CapFloorTermVolCurve . . . 93
11.3.3 CapFloorTermVolSurface . . . 94
11.3.4 OptionletStripper1 . . . 95
11.3.5 StrippedOptionletAdapter . . . 95
11.3.6 OptionletVolatilityStructureHandle . . . 95
11.3.7 RelinkableOptionletVolatilityStructureHandle . . . 95
11.4 Swaption Volatility . . . 95
11.4.1 ConstantSwaptionVolatility . . . 95
11.4.2 SwaptionVolatilityMatrix . . . 96
11.4.3 SwaptionVolCube1 . . . 97
11.4.4 SwaptionVolCube2 . . . 97
11.4.5 SwaptionVolatilityStructureHandle . . . 98
11.4.6 RelinkableSwaptionVolatilityStructureHandle . . . 98
11.5 SABR . . . 98
11.5.1 SabrSmileSection . . . 98
11.5.2 sabrVolatility . . . 98
11.5.3 shiftedSabrVolatility . . . 98
11.5.4 sabrFlochKennedyVolatility . . . 99
11.6 Credit Term Structures . . . 99
11.6.1 FlatHazardRate . . . 99
11.6.2 PiecewiseFlatHazardRate . . . 99
11.6.3 SurvivalProbabilityCurve . . . 100
11.7 Inflation Term Structures . . . 100
11.7.1 ZeroInflationCurve . . . 100
11.7.2 YoYInflationCurve . . . 100
11.7.3 PiecewiseZeroInflation . . . 100

12 Helpers **101**
12.1 Interest Rate . . . 101
12.1.1 DepositRateHelper . . . 101
12.1.2 FraRateHelper . . . 101
12.1.3 Futures . . . 103
12.1.4 SwapRateHelper . . . 107
12.1.5 OISRateHelper . . . 108
12.1.6 DatedOISRateHelper . . . 108
12.1.7 FxSwapRateHelper . . . 108
12.1.8 CrossCurrencyBasisSwapRateHelper . . . 108
12.1.9 FixedRateBondHelper . . . 109
12.1.10 BondHelper . . . 109
12.1.11 BondHelperVector . . . 109
12.1.12 RateHelperVector . . . 110
12.2 Volatility . . . 110
12.2.1 CapHelper . . . 110
12.2.2 SwaptionHelper . . . 110
12.2.3 HestonModelHelper . . . 111
12.3 Credit . . . 112
12.3.1 SpreadCdsHelper . . . 112
12.4 Inflation . . . 112

13 Fixed Income **113**
13.1 Gearing in swaps . . . 113
13.2 Vanilla Swap . . . 115
13.3 Yield Curve . . . 116

14 Inflation **119**

15 Credit **121**

16 Equity **123**

17 Indices and tables **125**

Index **127**

Index **127**

List of figures

List of tables

Contents:

Chapter 1

Basics

1.1 Settings

```
today = ql.Date(15,6,2020)
ql.Settings.instance().evaluationDate = today
ql.Settings.instance().setEvaluationDate(today)
```

Moves date of referenced curves:

```
crv = ql.FlatForward(2, ql.TARGET(), 0.05, ql.Actual360())
crv.referenceDate()
```

Changes evaluation date of calculation:

```
today = ql.Date(15,6,2020)
ql.Settings.instance().evaluationDate = today
schedule = ql.MakeSchedule(ql.Date(15,6,2020), ql.Date(15,6,2021), ql.Period('6M'))
leg = ql.FixedRateLeg(schedule, ql.Actual360(), [100.], [0.05])
rate = ql.InterestRate(.03, ql.Thirty360(), ql.Compounded, ql.Annual)
print( ql.CashFlows.npv(leg, rate, False) )
```

4.958531764309427

```
ql.Settings.instance().evaluationDate = ql.Date(15,12,2020)
print( ql.CashFlows.npv(leg, rate, False) )
```

2.4906934531375144

1.2 Array

creates an empty array

`ql.Array()`

creates the array and fills it with value

`ql.Array(`*size*, *value*`)`

creates the array and fills it according to a0=value,ai=ai1+increment

`ql.Array(`*size*, *value*, *increment*`)`

1.3 Matrix

creates a null matrix

`ql.Matrix()`

creates a matrix with the given dimensions

`ql.Matrix(`*rows*, *columns*`)`

creates the matrix and fills it with value

`ql.Matrix(`*rows*, *columns*, *value*`)`

```
ql.Matrix()
ql.Matrix(2,2)
ql.Matrix(2,2,0.5)
```

```
A = ql.Matrix(3,3)
A[0][0]=0.2
A[0][1]=8.4
A[0][2]=1.5
A[1][0]=0.6
A[1][1]=1.4
A[1][2]=7.3
A[2][0]=0.8
A[2][1]=4.4
A[2][2]=3.2
```

1.4 Observable

```
import QuantLib as ql

flag = None
def raiseFlag():
    global flag
    flag = 1

me = ql.SimpleQuote(0.0)
obs = ql.Observer(raiseFlag)
obs.registerWith(me)
me.setValue(3.14)
if not flag:
    print("Case 1: Observer was not notified of market element change")
flag = None
obs.unregisterWith(me)
me.setValue(3.14)
if not flag:
    print("Case 2: Observer was not notified of market element change")
```

1.5 Quotes

1.5.1 SimpleQuote

`ql.SimpleQuote(`*value*`)`

```
s = ql.SimpleQuote(0.01)
```

Functions

- value
- setValue
- isValid

```
s.value()
s.setValue(0.05)
s.isValid()
```

1.5.2 DerivedQuote

`ql.DerivedQuote(`*quoteHandle, function*`)`

```
d1 = ql.SimpleQuote(0.06)
d2 = ql.DerivedQuote(ql.QuoteHandle(d1),lambda x: 10*x)
```

1.5.3 CompositeQuote

`ql.CompositeQuote(quoteHandle, quoteHandle, function)`

```
c1 = ql.SimpleQuote(0.02)
c2 = ql.SimpleQuote(0.03)

def f(x,y):
    return x+y

c3 = ql.CompositeQuote(ql.QuoteHandle(c1),ql.QuoteHandle(c2), f)
c3.value()

c4 = ql.CompositeQuote(ql.QuoteHandle(c1),ql.QuoteHandle(c2), lambda x,y:x+y)
c4.value()
```

1.5.4 DeltaVolQuote

A class for FX-style quotes where delta-maturity pairs are quoted in implied vol

`ql.DeltaVolQuote(`*delta, volQuoteHandle, maturity, deltaType*`)`

`ql.DeltaVolQuote(`*volQuoteHandle, deltaType, maturity, atmType*`)`

```
deltaType = ql.DeltaVolQuote.Fwd     # Also supports: Spot, PaSpot, PaFwd
atmType = ql.DeltaVolQuote.AtmFwd    # Also supports: AtmSpot, AtmDeltaNeutral, AtmVegaMax, AtmGammaMax, AtmPutCall50

maturity = 1.0
volAtm, vol25DeltaCall, vol25DeltaPut = 0.08, 0.075, 0.095

atmDeltaQuote = ql.DeltaVolQuote(ql.QuoteHandle(ql.SimpleQuote(volAtm)), deltaType, maturity, atmType)
```

(continues on next page)

(continued from previous page)

```
vol25DeltaPutQuote = ql.DeltaVolQuote(-0.25, ql.QuoteHandle(ql.SimpleQuote(vol25DeltaPut)), maturity,␣
↪deltaType)
vol25DeltaCallQuote = ql.DeltaVolQuote(0.25, ql.QuoteHandle(ql.SimpleQuote(vol25DeltaCall)), maturity,␣
↪deltaType)
```

Chapter 2

CashFlows, Legs and Interest Rates

2.1 Interest Rates

Concrete interest rate class

`ql.InterestRate(`*rate*, *dayCount*, *compounding*, *frequency*`)`

```
rate = ql.InterestRate(0.05, ql.Actual360(), ql.Compounded, ql.Annual)
```

Here are some common member functions:

- **rate()** : a floating point number that returns the value of the rate of return;
- **dayCounter()** : DayCounter object, which returns the member variable that controls the day calculation rule;
- **compounding()** : an integer that returns the interest rate method;
- **frequency()** : Integer, returns the frequency of interest payments;
- **discountFactor(d1, d2)** : float, d1 and d2 are both Date objects ($d1 < d2$), returning the discount factor size from d1 to d2;
- **compoundFactor(d1, d2)** : float, d1 and d2 are both Date objects ($d1 < d2$), returning the size of the interest factor from d1 to d2;
- **equivalentRate(resultDC, comp, freq, d1, d2)** : The InterestRate object returns an InterestRate object equivalent to the current object. The configuration parameters of the object include resultDC , comp , freq :

- Both d1 and d2 are Date objects ($d1 < d2$)
- **resultDC** : DayCounter object, configure the number of days calculation rules;
- **comp** : integer, configuration interest rate, the value range is some reserved variables of quantlib-python;
- **freq** : integer, configuration payoff frequency, the range of values is some reserved variables of quantlib-python.

In some cases, it is necessary to recalculate the rate of return based on the size of the interest factor. The InterestRate class provides the function impliedRate implement this function:

- **impliedRate(compound, resultDC, comp, freq, d1, d2)** : The InterestRate object returns the inverse calculated InterestRate object whose configuration parameters include resultDC , comp , freq :

- Both d1 and d2 are Date objects ($d1 < d2$)
- **resultDC** : DayCounter object, configure the number of days calculation rules;
- **comp** : integer, configuration interest rate, the value range is some reserved variables of quantlib-python;
- **freq** : integer, configuration payoff frequency, the range of values is some reserved variables of quantlib-python.

```
print("Rate: ", rate.rate())
print("DayCount: ", rate.dayCounter())
print("DiscountFactor: ", rate.discountFactor(1))
print("DiscountFactor: ", rate.discountFactor(ql.Date(15,6,2020), ql.Date(15,6,2021)))
print("CompoundFactor: ", rate.compoundFactor(ql.Date(15,6,2020), ql.Date(15,6,2021)))
print("EquivalentRate: ", rate.equivalentRate(ql.Actual360(), ql.Compounded, ql.Semiannual, ql.Date(15,6,
↪2020), ql.Date(15,6,2021)))

factor = rate.compoundFactor(ql.Date(15,6,2020), ql.Date(15,6,2021))
print("ImpliedRate: ", rate.impliedRate(factor, ql.Actual360(), ql.Continuous, ql.Annual, ql.Date(15,6,
↪2020), ql.Date(15,6,2021)))
```

2.2 CashFlows

2.2.1 SimpleCashFlow

`ql.SimpleCashFlow`(*amount, date*)

```
amount = 105
date = ql.Date(15,6,2020)
cf = ql.SimpleCashFlow(amount, date)
```

2.2.2 Redemption

`ql.Redemption`(*amount, date*)

```
amount = 100
date = ql.Date(15,6,2020)
redemption = ql.Redemption(amount, date)
```

2.2.3 AmortizingPayment

`ql.AmortizingPayment`(*amount, date*)

```
amount = 100
date = ql.Date(15,6,2020)
ql.AmortizingPayment(amount, date)
```

2.3 Coupons

2.3.1 FixedRateCoupon

`ql.FixedRateCoupon`(*paymentDate, nominal, rate, dayCounter, startDate, endDate*)

```
amount = 105
nominal = 100.
paymentDate = ql.Date(15,6,2020)
startDate = ql.Date(15,12,2019)
rate = .05
dayCounter = ql.Actual360()
coupon = ql.FixedRateCoupon(endDate, nominal, rate, dayCounter, startDate, endDate)
```

2.3.2 IborCoupon

ql.IborCoupon(*paymentDate*, *nominal*, *startDate*, *endDate*, *fixingDays*, *index*)

```
nominal = 100.
startDate = ql.Date(15,12,2020)
endDate = ql.Date(15,6,2021)
rate = .05
dayCounter = ql.Actual360()
index = ql.Euribor6M()
coupon = ql.IborCoupon(endDate, nominal, startDate, endDate, 2, index)
```

2.3.3 OvernightIndexedCoupon

ql.OvernightIndexedCoupon(*paymentDate*, *nominal*, *startDate*, *endDate*, *overnightIndex*, *gearing=1.0*, *spread=0.0*, *refPeriodStart=ql.Date()*, *refPeriodEnd=ql.Date()*, *dayCounter=ql.DayCounter()*, *telescopicValueDates=False*)

```
paymentDate = ql.Date(15, 9, 2020)
nominal = 100
startDate = ql.Date(15, 6, 2002)
endDate = ql.Date(15,9,2020)
overnightIndex = ql.Eonia()
ql.OvernightIndexedCoupon(paymentDate, nominal, startDate, endDate, overnightIndex)
```

2.3.4 CappedFlooredCoupon

Capped and/or floored floating-rate coupon

ql.CappedFlooredCoupon(*FloatingRateCoupon*, *cap*, *floor*)

2.3.5 CappedFlooredIborCoupon

2.3.6 CmsCoupon

ql.CmsCoupon(*paymentDate*, *nominal*, *startDate*, *endDate*, *fixingDays*, *swapIndex*)

```
nominal = 100.
startDate = ql.Date(15,12,2020)
endDate = ql.Date(15,6,2021)
rate = .05
dayCounter = ql.Actual360()
index = ql.Euribor6M()
fixingDays = 2
swapIndex = ql.EuriborSwapIsdaFixA(ql.Period("2Y"))
cms = ql.CmsCoupon(endDate, nominal, startDate, endDate, fixingDays, swapIndex)
```

2.3.7 CappedFlooredCmsCoupon

ql.CappedFlooredCmsCoupon(*paymentDate, nominal, startDate, endDate, fixingDays, swapIndex, rate, spread*)

2.3.8 CmsSpreadCoupon

ql.CmsSpreadCoupon(*paymentDate, nominal, startDate, endDate, fixingDays, spreadIndex*)

ql.CmsSpreadCoupon(*paymentDate, nominal, startDate, endDate, fixingDays, spreadIndex, gearing=1, spread=0, refPeriodStart=ql.Date(), refPeriodEnd=ql.Date(), dayCounter=ql.DayCounter(), isInArrears=False, exCouponDate=ql.Date()*)

```
nominal = 100.
startDate = ql.Date(15,12,2020)
endDate = ql.Date(15,6,2021)
fixingDays = 2
swapIndex1 = ql.EuriborSwapIsdaFixA(ql.Period("10Y"))
swapIndex2 = ql.EuriborSwapIsdaFixA(ql.Period("2Y"))
spreadIndex = ql.SwapSpreadIndex("CMS 10Y-2Y", swapIndex1, swapIndex2)
ql.CmsSpreadCoupon(endDate, nominal, startDate, endDate, fixingDays, spreadIndex)
```

2.3.9 CappedFlooredCmsSpreadCoupon

ql.CmsSpreadCoupon(*paymentDate, nominal, startDate, endDate, fixingDays, spreadIndex, gearing=1, spread=0, cap=Null, floor=Null, refPeriodStart=ql.Date(), refPeriodEnd=ql.Date(), dayCounter=ql.DayCounter(), isInArrears=False, exCouponDate=ql.Date()*)

```
nominal = 100.
startDate = ql.Date(15,12,2020)
endDate = ql.Date(15,6,2021)
fixingDays = 2
swapIndex1 = ql.EuriborSwapIsdaFixA(ql.Period("10Y"))
swapIndex2 = ql.EuriborSwapIsdaFixA(ql.Period("2Y"))
spreadIndex = ql.SwapSpreadIndex("CMS 10Y-2Y", swapIndex1, swapIndex2)
ql.CappedFlooredCmsSpreadCoupon(endDate, nominal, startDate, endDate, fixingDays, spreadIndex)

gearing = 1
spread = 0
cap=0
floor=0

ql.CappedFlooredCmsSpreadCoupon(endDate, nominal, startDate, endDate, fixingDays, spreadIndex, gearing,␣
↪spread, cap, floor)

refPeriodStart = ql.Date()
refPeriodEnd = ql.Date()
dayCounter = ql.Actual360()
isInArrears = False
exCouponDate = ql.Date()
ql.CappedFlooredCmsSpreadCoupon(endDate, nominal, startDate, endDate, fixingDays, spreadIndex, gearing,␣
↪spread, cap, floor, refPeriodStart, refPeriodEnd, dayCounter, isInArrears, exCouponDate)
```

2.4 Legs

2.4.1 Leg

```
date = ql.Date().todaysDate()
cf1 = ql.SimpleCashFlow(5.0, date+365)
cf2 = ql.SimpleCashFlow(5.0, date+365*2)
cf3 = ql.SimpleCashFlow(105.0, date+365*3)
leg = ql.Leg([cf1, cf2, cf3])
```

2.4.2 FixedRateLeg

helper class building a sequence of fixed rate coupons

`ql.FixedRateLeg`(*schedule*, *dayCount*, *nominals*, *fixedRate*, *BusinessDayConvention*, *FirstPeriodDayCounter*, *ExCouponPeriod*, *PaymentCalendar*)

```
schedule = ql.MakeSchedule(ql.Date(15,6,2020), ql.Date(15,6,2021), ql.Period('6M'))
dayCount = ql.Actual360()
leg = ql.FixedRateLeg(schedule, dayCount, [100.], [0.05])
leg = ql.FixedRateLeg(schedule, ql.Actual360(), [100.], [0.05], ql.Following, ql.Actual360(), ql.Period(
↪'3M'), ql.TARGET())
```

2.4.3 IborLeg

helper class building a sequence of capped/floored ibor-rate coupon

`ql.IborLeg`(*nominals*, *schedule*, *index*, *paymentDayCounter = DayCounter()*, *paymentConvention = Following*, *fixingDays = 0*, *gearings = 1*, *spreads*, *caps*, *floors*, *isInArrears*, *exCouponPeriod*, *exCouponCalendar*, *exCouponConvention = Unadjusted*, *exCouponEndOfMonth = False*)

```
schedule = ql.MakeSchedule(ql.Date(15,6,2020), ql.Date(15,6,2021), ql.Period('6M'))
index = ql.Euribor3M()
leg = ql.IborLeg([100], schedule, index)
```

```
leg = ql.IborLeg([100], schedule, index, ql.Actual360())
leg = ql.IborLeg([100], schedule, index, ql.Actual360(), ql.ModifiedFollowing)
leg = ql.IborLeg([100], schedule, index, ql.Actual360(), ql.ModifiedFollowing, [2])
leg = ql.IborLeg([100], schedule, index, ql.Actual360(), ql.ModifiedFollowing, fixingDays=[2],␣
↪gearings=[1])

leg = ql.IborLeg([100], schedule, index, ql.Actual360(), ql.ModifiedFollowing, fixingDays=[2],␣
↪gearings=[1], spreads=[0])
leg = ql.IborLeg([100], schedule, index, ql.Actual360(), ql.ModifiedFollowing, fixingDays=[2],␣
↪gearings=[1], spreads=[0], caps=[0])
leg = ql.IborLeg([100], schedule, index, ql.Actual360(), ql.ModifiedFollowing, fixingDays=[2],␣
↪gearings=[1], spreads=[0], floors=[0])
```

2.4.4 OvernightLeg

helper class building a sequence of overnight coupons

`ql.OvernightLeg(`*nominals, schedule, overnightIndex, dayCount, BusinessDayConvention, gearing, spread, TelescopicValueDates*`)`

```
nominal = 100
schedule = ql.MakeSchedule(ql.Date(15,6,2020), ql.Date(15,6,2021), ql.Period('3M'))
overnightIndex = ql.OvernightIndex('CNYRepo7D', 1, ql.CNYCurrency(), ql.China(), ql.Actual365Fixed())
ql.OvernightLeg([nominal], schedule, overnightIndex, ql.Actual360(), ql.Following, [1],[0], True)
```

2.5 Pricers

2.5.1 BlackIborCouponPricer

`ql.BlackIborCouponPricer(`*OptionletVolatilityStructureHandle*`)`

```
volatility = 0.10;
vol = ql.ConstantOptionletVolatility(2, ql.TARGET(), ql.Following, volatility, ql.Actual360())
pricer = ql.BlackIborCouponPricer(ql.OptionletVolatilityStructureHandle(vol))
```

Example: In arrears coupon

```
crv = ql.FlatForward(0, ql.TARGET(), -0.01, ql.Actual360())
yts = ql.YieldTermStructureHandle(crv)
index = ql.Euribor3M(yts)

schedule = ql.MakeSchedule(ql.Date(15,6,2021), ql.Date(15,6,2023), ql.Period('6M'))

leg = ql.IborLeg([100], schedule, index, ql.Actual360(), ql.ModifiedFollowing, isInArrears=True)

volatility = 0.10;
vol = ql.ConstantOptionletVolatility(2, ql.TARGET(), ql.Following, volatility, ql.Actual360())
pricer = ql.BlackIborCouponPricer(ql.OptionletVolatilityStructureHandle(vol))
ql.setCouponPricer(leg, pricer)

npv = ql.CashFlows.npv(leg, yts, True)
print(f"LEG NPV: {npv:,.2f}")
```

2.5.2 LinearTsrPricer

`ql.LinearTsrPricer(`*swaptionVolatilityStructure, meanReversion*`)`

```
volQuote = ql.QuoteHandle(ql.SimpleQuote(0.2))
swaptionVol = ql.ConstantSwaptionVolatility(0, ql.TARGET(), ql.ModifiedFollowing, volQuote, ql.
↪Actual365Fixed())
swvol_handle = ql.SwaptionVolatilityStructureHandle(swaptionVol)

mean_reversion = ql.QuoteHandle(ql.SimpleQuote(0.01))
cms_pricer = ql.LinearTsrPricer(swvol_handle, mean_reversion)
```

2.5.3 LognormalCmsSpreadPricer

2.5.4 NumericHaganPricer

2.5.5 AnalyticHaganPricer

2.6 Cashflow Analysis Functions

2.6.1 Date Inspectors

`ql.CashFlows.startDate(`*leg*`)`

`ql.CashFlows.maturityDate(`*leg*`)`

2.6.2 Cashflow Inspectors

the last cashflow paying before or at the given date

`ql.CashFlows.previousCashFlowDate(`*leg, includeSettlementDateFlows, settlementDate=ql.Date()*`)`

```
ql.CashFlows.previousCashFlowDate(leg, True)
ql.CashFlows.previousCashFlowDate(leg, True, ql.Date(15,12,2020))
```

the first cashflow paying after the given date

`ql.CashFlows.nextCashFlowDate(`*leg, includeSettlementDateFlows, settlementDate=ql.Date()*`)`

2.6.3 YieldTermstructure

NPV of the cash flows

`ql.CashFlows.npv(`*leg, discountCurve, includeSettlementDateFlows, settlementDate=ql.Date(), npvDate=ql.Date()*`)`

```
yts = ql.YieldTermStructureHandle(ql.FlatForward(ql.Date(15,1,2020), 0.04, ql.Actual360()))
ql.CashFlows.npv(leg, yts, True)
ql.CashFlows.npv(leg, yts, True, ql.Date(15,6,2020))
ql.CashFlows.npv(leg, yts, True, ql.Date(15,6,2020), ql.Date(15,12,2020))
```

Basis-point sensitivity of the cash flows

`ql.CashFlows.bps(`*leg, discountCurve, includeSettlementDateFlows, settlementDate=ql.Date(), npvDate=ql.Date()*`)`

```
yts = ql.YieldTermStructureHandle(ql.FlatForward(ql.Date(15,1,2020), 0.04, ql.Actual360()))
ql.CashFlows.bps(leg, yts, True)
```

At-the-money rate of the cash flows

`ql.CashFlows.atmRate(`*leg, discountCurve, includeSettlementDateFlows, settlementDate=ql.Date(), ql.npvDate=Date(), npv=Null< Real >()*`)`

```
crv = ql.FlatForward(ql.Date(15,1,2020), 0.04, ql.Actual360())
ql.CashFlows.atmRate(leg, crv, True, ql.Date(15,6,2020))
```

2.6.4 Yield (a.k.a. Internal Rate of Return, i.e. IRR)

`ql.CashFlows.npv(`*leg, rate, includeSettlementDateFlows, settlementDate=ql.Date(), npvDate=ql.Date()*`)`

```
rate = ql.InterestRate(.03, ql.ActualActual(), ql.Compounded, ql.Annual)
ql.CashFlows.npv(leg, rate, True)
```

`ql.CashFlows.bps(`*leg, rate, includeSettlementDateFlows, settlementDate=ql.Date(), npvDate=ql.Date()*`)`

```
rate = ql.InterestRate(.03, ql.ActualActual(), ql.Compounded, ql.Annual)
ql.CashFlows.bps(leg, rate, True)
```

`ql.CashFlows.basisPointValue(`*leg, InterestRate, includeSettlementDateFlows, settlementDate=ql.Date(), ql.npvDate=Date()*`)`

```
rate = ql.InterestRate(.03, ql.ActualActual(), ql.Compounded, ql.Annual)
ql.CashFlows.basisPointValue(leg, rate, True)
```

`ql.CashFlows.basisPointValue(`*leg, rate, dayCounter, compounding, frequency, includeSettlementDateFlows, settlementDate=ql.Date(), ql.npvDate=Date()*`)`

```
ql.CashFlows.basisPointValue(leg, 0.05, ql.Actual360(), ql.Compounded, ql.Annual, True)
```

`ql.CashFlows.duration(`*leg, InterestRate, ql.Duration.Type, includeSettlementDateFlows, settlementDate=ql.Date(), npvDate=ql.Date()*`)`

```
rate = ql.InterestRate(.03, ql.ActualActual(), ql.Compounded, ql.Annual)

ql.CashFlows.duration(leg, rate, ql.Duration.Simple, False)
ql.CashFlows.duration(leg, rate, ql.Duration.Macaulay, False)
ql.CashFlows.duration(leg, rate, ql.Duration.Modified, False)
```

`ql.CashFlows.duration(`*leg, rate, dayCounter, compounding, frequency, ql.Duration.Type, includeSettlementDateFlows, settlementDate=ql.Date(), npvDate=ql.Date()*`)`

```
rate = 0.05
ql.CashFlows.duration(leg, rate, ql.Actual360(), ql.Compounded, ql.Annual, ql.Duration.Simple, False)
```

`ql.CashFlows.convexity(`*leg, InterestRate, includeSettlementDateFlows, settlementDate=ql.Date(), npvDate=ql.Date()*`)`

```
rate = ql.InterestRate(.03, ql.ActualActual(), ql.Compounded, ql.Annual)
ql.CashFlows.convexity(leg, rate, False)
```

`ql.CashFlows.convexity(`*leg, rate, dayCounter, compounding, frequency, includeSettlementDateFlows, settlementDate=ql.Date(), npvDate=ql.Date()*`)`

```
rate = 0.05
ql.CashFlows.convexity(leg, rate, ql.Actual360(), ql.Compounded, ql.Annual, False)
```

`ql.CashFlows.yieldRate(`*leg, rate, dayCounter, compounding, frequency, includeSettlementDateFlows, settlementDate=ql.Date(), npvDate=ql.Date(), accuracy=1e-10, maxIterations=100, guess=0.0*`)`

```
ql.CashFlows.yieldRate(leg, 5, ql.Actual360(), ql.Compounded, ql.Annual, True)
ql.CashFlows.yieldRate(leg, 5, ql.Actual360(), ql.Compounded, ql.Annual, True, ql.Date(15,6,2020))
ql.CashFlows.yieldRate(leg, 5, ql.Actual360(), ql.Compounded, ql.Annual, True, ql.Date(15,6,2020), ql.
↪Date(15,12,2020))
ql.CashFlows.yieldRate(leg, 5, ql.Actual360(), ql.Compounded, ql.Annual, True, ql.Date(15,6,2020), ql.
↪Date(15,12,2020), 1e-5)
ql.CashFlows.yieldRate(leg, 5, ql.Actual360(), ql.Compounded, ql.Annual, True, ql.Date(15,6,2020), ql.
↪Date(15,12,2020), 1e-5, 100)
ql.CashFlows.yieldRate(leg, 5, ql.Actual360(), ql.Compounded, ql.Annual, True, ql.Date(15,6,2020), ql.
↪Date(15,12,2020), 1e-5, 100, 0.04)
```

2.6.5 Z-spread

implied Z-spread.

`ql.CashFlows.zSpread(`*leg*, *npv*, *YieldTermStructure*, *dayCounter*, *compounding*, *frequency*, *includeSettlementDateFlows*, *settlementDate=ql.Date()*, *npvDate=ql.Date()*, *accuracy=1e-10*, *maxIterations=100*, *guess=0.0*`)`

```
crv = ql.FlatForward(ql.Date(15,1,2020), 0.04, ql.Actual360())
ql.CashFlows.zSpread(leg, 5.5, crv, ql.Actual360(), ql.Compounded, ql.Annual, True)
```

Chapter 3

Currencies

3.1 Currency

The common member functions are as follows:

- name() : returns a string, the name of the currency;
- code() : Returns a string, the ISO4217 code of the currency, usually three uppercase English letters;
- numericCode() : returns an integer, the number corresponding to the ISO4217 code of the currency;
- symbol() : Returns a string, which is a symbol often used to represent the currency in the real world. The dollar is "$" and the yen is "". It should be noted that this function may return Unicode, which may cause the program to fail in Python;
- fractionSymbol() : Returns a string, which is a symbol often used to represent the smallest unit of the currency in the real world. Like symbol() , this function may return Unicode, which may cause the program to fail in Python;
- fractionsPerUnit() : Returns an integer, the multiple of a unit of currency relative to the smallest unit of the currency, usually 100.
- format() : Returns a string, a "format string" used to format the print result.
- empty() : Returns a Boolean value. If the object is instantiated by a derived class, it returns True ; if the object is instantiated by Currency , it returns False . After all, the currency information in the base class object is "empty."
- rounding() : Returns a Rounding object, which is the rounding rule of the currency. By default, no rounding is performed.

3.2 Money

To make algebraic calculations of currencies in QuantLib, you need to transform the Currency object into a Money object.

`ql.Money(`*currency*, *value*`)`

`ql.Money(`*value*, *currency*`)`

```
cur = ql.EURCurrency()
money1 = ql.Money(cur, 100)
money2 = ql.Money(100, cur)
```

Money is not usually constructed explicitly, but is generated by multiplying a floating-point number with a Currency object:

```
money3 = 100 * cur
money3.value()
```

3.3 ExhangeRate

The class in QuantLib that describes exchange rate information between currencies is ExchangeRate , and every two currencies in the Currency system can generate an ExchangeRate object.

ExchangeRate constructor is very fixed and accepts three parameters: ExchangeRate(source, target, rate)

source a Currency object representing the source currency
target a Currency object representing the target currency
rate A floating point number that represents the exchange rate for “source to target”

Common member functions are as follows:

source() returns the Currency object, which is the source currency;
target() returns the Currency object, which is the target currency;
rate() returns a floating point number, ie the exchange rate;
type() returns the built-in integer constant,
ExchangeRate.Direct equal to 0, indicating that the exchange rate is constructed directly through the constructor;
ExchangeRate.Derived equal to 1, indicating that the exchange rate is constructed indirectly through other exchange rate objects;
exchange(amount) amount is a Money object that converts amount into other currencies of equal value;
chain(r1, r2) r1 and r2 are ExchangeRate objects. The currency involved must form a triangular relationship. This function will return an ExchangeRate object to complete the missing side of the triangle relationship.

```
usd = ql.USDCurrency()
eur = ql.EURCurrency()
usdToeur = ql.ExchangeRate(eur, usd, 1.14)
m_usd = 5 * usd
m_eur = 4.39 * eur
print( 'Converting from USD: ', m_usd, ' = ', usdToeur.exchange(m_usd))
print( 'Converting from EUR: ', m_eur, ' = ', usdToeur.exchange(m_eur))

print(usdToeur.source())
print(usdToeur.target())
print(usdToeur.rate())
```

3.4 ExchangeRateManager

```
ql.ExchangeRateManager.instance()
```

Chapter 4

Dates and Conventions

4.1 Conventions

4.1.1 Compounding

- *ql.Simple*
- *ql.Compounded*
- *ql.Continuous*
- *ql.SimpleThenCompounded*
- *ql.CompoundedThenSimple*

4.1.2 Frequencies

- *ql.NoFrequency* : no interest;
- *ql.Once* : pay interest once, common in zero-coupon bonds;
- *ql.Annual* : paying interest once a year;
- *ql.Semiannual* : Semiannual interest semi-annually;
- *ql.EveryFourthMonth* : every 4 months;
- *ql.Quarterly* : Quarterly quarterly;
- *ql.Bimonthly* : paying interest every two months;
- *ql.Monthly* : monthly interest payment;
- *ql.EveryFourthWeek* : every 4 weeks;
- *ql.Biweekly* : Biweekly interest every two weeks;
- *ql.Weekly* : paying once a week;
- *ql.Daily* : pay interest once a day.

4.1.3 Weekday correction

- *ql.Following* : The date is corrected to the first working day that follows.
- *ql.ModifiedFollowing* : The date is corrected to the first working day after that, unless this working day is in the next month; if the modified working day is in the next month, the date is corrected to the last working day that appears before, to ensure the original The date and the revised date are in the same month.
- *ql.Preceding* : Correct the date to the last business day that Preceding before.
- *ql.ModifiedPreceding* : modify the date to the last working day that appeared before, unless the working sunrise is now the previous month; if the modified working sunrise is now the previous month, the date is modified to the first working day after that The original date and the revised date are guaranteed to be in the same month.
- *ql.Unadjusted* : No adjustment.

4.1.4 DateGeneration

The valuation of many products relies on an analysis of future cash flows, so accurately generating a list of dates for future cash flows is crucial. After the start and end dates are given, the date list can be generated in the manner of "reverse method" or "forward method".

Example:

Monthly periods with start date is 07-05-2020 and the end date is 15-08-2020:

```
start = ql.Date(7,5,2020)
end = ql.Date(15,8,2020)

rules = {
    'Backward': ql.DateGeneration.Backward,
    'Forward': ql.DateGeneration.Forward,
    'Zero': ql.DateGeneration.Zero,
    'ThirdWednesDay': ql.DateGeneration.ThirdWednesday,
    'Twentieth': ql.DateGeneration.Twentieth,
    'TwentiethIMM': ql.DateGeneration.TwentiethIMM,
    'CDS': ql.DateGeneration.CDS

}

for name, rule in rules.items():
    schedule = ql.MakeSchedule(start, end, ql.Period('1m'), rule=rule)
    print(name, [dt for dt in schedule])
```

4.2 Date

Constructors

`ql.Date(`*serialNumber*`)`

serialNumber is an integer, such as 24214, and 1 corresponds to 1899-12-31. The usage is the same as in Excel. (The range of serialNumber is limited to 367 to 109574, and the corresponding date range is 1901-01-01 to 2199-12-31.)

```
ql.Date(44000)
```

`ql.Date(`*day, month, year*`)`

where day and year are integers; month can be an integer or a special object reserved in quantlib-python that is specifically used to represent the month (ql.January (equal to 1), . . . , ql.December(equal to 12))

```
ql.Date(18, 6, 2020)
ql.Date(18, ql.June, 2020)
```

`ql.Date(`*dateString, formatString*`)`

```
ql.Date('18-06-2020', '%d-%m-%Y')
```

Member functions

- **ISO**
- **weekday()** : an integer that returns the number corresponding to the week: - Sunday: 1 - . . . - Saturday: 7
- **dayOfMonth()** : integer, the date returned is the day of the month
- **dayOfYear()** : integer, the date returned is the day of the year
- **month()** : an integer that returns the month corresponding to the date
- **year()** : an integer that returns the year corresponding to the date

- **serialNumber()** integer, the number of days corresponding to the return date (starting from 1899-12-31)

```
print('Original Date:', today)
print('ISO format:', today.ISO())
print('Weekday:', today.weekday())
print('Day of Month:', today.dayOfMonth())
print('Day of Year:', today.dayOfYear())
print('Month:', today.month())
print('Year:', today.year())
print('Serial Number:', today.serialNumber())
```

Static functions

- **Date.todaysDate()** : Date object, which returns the current date of the system.
- **Date.minDate()** : Date object, which returns the minimum date that QuantLib can represent.
- **Date.maxDate()** : Date object, which returns the maximum date that QuantLib can represent.
- **Date.isLeap(y)** : Boolean value to determine whether y is a leap year
- **Date.endOfMonth(d)** : Date object, which returns the date corresponding to the end of the month where the date d is located
- **Date.isEndOfMonth(d)** : Boolean value to determine whether d is at the end of the month
- **Date.nextWeekday(d, w)** : Date object, which returns the date corresponding to the first week w after date d (for example, the first Friday after 2018-03-12)
- **Date.nthWeekday(n, w, m, y)** : Date object, which returns the date corresponding to the n week w in the given month m and year y (for example, the third Wednesday of July 2010)

```
print('Today :', ql.Date.todaysDate())
print('Min Date :', ql.Date.minDate())
print('Max Date :', ql.Date.maxDate())
print('Is Leap :', ql.Date.isLeap(2011))
print('End of Month :', ql.Date.endOfMonth(ql.Date(4, ql.August, 2009)))
print('Is Month End :', ql.Date.isEndOfMonth(ql.Date(29, ql.September, 2009)))
print('Is Month End :', ql.Date.isEndOfMonth(ql.Date(30, ql.September, 2009)))
print('Next WD :', ql.Date.nextWeekday(ql.Date(1, ql.September, 2009), ql.Friday))
print('n-th WD :', ql.Date.nthWeekday(3, ql.Wednesday, ql.September, 2009))
```

4.3 Period

`ql.Period(`*n*, *units*`)`

```
ql.Period(1, ql.Days)
```

`ql.Period(`*periodString*`)`

```
ql.Period('1d')
```

`ql.Period(`*frequency*`)`

```
ql.Period(ql.Annual)
```

4.4 Calendar

The class *ql.Calendar* provides the interface for determining whether a date is a business day or a holiday for a given exchange or a given country, and for incrementing/decrementing a date of a given number of business days.

Available Calendars

Argentina, Australia, Austria, BespokeCalendar, Botswana, Brazil, Canada, China, CzechRepublic, Denmark, Finland, France, Germany, HongKong, Hungary, Iceland, India, Indonesia, Israel, Italy, Japan, JointCalendar, Mexico, NewZealand, Norway, NullCalendar, Poland, Romania, Russia, SaudiArabia, Singapore, Slovakia, SouthAfrica, SouthKorea, Sweden, Switzerland, Taiwan, TARGET, Thailand, Turkey, Ukraine, UnitedKingdom, UnitedStates, WeekendsOnly

```
calendar1 = ql.UnitedKingdom()
calendar2 = ql.TARGET()
```

Calendar Markets

Argentina : [‘Merval’]
Brazil : [‘Exchange’, ‘Settlement’]
Canada : [‘Settlement’, ‘TSX’]
China : [‘IB’, ‘SSE’]
CzechRepublic : [‘PSE’]
France : [‘Exchange’, ‘Settlement’]
Germany : [‘Eurex’, ‘FrankfurtStockExchange’, ‘Settlement’, ‘Xetra’]
HongKong : [‘HKEx’]
Iceland : [‘ICEX’]
India : [‘NSE’]
Indonesia : [‘BEJ’, ‘JSX’]
Israel : [‘Settlement’, ‘TASE’]
Italy : [‘Exchange’, ‘Settlement’]
Mexico : [‘BMV’]
Russia : [‘MOEX’, ‘Settlement’]
SaudiArabia : [‘Tadawul’]
Singapore : [‘SGX’]
Slovakia : [‘BSSE’]
SouthKorea : [‘KRX’, ‘Settlement’]
Taiwan : [‘TSEC’]
Ukraine : [‘USE’]
UnitedKingdom : [‘Exchange’, ‘Metals’, ‘Settlement’]
UnitedStates : [‘FederalReserve’, ‘GovernmentBond’, ‘LiborImpact’, ‘NERC’, ‘NYSE’, ‘Settlement’]

```
calendar1 = ql.UnitedKingdom(ql.UnitedKingdom.Metals)
calendar2 = ql.UnitedStates(ql.UnitedStates.NYSE)
```

Some commonly used member functions:

- **isBusinessDay(d)** : A Boolean value that determines whether d is a business day.
- **isHoliday(d)** : A boolean value that determines whether d is a holiday.
- **isWeekend(w)** : A Boolean value that determines whether w is a weekend (in some countries, weekends are not scheduled on Saturdays and Sundays).
- **isEndOfMonth(d)** : A boolean value that determines whether d is the last working day at the end of the month.
- **endOfMonth(d)** : date, returns the last working day of the month in which d is located.

```
cal = ql.TARGET()
mydate = ql.Date(1, ql.May, 2017)

print('Is BD :', cal.isBusinessDay(mydate))
print('Is Holiday :', cal.isHoliday(mydate))
print('Is Weekend :', cal.isWeekend(ql.Friday))
print('Is Last BD :', cal.isEndOfMonth(ql.Date(5, ql.April, 2018)))
print('Last BD :', cal.endOfMonth(mydate))
```

Custom Holiday List

The Calendar object in QuantLib can conveniently implement custom holidays. Generally, only the following two functions are needed:

- **addHoliday(d)** : add d as a holiday.
- **removeHoliday(d)** : remove d from the holiday table.

```
cal = ql.TARGET()

day1 = ql.Date(26, 2, 2020)
day2 = ql.Date(10, 4, 2020)

print('Is Business Day : ', cal.isBusinessDay(day1))
print('Is Business Day : ', cal.isBusinessDay(day2))

cal.addHoliday(day1)
cal.removeHoliday(day2)

print('Is Business Day : ', cal.isBusinessDay(day1))
print('Is Business Day : ', cal.isBusinessDay(day2))
```

```
myCalendar = ql.WeekendsOnly()
days = [1,14,15,1,21,26,2,16,15,18,19,9,27,1,19,8,17,25,31]
months = [1,4,4,5,5,6,8,9,9,10,10,11,12,12,12,12]
name = ['Año Nuevo','Viernes Santo','Sabado Santo','Dia del Trabajo','Dia de las Glorias Navales','San␣
↪Pedro y San Pablo','Elecciones Primarias','Dia de la Virgen del Carmen','Asuncion de la Virgen',
↪'Independencia Nacional','Glorias del Ejercito','Encuentro de dos mundos','Día de las Iglesias␣
↪Evangélicas y Protestantes','Día de todos los Santos','Elecciones Presidenciales y Parlamentarias',
↪'Inmaculada Concepción','Segunda vuelta Presidenciales','Navidad','Feriado Bancario']
start_year = 2018
n_years = 10
for i in range(n_years+1):
    for x,y in zip(days,months):
        date = ql.Date(x,y,start_year+i)
        myCalendar.addHoliday(date)
```

Holiday List

Returns the holidays between two dates.

`ql.Calendar.holidayList(`*calendar*, *from*, *to*, *includeWeekEnds=False*`)`

```
ql.Calendar.holidayList(ql.TARGET(), ql.Date(1,12,2019), ql.Date(31,12,2019))
```

Calendar object uses the following two functions to modify the date:

- **adjust(d, convention)** : Date, modify d according to the convention conversion mode.
- **advance(d, period, convention, endOfMonth)** : date, the date is moved backward by time interval period and then modified according to the conversion mode convention ; the parameter endOfMonth indicates that if d is the end of the month, the date after the correction is also at the end of the month.

Finally, the following function can be used to calculate the number of working days during the two days:

- **businessDaysBetween(from, to, includeFirst, includeLast)** : Calculate the number of working days between the dates from and to (whether or not the dates are included).

```
cal = ql.TARGET()

firstDate = ql.Date(31, ql.January, 2018)
secondDate = ql.Date(1, ql.April, 2018)

print('Date 2 Adj :', cal.adjust(secondDate, ql.Preceding))
print('Date 2 Adj :', cal.adjust(secondDate, ql.ModifiedPreceding))

mat = ql.Period(2, ql.Months)

print('Date 1 Month Adv :',
    cal.advance(firstDate, mat, ql.Following, False))
print('Date 1 Month Adv :',
    cal.advance(firstDate, mat, ql.ModifiedFollowing, False))

print('Business Days Between :',
    cal.businessDaysBetween(
        ql.Date(5, ql.March, 2018), ql.Date(30, ql.March, 2018),
        True, True))
```

JointCalenar

`ql.JointCalendar(`*calendar1, calendar2, calendar3, calendar4, JointCalendarRule=JoinHolidays*`)`

```
joint_calendar = ql.JointCalendar(ql.TARGET(), ql.Poland())
```

4.5 DayCounter

https://www.isda.org/a/pIJEE/The-Actual-Actual-Day-Count-Fraction-1999.pdf

The "Day Count Convention" is critical for the valuation of financial products, especially for fixed-income products. QuantLib provides the following common rules:

- **Actual360** : Actual / 360
- **Actual365Fixed** : Actual / 365(Fixed)

  - Standard
  - Canadian
  - NoLeap

- **ActualActual** : Actual / Actual

  - ISMA
  - Bond
  - ISDA
  - Historical
  - Actual365
  - AFB
  - Euro

- **Business252** : Business / 252
- **Thirty360** : 30 / 360
- **SimpleDayCounter**

```
dayCounters = {
    'SimpleDayCounter': ql.SimpleDayCounter(),
```

(continues on next page)

(continued from previous page)

```
    'Thirty360': ql.Thirty360(),
    'Actual360': ql.Actual360(),
    'Actual365Fixed': ql.Actual365Fixed(),
    'Actual365Fixed(Canadian)': ql.Actual365Fixed(ql.Actual365Fixed.Canadian),
    'Actual365FixedNoLeap': ql.Actual365Fixed(ql.Actual365Fixed.NoLeap),
    'ActualActual': ql.ActualActual(),
    'Business252': ql.Business252()
}


startDate = ql.Date(15,5,2015)
endDate = ql.Date(15,6,2015)
r = 0.05
nominal = 100e6

for name, dc in dayCounters.items():
    amount = ql.FixedRateCoupon(endDate, nominal, r, dc, startDate, endDate).amount()
    print(name, f"{amount:,.2f}")
```

4.6 Schedule

`Schedule(`*effectiveDate, terminationDate, tenor, calendar, convention, terminationDateConvention, rule, endOfMonth, firstDate=Date(), nextToLastDate=Date()*`)`

The types and explanations of these variables are as follows:

- **effectiveDate, terminationDate** : Date, the start and end date of the calendar list, such as the value date and expiration date of the bond.
- **tenor** : Period object, the interval between two adjacent dates, such as the bond frequency (1 year or 6 months) or interest rate swap rate (3 months).
- **calendar** : A calendar table that generates a specific calendar of dates to follow.
- **convention** : integer, how to adjust the non-working day (except the last date), the value range is some reserved variables of quantlib-python.
- **terminationDateConvention** : Integer, if the last date is a non-working day, how to adjust it, the value range is some reserved variables of quantlib-python.
- **Rule** : A member of DateGeneration that generates the rules for the date.
- **endOfMonth** : If the start date is at the end of the month, whether other dates are required to be scheduled at the end of the month (except the last date).
- **firstDate** : nextToLastDate (optional): Date, the start and end date (not commonly used) provided for the generated method rule .

```
effectiveDate = ql.Date(15,6,2020)
terminationDate = ql.Date(15,6,2022)
frequency = ql.Period('6M')
calendar = ql.TARGET()
convention = ql.ModifiedFollowing
terminationDateConvention = ql.ModifiedFollowing
rule = ql.DateGeneration.Backward
endOfMonth = False
schedule = ql.Schedule(effectiveDate, terminationDate, frequency, calendar, convention,␣
↪terminationDateConvention, rule, endOfMonth)
```

4.7 MakeSchedule

`ql.MakeSchedule(`*effectiveDate, terminationDate, frequency*`)`

Optional params:

- calendar=None
- convention=None
- terminalDateConvention=None,
- rule=None
- forwards=False
- backwards=False,
- endOfMonth=None
- firstDate=None
- nextToLastDate=None

```
effectiveDate = ql.Date(15,6,2020)
terminationDate = ql.Date(15,6,2022)
frequency = ql.Period('6M')
schedule = ql.MakeSchedule(effectiveDate, terminationDate, frequency)
```

4.8 TimeGrid

`ql.TimeGrid(`*end, steps*`)`

```
t = ql.TimeGrid(10, 5)
t.dt(4)
```

If there are certain times that need to appear in the TimeGrid, pass them in as a list

`ql.TimeGrid(`*requiredTimes, steps*`)`

```
[t for t in ql.TimeGrid([1, 2.5, 4], 10)]
```

Chapter 5

Indexes

5.1 Interest Rate

5.1.1 IborIndex

`ql.IborIndex(`*familyName, tenor, settlementDays, currency, fixingCalendar, convention, endOfMonth, dayCounter, =Handleql.YieldTermStructure()*`)`

```
ql.IborIndex('MyIndex', ql.Period('6m'), 2, ql.EURCurrency(), ql.TARGET(), ql.ModifiedFollowing, True, 
↪ql.Actual360())
ql.Libor('MyIndex', ql.Period('6M'), 2, ql.USDCurrency(), ql.TARGET(), ql.Actual360())
ql.Euribor(ql.Period('6M'))
ql.USDLibor(ql.Period('6M'))
ql.Euribor6M()
```

Derived Classes:

- ql.Euribor()

Constructors for derived classes:

`ql.Euribor(`*period*`)`

`ql.Euribor(`*period, yts*`)`

5.1.2 OvernightIndex

`ql.OvernightIndex(`*name, fixingDays, currency, calendar, dayCounter, =ql.YieldTermStructureHandle()*`)`

```
name = 'CNYRepo7D'
fixingDays = 1
currency = ql.CNYCurrency()
calendar = ql.China()
dayCounter = ql.Actual365Fixed()
overnight_index = ql.OvernightIndex(name, fixingDays, currency, calendar, dayCounter)
```

5.1.3 SwapIndex

`ql.SwapIndex`(*familyName*, *tenor*, *settlementDays*, *currency*, *fixingCalendar*, *fixedLegTenor*, *convention*, *dayCounter*, *index*, *=Handleql.YieldTermStructure()*)

Derived Classes:

- ql.ChfLiborSwapIsdaFix
- ql.EuriborSwapIsdaFixA
- ql.EuriborSwapIsdaFixB
- ql.EuriborSwapIfrFix
- ql.EurLiborSwapIfrFix
- ql.EurLiborSwapIsdaFixA
- ql.EurLiborSwapIsdaFixB
- ql.GbpLiborSwapIsdaFix
- ql.JpyLiborSwapIsdaFixAm
- ql.JpyLiborSwapIsdaFixPm
- ql.OvernightIndexedSwapIndex
- ql.UsdLiborSwapIsdaFixAm
- ql.UsdLiborSwapIsdaFixPm

Constructors for derived classes:

`ql.EuriborSwapIsdaFixA`(*period*)

`ql.EuriborSwapIsdaFixA`(*period*, *yts*)

`ql.EuriborSwapIsdaFixA`(*period*, *forward_yts*, *discounting_yts*)

5.1.4 SwapSpreadIndex

`SwapSpreadIndex`(*familyName*, *swapIndex1*, *swapIndex2*, *gearing1=1.0*, *gearing2=- 1.0*)

5.2 Inflation

5.2.1 Zero Inflation

`ql.{InflationIndex}(interpolated=bool)`

`ql.{InflationIndex}(bool, ZeroInflationTermStructure)`

- ql.UKRPI
- ql.USCPI
- ql.EUHICP
- ql.EUHICPXT

5.2.2 YoY inflation

- ql.YYEUHICP
- ql.YYEUHICPXT
- ql.YYFRHICP
- ql.YYUKRPI
- ql.YYUSCPI
- ql.YYZACPI

5.3 Fixings

```
fixingDates = [cf.fixingDate() for cf in map(ql.as_floating_rate_coupon, loan)]
euribor3m.clearFixings()

euribor3m.addFixing(ql.Date(17, 7, 2018), -0.3)
euribor3m.addFixings([ql.Date(12, 7, 2018), ql.Date(13, 7, 2018)], [-0.3, -0.3])
```

```
[dt for dt in index.timeSeries().dates()]
[dt for dt in index.timeSeries().values()]
```

To get the fixing dates form an instrument:

```
swap3 = ql.MakeVanillaSwap(ql.Period('3y'), ql.Euribor6M(), 0.01, ql.Period("-2D"))
fixingDates = [cf.fixingDate() for cf in map(ql.as_floating_rate_coupon, swap3.floatingLeg())]
```

Indexes have calendars and will not accept invalid fixing dates:

```
index.isValidFixingDate(ql.Date(25,12,2019))
c = index.fixingCalendar()
c.name()
```

5.4 IndexManager

```
ql.IndexManager.instance().histories()

for dt, value in zip(im.getHistory('EURIBOR6M ACTUAL/360').dates(),im.getHistory('EURIBOR6M ACTUAL/360').
↪values()):
    print(dt, value)


IndexManager.instance().clearHistory(index.name())
```

Chapter 6

Instruments

6.1 Fixed Income

6.1.1 Forwards

6.1.1.1 Forward Rate Agreement

class ql.ForwardRateAgreement(*valueDate*, *maturityDate*, *position*, *strikeForward*, *notional*, *iborIndex*, *discountCurve=ql.YieldTermStructureHandle()*)

```
fra = ql.ForwardRateAgreement(
    ql.Date(15,6,2020),
    ql.Date(15,12,2020),
    ql.Position.Long,
    0.01,
    1e6,
    ql.Euribor6M(yts),
    yts
)
```

.NPV()

.businessDayConvention()

.calendar()

.dayCounter()

.discountCurve()

.fixingDate()

.forwardRate()

.forwardValue()

.impliedYield(*underlyingSpotValue*, *forwardValue*, *settlementDate*, *compoundingConvention*, *dayCounter*)

.incomeDiscountCurve()

.isExpired()

.settlementDate()

.spotIncome(*yts*)

.spotValue()

6.1.1.2 FixedRateBondForward

class ql.FixedRateBondForward(*valueDate*, *maturityDate*, *Position::Type*, *strike*, *settlementDays*, *dayCounter*, *calendar*, *businessDayConvention*, *FixedRateBond*, *yieldTermStructure=ql.YieldTermStructureHandle()*, *incomeDiscountCurve=ql.YieldTermStructureHandle()*)

Position:

- *ql.Position.Long*
- *ql.Position.Short*

```
valueDate = ql.Date(24, 6, 2020)
maturityDate = ql.Date(31, 5, 2032)
position = ql.Position.Long
strike = 100
settlementDays = 2
dayCounter = ql.Actual360()
calendar = ql.TARGET()
businessDayConvention = ql.Following
bond = ql.FixedRateBond(2, ql.TARGET(), 100.0, ql.Date(31, 5, 2032), ql.Date(30, 5, 2035), ql.
↪Period('1Y'), [0.05], ql.ActualActual())
bond.setPricingEngine(engine)
fwd = ql.FixedRateBondForward(
    valueDate, maturityDate, position, strike, settlementDays,
    dayCounter , calendar, businessDayConvention, bond, yts, yts)
```

6.1.2 Bonds

6.1.2.1 Bond

Redemptions and maturity are calculated from the coupon data, if available. Therefore, redemptions must not be included in the passed cash flows.

class ql.Bond(*settlementDays*, *calendar*, *issueDate*, *coupons*)

```
start = ql.Date(15,12,2019)
maturity = ql.Date(15,12,2020)
schedule = ql.MakeSchedule(start, maturity, ql.Period('6M'))

interest = ql.FixedRateLeg(schedule, ql.Actual360(), [100.], [0.05])
bond = ql.Bond(0, ql.TARGET(), start, interest)
```

.bondYield(*dayCounter*, *compounding*, *frequency*, *accuracy=1e-08*, *maxEvaluations=100*)

.bondYield(*cleanPrice*, *dayCounter*, *compounding*, *frequency*, *settlementDate=Date*, *accuracy=1e-08*, *maxEvaluations=100*)

```
bond.bondYield(100, ql.Actual360(), ql.Compounded, ql.Annual)
```

.dirtyPrice()

```
bond.dirtyPrice()
```

.dirtyPrice(*yield*, *dayCount*, *compounding*, *frequency*)

```
bond.dirtyPrice(0.05, ql.Actual360(), ql.Compounded, ql.Annual)
```

6.1.2.2 ZeroCouponBond

ql.ZeroCouponBond(*settlementDays*, *calendar*, *faceAmount*, *maturityDate*)

```
bond = ql.ZeroCouponBond(2, ql.TARGET(), 100, ql.Date(20,6,2020))
```

6.1.2.3 FixedRateBond

ql.FixedRateBond(*settlementDays*, *calendar*, *faceAmount*, *startDate*, *maturityDate*, *tenor*, *coupon*, *paymentConvention*)

ql.FixedRateBond(*settlementDays*, *faceAmount*, *schedule*, *coupon*, *paymentConvention*)

```
bond = ql.FixedRateBond(2, ql.TARGET(), 100.0, ql.Date(15,12,2019), ql.Date(15,12,2024), ql.Period('1Y'),
↪ [0.05], ql.ActualActual())
```

6.1.2.4 AmortizingFixedRateBond

ql.AmortizingFixedRateBond(*settlementDays*, *notionals*, *schedule*, *coupons*, *accrualDayCounter*, *paymentConvention=Following*, *issueDate=Date()*)

```
notionals = [100,100,100,50]
schedule = ql.MakeSchedule(ql.Date(25,1,2018), ql.Date(25,1,2022), ql.Period('1y'))
bond = ql.AmortizingFixedRateBond(0, notionals, schedule, [0.03], ql.Thirty360())
```

6.1.2.5 FloatingRateBond

ql.FloatingRateBond(*settlementDays*, *faceAmount*, *schedule*, *index*, *dayCounter*, *paymentConvention*)

```
schedule = ql.MakeSchedule(ql.Date(15,6,2020), ql.Date(15,6,2022), ql.Period('6m'))
index = ql.Euribor6M()
ql.FloatingRateBond(2,100, schedule, index, ql.Actual360(), spreads=[0.01])
```

6.1.2.6 AmortizingFloatingRateBond

ql.FloatingRateBond(*settlementDays*, *notionals*, *schedule*, *index*, *dayCounter*)

```
notional = [100, 50]
schedule = ql.MakeSchedule(ql.Date(15,6,2020), ql.Date(15,6,2022), ql.Period('1Y'))
index = ql.Euribor6M()
ql.AmortizingFloatingRateBond(2, notional, schedule, index, ql.ActualActual())
```

6.1.2.7 CMS Rate Bond

ql.CmsRateBond(*settlementDays*, *faceAmount*, *schedule*, *index*, *dayCounter*, *paymentConvention*, *fixingDays*, *gearings*, *spreads*, *caps*, *floors*)

```
schedule = ql.MakeSchedule(ql.Date(15,6,2020), ql.Date(15,6,2022), ql.Period('1Y'))
index = ql.EuriborSwapIsdaFixA(ql.Period('10y'))
ql.CmsRateBond(2, 100, schedule, index, ql.Actual360(), ql.ModifiedFollowing, fixingDays=2, gearings=[1],
↪ spreads=[0], caps=[], floors=[])
```

6.1.2.8 Callable Bond

ql.CallableFixedRateBond(*settlementDays*, *faceAmount*, *schedule*, *coupons*, *accrualDayCounter*, *paymentConvention*, *redemption*, *issueDate*, *putCallSchedule*)

```
schedule = ql.MakeSchedule(ql.Date(15,6,2020), ql.Date(15,6,2022), ql.Period('1Y'))
putCallSchedule = ql.CallabilitySchedule()

callability_price  = ql.CallabilityPrice(100, ql.CallabilityPrice.Clean)

putCallSchedule.append(
    ql.Callability(callability_price, ql.Callability.Call, ql.Date(15,6,2021))
)

ql.CallableFixedRateBond(2, 100, schedule, [0.01], ql.Actual360(), ql.ModifiedFollowing, 100, ql.Date(15,
↪6,2020), putCallSchedule)
```

6.1.2.9 Convertible Bond

6.1.2.10 BondFunctions

```
bond = ql.FixedRateBond(
    2, ql.TARGET(), 100.0,
    ql.Date(15,12,2019), ql.Date(15,12,2024), ql.Period('1Y'),
    [0.05], ql.ActualActual())
```

Date Inspectors

```
ql.BondFunctions.startDate(bond)
ql.BondFunctions.maturityDate(bond)
ql.BondFunctions.isTradable(bond)
```

Cashflow Inspectors

```
ql.BondFunctions.previousCashFlowDate(bond)
ql.BondFunctions.previousCashFlowDate(bond, ql.Date(15,12,2020))
ql.BondFunctions.previousCashFlowAmount(bond)
ql.BondFunctions.previousCashFlowAmount(bond, ql.Date(15,12,2020))
ql.BondFunctions.nextCashFlowDate(bond)
ql.BondFunctions.nextCashFlowDate(bond, ql.Date(15,12,2020))
ql.BondFunctions.nextCashFlowAmount(bond)
ql.BondFunctions.nextCashFlowAmount(bond, ql.Date(15,12,2020))
```

Coupon Inspectors

```
ql.BondFunctions.previousCouponRate(bond)
ql.BondFunctions.nextCouponRate(bond)
ql.BondFunctions.accrualStartDate(bond)
ql.BondFunctions.accrualEndDate(bond)
```

(continues on next page)

(continued from previous page)

```
ql.BondFunctions.accrualPeriod(bond)
ql.BondFunctions.accrualDays(bond)
ql.BondFunctions.accruedPeriod(bond)
ql.BondFunctions.accruedDays(bond)
ql.BondFunctions.accruedAmount(bond)
```

YieldTermStructure

```
crv = ql.FlatForward(2, ql.TARGET(), 0.04, ql.Actual360())
ql.BondFunctions.cleanPrice(bond, crv)
ql.BondFunctions.bps(bond, crv)
ql.BondFunctions.atmRate(bond, crv)
```

Yield (a.k.a. Internal Rate of Return, i.e. IRR) functions

```
rate = ql.InterestRate(0.05, ql.Actual360(), ql.Compounded, ql.Annual)
ql.BondFunctions.cleanPrice(bond, rate)
ql.BondFunctions.bps(bond, rate)
ql.BondFunctions.duration(bond, rate)
ql.BondFunctions.convexity(bond, rate)
ql.BondFunctions.basisPointValue(bond, rate)
ql.BondFunctions.yieldValueBasisPoint(bond, rate)
```

Z-spread functions

```
crv = ql.FlatForward(2, ql.TARGET(), 0.04, ql.Actual360())
ql.BondFunctions.zSpread(bond, 101, crv, ql.Actual360(), ql.Compounded, ql.Annual)
```

6.1.3 Swaps

6.1.3.1 VanillaSwap

`ql.VanillaSwap`(*type*, *nominal*, *fixedSchedule*, *fixedRate*, *fixedDayCount*, *floatSchedule*, *index*, *spread*, *floatingDayCount*)

Types:

- *ql.VanillaSwap.Payer*
- *ql.VanillaSwap.Receiver*

```
calendar = ql.TARGET()
start = ql.Date(17,6,2019)
maturity = calendar.advance(start, ql.Period('5y'))

fixedSchedule = ql.MakeSchedule(start, maturity, ql.Period('1Y'))

floatSchedule = ql.MakeSchedule(start, maturity, ql.Period('6M'))

swap = ql.VanillaSwap(
    ql.VanillaSwap.Payer, 100,
    fixedSchedule, 0.01, ql.Thirty360(),
    floatSchedule, ql.Euribor6M(), 0, ql.Actual360()
)
```

6.1.3.2 Swap

ql.Swap(*firstLeg*, *secondLeg*)

```
fixedSchedule = ql.MakeSchedule(start, maturity, ql.Period('1Y'))
fixedLeg = ql.FixedRateLeg(fixedSchedule, ql.Actual360(), [100], [0.01])

floatSchedule = ql.MakeSchedule(start, maturity, ql.Period('6M'))
floatLeg = ql.IborLeg([100], floatSchedule, ql.Euribor6M(), ql.Actual360())

swap = ql.Swap(fixedLeg, floatLeg)
```

6.1.3.3 MakeVanillaSwap

ql.MakeVanillaSwap(*tenor*, *index*, *fixedRate*, *forwardStart*)

Optional params:

- fixedLegDayCount
- Nominal
- receiveFixed,
- swapType
- settlementDays
- effectiveDate
- terminationDate
- dateGenerationRule
- fixedLegTenor
- fixedLegCalendar
- fixedLegConvention
- fixedLegDayCount
- floatingLegTenor
- floatingLegCalendar
- floatingLegConvention
- floatingLegDayCount
- floatingLegSpread
- discountingTermStructure
- pricingEngine
- fixedLegTerminationDateConvention
- fixedLegDateGenRule
- fixedLegEndOfMonth
- fixedLegFirstDate
- fixedLegNextToLastDate,
- floatingLegTerminationDateConvention
- floatingLegDateGenRule
- floatingLegEndOfMonth
- floatingLegFirstDate
- floatingLegNextToLastDate

```
tenor = ql.Period('5y')
index = ql.Euribor6M()
fixedRate = 0.05
forwardStart = ql.Period("2D")

swap = ql.MakeVanillaSwap(tenor, index, fixedRate, forwardStart, Nominal=100)
swap = ql.MakeVanillaSwap(tenor, index, fixedRate, forwardStart, swapType=ql.VanillaSwap.Payer)
```

6.1.3.4 Amortizing Swap

```
calendar = ql.TARGET()
start = ql.Date(17,6,2019)
maturity = calendar.advance(start, ql.Period('2y'))


fixedSchedule = ql.MakeSchedule(start, maturity, ql.Period('1Y'))
fixedLeg = ql.FixedRateLeg(fixedSchedule, ql.Actual360(), [100, 50], [0.01])

floatSchedule = ql.MakeSchedule(start, maturity, ql.Period('6M'))
floatLeg = ql.IborLeg([100, 100, 50, 50], floatSchedule, ql.Euribor6M(), ql.Actual360())

swap = ql.Swap(fixedLeg, floatLeg)
```

6.1.3.5 FloatFloatSwap

```
ql.FloatFloatSwap(ql.VanillaSwap.Payer,
                [notional] * (len(float3m)-1),
                [notional] * (len(float6m)-1),
                float3m,
                index3m,
                ql.Actual360(),
                float6m,
                index6m,
                ql.Actual360(), False, False,
                [1] * (len(float3m)-1),
                [spread] * (len(float3m)-1))
```

6.1.3.6 AssetSwap

`ql.AssetSwap(`*payFixed*, *bond*, *cleanPrice*, *index*, *spread*`)`

`ql.AssetSwap(`*payFixed*, *bond*, *cleanPrice*, *index*, *spread*, *schedule*, *dayCount*, *bool*`)`

```
payFixedRate = True
bond = ql.FixedRateBond(2, ql.TARGET(), 100.0, ql.Date(15,12,2019), ql.Date(15,12,2024),
    ql.Period('1Y'), [0.05], ql.ActualActual()
    )
bondCleanPrice = 100
index = ql.Euribor6M()
spread = 0.0
ql.AssetSwap(payFixedRate, bond, bondCleanPrice, index, spread, ql.Schedule(), ql.ActualActual(), True)
```

6.1.3.7 OvernightIndexedSwap

`ql.OvernightIndexedSwap(`*swapType*, *nominal*, *schedule*, *fixedRate*, *fixedDC*, *overnightIndex*`)`

Or array of nominals

`ql.OvernightIndexedSwap(`*swapType*, *nominals*, *schedule*, *fixedRate*, *fixedDC*, *overnightIndex*`)`

Optional params:

- spread=0.0
- paymentLag=0
- paymentAdjustment=ql.Following()
- paymentCalendar=ql.Calendar()
- telescopicValueDates=false

Types:

- *ql.OvernightIndexedSwap.Receiver*
- *ql.OvernightIndexedSwap.Receiver*

```
swapType = ql.OvernightIndexedSwap.Receiver
nominal = 100
schedule = ql.MakeSchedule(ql.Date(15,6,2020), ql.Date(15,6,2021), ql.Period('1Y'), calendar=ql.TARGET())
fixedRate = 0.01
fixedDC = ql.Actual360()
overnightIndex = ql.Eonia()
ois_swap = ql.OvernightIndexedSwap(swapType, nominal, schedule, fixedRate, fixedDC, overnightIndex)
```

6.1.3.8 MakeOIS

`ql.MakeOIS(`*swapTenor, overnightIndex, fixedRate*`)`

Optional params:

- fwdStart=Period(0, Days)
- receiveFixed=True,
- swapType=OvernightIndexedSwap.Payer
- nominal=1.0
- settlementDays=2
- effectiveDate=None
- terminationDate=None
- dateGenerationRule=DateGeneration.Backward
- paymentFrequency=Annual
- paymentAdjustmentConvention=Following
- paymentLag=0
- paymentCalendar=None
- endOfMonth=True
- fixedLegDayCount=None
- overnightLegSpread=0.0
- discountingTermStructure=None
- telescopicValueDates=False
- pricingEngine=None

```
swapTenor = ql.Period('1Y')
overnightIndex = ql.Eonia()
fixedRate = 0.01
ois_swap = ql.MakeOIS(swapTenor, overnightIndex, fixedRate)
```

6.1.3.9 NonstandardSwap

`ql.NonstandardSwap(`*swapType, fixedNominal, floatingNominal, fixedSchedule, fixedRate, fixedDayCount, floatingSchedule, iborIndex, gearing, spread, floatDayCount*`)`

Optional params:

- intermediateCapitalExchange = False
- finalCapitalExchange = False,
- paymentConvention = None

```
swapType = ql.VanillaSwap.Payer
fixedNominal = [100, 100]
floatingNominal  = [100] * 4
fixedSchedule = ql.MakeSchedule(ql.Date(15,6,2020), ql.Date(15,6,2022), ql.Period('1Y'))
fixedRate = [0.02] * 2
```

(continues on next page)

(continued from previous page)

```
fixedDayCount = ql.Thirty360()
floatingSchedule = ql.MakeSchedule(ql.Date(15,6,2020), ql.Date(15,6,2022), ql.Period('6M'))
iborIndex = ql.Euribor6M()
gearing = [1.] * 4
spread = [0.] * 4
floatDayCount = iborIndex.dayCounter()
nonstandardSwap = ql.NonstandardSwap(
    swapType, fixedNominal, floatingNominal,
    fixedSchedule, fixedRate, fixedDayCount,
    floatingSchedule, iborIndex, gearing, spread, floatDayCount)
```

6.1.4 Swaptions

Exercises

- *ql.EuropeanExercise(start)*
- *ql.AmericanExercise(earliestDate, latestDate)*
- *ql.BermudanExercise(dates)*

Settlement Type/Method

- ***ql.Settlement.Cash***
  - *ql.Settlement.CollateralizedCashPrice*
  - *ql.Settlement.ParYieldCurve*
- ***ql.Settlement.Physical***
  - *ql.Settlement.PhysicalCleared*
  - *ql.Settlement.PhysicalOTC*

6.1.4.1 Swaption

`ql.Swaption`(*swap*, *exercise*, *settlementType=ql.Settlement.Physical*, *settlementMethod=ql.Settlement.PhysicalOTC*)

```
calendar = ql.TARGET()
today = ql.Date().todaysDate()
exerciseDate = calendar.advance(today, ql.Period('5y'))
exercise = ql.EuropeanExercise(exerciseDate)
swap = ql.MakeVanillaSwap(ql.Period('5y'), ql.Euribor6M(), 0.05, ql.Period('5y'))
swaption = ql.Swaption(swap, exercise)

swaption = ql.Swaption(swap, exercise, ql.Settlement.Cash, ql.Settlement.ParYieldCurve)
swaption = ql.Swaption(swap, exercise, ql.Settlement.Physical, ql.Settlement.PhysicalCleared)
```

6.1.4.2 Nonstandard Swaption

6.1.4.3 FloatFloatSwaption

6.1.5 Caps & Floors

6.1.5.1 Cap

`ql.Cap`(*floatingLeg*, *exerciseRates*)

```
schedule = ql.MakeSchedule(ql.Date(15,6,2020), ql.Date(16,6,2022), ql.Period('6M'))
ibor_leg = ql.IborLeg([100], schedule, ql.Euribor6M())
strike = 0.01
cap = ql.Cap(ibor_leg, [strike])
```

6.1.5.2 Floor

ql.Floor(*floatingLeg*, *exerciseRates*)

```
schedule = ql.MakeSchedule(ql.Date(15,6,2020), ql.Date(16,6,2022), ql.Period('6M'))
ibor_leg = ql.IborLeg([100], schedule, ql.Euribor6M())
strike = 0.00
floor = ql.Floor(ibor_leg, [strike])
```

6.1.5.3 Collar

ql.Collar(*floatingLeg*, *capRates*, *floorRates*)

```
schedule = ql.MakeSchedule(ql.Date(15,6,2020), ql.Date(16,6,2022), ql.Period('6M'))
ibor_leg = ql.IborLeg([100], schedule, ql.Euribor6M())
capStrike = 0.02
floorStrike = 0.00
collar = ql.Collar(ibor_leg, [capStrike], [floorStrike])
```

6.2 Inflation

6.2.1 CPI Bond

ql.CPIBond(*settlementDays*, *notional*, *growthOnly*, *baseCPI*, *contractObservationLag*, *inflationIndex*, *observationInterpolation*, *fixedSchedule*, *fixedRates*, *fixedDayCounter*, *fixedPaymentConvention*)

```
calendar = ql.UnitedKingdom()

today = ql.Date(5,3,2008)
evaluationDate = calendar.adjust(today)
issue_date = calendar.advance(evaluationDate,-1, ql.Years)
maturity_date = ql.Date(2,9,2052)

settlementDays = 3
notional = 1000000
growthOnly = False
baseCPI = 206.1
contractObservationLag = ql.Period(3, ql.Months)
inflationIndex = ql.UKRPI(False)
observationInterpolation = ql.CPI.Flat
fixedSchedule = ql.MakeSchedule(issue_date, maturity_date, ql.Period(ql.Semiannual))
fixedRates = [0.1]
fixedDayCounter = ql.Actual365Fixed()
fixedPaymentConvention = ql.ModifiedFollowing

bond = ql.CPIBond(settlementDays,
                    notional,
                    growthOnly,
                    baseCPI,
```

(continues on next page)

(continued from previous page)

```
                    contractObservationLag,
                    inflationIndex,
                    observationInterpolation,
                    fixedSchedule,
                    fixedRates,
                    fixedDayCounter,
                    fixedPaymentConvention)
```

6.2.2 CPISwap

`ql.CPISwap(swapType, nominal, subtractInflationNominal, spread, floatDayCount, schedule, floatPaymentConv`

```
swapType = ql.CPISwap.Payer
nominal = 1e6
subtractInflationNominal = True
spread = 0.0
floatDayCount = ql.Actual365Fixed()
floatPaymentConvention = ql.ModifiedFollowing
fixingDays = 0;
floatIndex = ql.GBPLibor(ql.Period('6M'))
fixedRate = 0.1;
baseCPI = 206.1;
fixedDayCount = ql.Actual365Fixed()
fixedPaymentConvention = ql.ModifiedFollowing;
fixedIndex = ql.UKRPI(False);
contractObservationLag = ql.Period('3M')
observationInterpolation = ql.CPI.Linear
startDate = ql.Date(2,10,2007)
endDate = ql.Date(2,10,2052)
schedule = ql.MakeSchedule(startDate, endDate, ql.Period('6m'))
zisV = ql.CPISwap(
    swapType, nominal, subtractInflationNominal, spread,
    floatDayCount, schedule, floatPaymentConvention, fixingDays, floatIndex,
    fixedRate, baseCPI, fixedDayCount, schedule, fixedPaymentConvention,
    contractObservationLag, fixedIndex, observationInterpolation)
```

6.2.3 ZeroCouponInflationSwap

`ql.ZeroCouponInflationSwap(`*swapType, notional, start, maturity, calendar, BusinessDayConvention, DayCounter, fixedRate, ZeroInflationIndex, observationLag*`)`

```
swapType = ql.ZeroCouponInflationSwap.Payer
calendar = ql.TARGET()
nominal = 1e6
startDate = ql.Date(11,1,2022)
endDate = ql.Date(11,1,2023)
fixedRate = 0.1;
dc = ql.Actual365Fixed()
inflationIndex = ql.EUHICPXT(True)
contractObservationLag = ql.Period(3, ql.Months)
bdc = ql.ModifiedFollowing
swap = ql.ZeroCouponInflationSwap(swapType, nominal, startDate, endDate, calendar, bdc, dc, fixedRate,
inflationIndex, contractObservationLag)
```

6.2.4 YearOnYearInflationSwap

`ql.YearOnYearInflationSwap(`*swapType, nominal, fixedSchedule, fixedRate, fixedDayCounter, yoySchedule, index, lag, spread, yoyDayCounter, paymentCalendar*`)`

```
swapType = ql.YearOnYearInflationSwap.Payer
nominal = 1e6
startDate = ql.Date(2,10,2007)
endDate = ql.Date(2,10,2052)

fixedSchedule = ql.MakeSchedule(startDate, endDate, ql.Period('6m'))
fixedRate = 0.1;
fixedDayCounter = ql.Actual365Fixed()
yoySchedule = ql.MakeSchedule(startDate, endDate, ql.Period('6m'))
index = ql.YYEUHICP(False)
lag = ql.Period('3m')
spread = 0.0
yoyDayCounter = ql.Actual365Fixed()
paymentCalendar = ql.TARGET()

swap = ql.YearOnYearInflationSwap(swapType, nominal, fixedSchedule, fixedRate, fixedDayCounter,␣
↪yoySchedule, index, lag, spread, yoyDayCounter, paymentCalendar)
```

6.2.5 YoYInflationCap

6.2.6 YoYInflationFloor

6.2.7 YoYInflationCollar

6.3 Credit

6.3.1 CreditDefaultSwap

`ql.CreditDefaultSwap(`*side, nominal, spread, cdsSchedule, convention, dayCounter*`)`

```
side = ql.Protection.Seller
nominal = 10e6
spread = 34.6 / 10000
cdsSchedule = ql.MakeSchedule(ql.Date(20, 12, 2019), ql.Date(20, 12, 2024), ql.Period('3M'),
                            ql.Quarterly, ql.TARGET(), ql.Following, ql.Unadjusted, ql.DateGeneration.
↪TwentiethIMM)

cds = ql.CreditDefaultSwap(side, nominal, spread, cdsSchedule, ql.Following, ql.Actual360())
```

6.3.2 CdsOption

`ql.CdsOption(`*CreditDefaultSwap, exercise, knocksOut=true*`)`

```
expiry = ql.Date(15,6,2020)
exercise = ql.EuropeanExercise(expiry)
ql.CdsOption(cds, exercise, True)
```

6.4 Options

6.4.1 Vanilla Options

`ql.VanillaOption`(*payoff*, *europeanExercise*)

Exercise Types:

- *ql.EuropeanExercise(date)*
- *ql.AmericanExercise(earliestDate, latestDate)*
- *ql.BermudanExercise(dates)*
- *ql.RebatedExercise*

Payoffs:

- *ql.Option.Call*
- *ql.Option.Put*

```
strike = 100.0
maturity = ql.Date(15,6,2025)
option_type = ql.Option.Call

payoff = ql.PlainVanillaPayoff(option_type, strike)
binaryPayoff = ql.CashOrNothingPayoff(option_type, strike, 1)

europeanExercise = ql.EuropeanExercise(maturity)
europeanOption = ql.VanillaOption(payoff, europeanExercise)

americanExercise = ql.AmericanExercise(ql.Date().todaysDate(), maturity)
americanOption = ql.VanillaOption(payoff, americanExercise)

bermudanExercise = ql.BermudanExercise([ql.Date(15,6,2024), ql.Date(15,6,2025)])
bermudanOption = ql.VanillaOption(payoff, bermudanExercise)

binaryOption = ql.VanillaOption(binaryPayoff, european_exercise)
```

6.4.2 Asian Options

`ql.DiscreteAveragingAsianOption`(*averageType*, *runningAccumulator*, *pastFixings*, *fixingDates*, *payoff*, *exercise*)

Averaging Types:

- *ql.ContinuousAveragingAsianOption(arithmeticAverage, vanillaPayoff, europeanExercise)*
- *ql.DiscreteAveragingAsianOption(arithmeticAverage, arithmeticRunningAccumulator, pastFixings, asianFutureFixingDates, vanillaPayoff, europeanExercise)*

Average Definitions:

- *ql.Average().Arithmetic*
- *ql.Average().Geometric*

```
today = ql.Date().todaysDate()
periods = [ql.Period("6M"), ql.Period("12M"), ql.Period("18M"), ql.Period("24M")]

pastFixings = 0 # Empty because this is a new contract
asianFutureFixingDates = [today + period for period in periods]
asianExpiryDate = today + periods[-1]

strike = 100
vanillaPayoff = ql.PlainVanillaPayoff(ql.Option.Call, strike)
europeanExercise = ql.EuropeanExercise(asianExpiryDate)
```

(continues on next page)

(continued from previous page)

```
arithmeticAverage = ql.Average().Arithmetic
arithmeticRunningAccumulator = 0.0
discreteArithmeticAsianOption = ql.DiscreteAveragingAsianOption(arithmeticAverage,␣
↪arithmeticRunningAccumulator, pastFixings, asianFutureFixingDates, vanillaPayoff, europeanExercise)

geometricAverage = ql.Average().Geometric
geometricRunningAccumulator = 1.0
discreteGeometricAsianOption = ql.DiscreteAveragingAsianOption(geometricAverage,␣
↪geometricRunningAccumulator, pastFixings, asianFutureFixingDates, vanillaPayoff, europeanExercise)

continuousGeometricAsianOption = ql.ContinuousAveragingAsianOption(geometricAverage, vanillaPayoff,␣
↪europeanExercise)
```

6.4.3 Barrier Options

`ql.BarrierOption(`*barrierType*, *barrier*, *rebate*, *payoff*, *exercise*`)`

Barrier Types:

- *ql.Barrier.UpIn*
- *ql.Barrier.UpOut*
- *ql.Barrier.DownIn*
- *ql.Barrier.DownOut*

```
T = 1
K = 100.
barrier = 110.
rebate = 0.
barrierType = ql.Barrier.UpOut

today = ql.Date().todaysDate()
maturity = today + ql.Period(int(T*365), ql.Days)

payoff = ql.PlainVanillaPayoff(ql.Option.Call, K)
amExercise = ql.AmericanExercise(today, maturity, True)
euExercise = ql.EuropeanExercise(maturity)

barrierOption = ql.BarrierOption(barrierType, barrier, rebate, payoff, euExercise)
```

`ql.DoubleBarrierOption(`*barrierType*, *barrier_lo*, *barrier_hi*, *rebate*, *payoff*, *exercise*`)`

Double Barrier Types:

- *ql.DoubleBarrier.KnockIn*
- *ql.DoubleBarrier.KnockOut*
- *ql.DoubleBarrier.KIKO*
- *ql.DoubleBarrier.KOKI*

```
T = 1
K = 100.
barrier_lo, barrier_hi = 90., 110.
rebate = 0.
barrierType = ql.DoubleBarrier.KnockOut

today = ql.Date().todaysDate()
maturity = today + ql.Period(int(T*365), ql.Days)

payoff = ql.PlainVanillaPayoff(ql.Option.Call, K)
euExercise = ql.EuropeanExercise(maturity)
```

(continues on next page)

(continued from previous page)

```
doubleBarrierOption = ql.DoubleBarrierOption(barrierType, barrier_lo, barrier_hi, rebate, payoff,␣
↪euExercise)
```

6.4.4 Basket Options

ql.BasketOption(*payoff*, *exercise*)

Payoff Types:

- *ql.MinBasketPayoff(payoff)*
- *ql.AverageBasketPayoff(payoff, numInstruments)*
- *ql.MaxBasketPayoff(payoff)*

```
today = ql.Date().todaysDate()
exp_date = today + ql.Period(1, ql.Years)
strike = 100
number_of_underlyings = 5

exercise = ql.EuropeanExercise(exp_date)
vanillaPayoff = ql.PlainVanillaPayoff(ql.Option.Call, strike)

payoffMin = ql.MinBasketPayoff(vanillaPayoff)
basketOptionMin = ql.BasketOption(payoffMin, exercise)

payoffAverage = ql.AverageBasketPayoff(vanillaPayoff, number_of_underlyings)
basketOptionAverage = ql.BasketOption(payoffAverage, exercise)

payoffMax = ql.MaxBasketPayoff(vanillaPayoff)
basketOptionMax = ql.BasketOption(payoffMax, exercise)
```

6.4.5 Cliquet Options

6.4.6 Forward Options

ql.ForwardVanillaOption(*moneyness*, *resetDate*, *payoff*, *exercise*)

```
today = ql.Date().todaysDate()
resetDate = today + ql.Period(1, ql.Years)
expiryDate = today + ql.Period(2, ql.Years)
moneyness, strike = 1., 100 # nb. strike is required for the payoff, but ignored in pricing

exercise = ql.EuropeanExercise(expiryDate)
vanillaPayoff = ql.PlainVanillaPayoff(ql.Option.Call, strike)

forwardStartOption = ql.ForwardVanillaOption(moneyness, resetDate, vanillaPayoff, exercise)
```

6.4.7 Quanto Options

Chapter 7

Math Tools

7.1 Solvers

QuantLib provides several types of one-dimensional solvers to solve the roots of single-parameter functions,

$$f(x) = 0$$

Where $f : R \to R$ is a function over a real number field.

The types of solvers provided by QuantLib are:

- Brent
- Bisection
- Secant
- Ridder
- Newton (requires member function derivative , calculates derivative)
- FalsePosition

The constructors for these solvers are default constructors and do not accept parameters. For example, the construction statement for the Brent solver instance is:

```
mySolv = Brent()
```

There are two ways to call the solver's member function:

```
mySolv.solve(f, accuracy, guess, step)
mySolv.solve(f, accuracy, guess, xMin, xMax)
```

f	single parameter function or function object, the return value is a floating point number
accuracy	Floating-point number representing the solution precision used to stop the calculation
guess	a floating-point number, the initial guess for the root
step	Floating point number. In the first calling method, there is no limit to the range of the root. The algorithm needs to search by itself to determine a range. step specifies the step size of the search algorithm.
xMin, xMax	floating point numbers, left and right interval range

```
ql.Settings.instance().evaluationDate = ql.Date(15,6,2020)
crv = ql.FlatForward(2, ql.TARGET(), 0.05, ql.Actual360())
yts = ql.YieldTermStructureHandle(crv)
engine = ql.DiscountingSwapEngine(yts)
```

(continues on next page)

(continued from previous page)

```
schedule = ql.MakeSchedule(ql.Date(15,9,2020), ql.Date(15,9,2021), ql.Period('6M'))
index = ql.Euribor3M(yts)
floatingLeg = ql.IborLeg([100], schedule, index)

def swapFairRate(rate):
    fixedLeg = ql.FixedRateLeg(schedule, ql.Actual360(), [100.], [rate])
    swap = ql.Swap(fixedLeg, floatingLeg)
    swap.setPricingEngine(engine)
    return swap.NPV()

solver = ql.Brent()

accuracy = 1e-5
guess = 0.0
step = 0.001
solver.solve(swapFairRate, accuracy, guess, step)
```

7.2 Integration

7.2.1 Gaussian Quadrature

Gaussian Quadrature evaluates an integral on a set interval. For example, Gauss-Legendre evaluates the definite integral on the inverval [-1,1]

```
import numpy as np
import QuantLib as ql

f = lambda x: x**2
g = lambda x: np.exp(x)

quad_ql_legendre = ql.GaussLegendreIntegration(128)
quad_ql_legendre(f), quad_ql_legendre(g)
```

Scipy also has an implementation that we can compare:

```
from scipy.integrate import quad
quad(f, -1, 1)[0], quad(g, -1, 1)[0]
```

Scipy requests an interval [a,b] to operate over. We can achieve the same thing with ql if we scale the input parameters using a wrapper function:

```
def quad_ql_ab(f, a, b, quad):
    multiplier, ratio = (b+a) / 2, (b-a) / 2
    y = lambda x: f(ratio*x + multiplier)
    return quad(y) * ratio

quad_ql = ql.GaussLegendreIntegration(128)
quad_ql_ab(f, -2, 2, quad_ql), quad_ql_ab(g, -2, 2, quad_ql), quad(f, -2, 2)[0], quad(g, -2, 2)[0]
```

Other supported quadratures are:

- GaussLegendreIntegration,
- GaussChebyshevIntegration,
- GaussChebyshev2ndIntegration,
- GaussLaguerreIntegration,
- GaussHermiteIntegration,

- GaussJacobiIntegration,
- GaussHyperbolicIntegration,
- GaussGegenbauerIntegration

The intervals and additional parameters for each quadrature varies (see eg. https://mathworld.wolfram.com/GaussianQuadrature.html)

7.3 Interpolation

Interpolation is one of the most commonly used tools in quantitative finance. The standard application scenario is interpolation of yield curves, volatility smile curves, and volatility surfaces. quantlib-python provides the following one- and two-dimensional interpolation methods:

`XXXInterpolation`(x, y)

- x : sequence of floating-point numbers, several discrete arguments
- y : sequence of floating-point numbers, the value of the function corresponding to the argument, the same length as x

The interpolation class defines the __call__ method. The usage of an interpolation class object is as follows, as a function

```
i(x, allowExtrapolation)
```

- x : floating point number, the point to be interpolated
- allowExtrapolation : boolean. Setting allowExtrapolation to True means extrapolation is allowed. The default value is False.

7.3.1 1D interpolation method

- **LinearInterpolation** (1-D)
- **LogLinearInterpolation** (1-D)
- **CubicInterpolation** (1-D)
- **LogCubicInterpolation** (1-D)
- **ForwardFlatInterpolation** (1-D)
- **BackwardFlatInterpolation** (1-D)
- **LogParabolicInterpolation** (1-D)

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

X = [1., 2., 3., 4., 5.]
Y = [0.5, 0.6, 0.7, 0.8, 0.9]

methods = {
    'Linear Interpolation': ql.LinearInterpolation(X, Y),
    'LogLinearInterpolation': ql.LogLinearInterpolation(X, Y),
    'CubicNaturalSpline': ql.CubicNaturalSpline(X, Y),
    'LogCubicNaturalSpline': ql.LogCubicNaturalSpline(X, Y),
    'ForwardFlatInterpolation': ql.ForwardFlatInterpolation(X, Y),
    'BackwardFlatInterpolation': ql.BackwardFlatInterpolation(X, Y),
    'LogParabolic': ql.LogParabolic(X, Y)
}

xx = np.linspace(1, 10)
fig = plt.figure(figsize=(15,4))
```

(continues on next page)

(continued from previous page)

```
plt.scatter(X, Y, label='Original Data')
for name, i in methods.items():
    yy = [i(x, allowExtrapolation=True) for x in xx]
    plt.plot(xx, yy, label=name);
plt.legend();
```

7.3.2 2D Interpolation Methods

- **BilinearInterpolation** (2-D)
- **BicubicSpline** (2-D)

```
import pandas as pd
X = [1., 2., 3., 4., 5.]
Y = [0.6, 0.7, 0.8, 0.9]
Z = [[(x-3)**2 + y for x in X] for y in Y]
df = pd.DataFrame(Z, columns=X, index=Y)

i = ql.BilinearInterpolation(X, Y, Z)

XX = np.linspace(0, 5, 9)
YY = np.linspace(0.55, 1.0, 10)

extrapolated = pd.DataFrame(
    [[i(x,y, True) for x in XX] for y in YY],
    columns=XX,
    index=YY)


from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.set_title("Surface Interpolation")

Xs, Ys = np.meshgrid(XX, YY)
surf = ax.plot_surface(
    Xs, Ys, extrapolated, rstride=1, cstride=1, cmap=cm.coolwarm
)
fig.colorbar(surf, shrink=0.5, aspect=5);
```

7.4 Optimization

7.5 Random Number Generators

Quantlib-Python provides the following three uniformly distributed (pseudo) random number generators:

- ql.KnuthUniformRng, Knuth algorithm
- ql.LecuyerUniformRng, L'Ecuyer algorithm
- ql.MersenneTwisterUniformRng, the famous Mersenne-Twister algorithm

The constructor of the random number generator,

`RandomNumberGenerator(seed)`

where seed is an integer, with a default value of 0, used as a seed to initialize the corresponding deterministic sequence

Member functions of the random number generator:

- next() : Returns a SampleNumber object as the result of the simulation.

```
r = rng.next()
v = r.value(r)
```

The user obtains a series of random numbers by repeatedly calling the member function next(). It should be noted that the type of r is SampleNumber , and the corresponding floating-point number needs to be called by calling value() .

The most common distribution in random simulations is the normal distribution. There are four types of normally distributed random number generators provided by quantlib-python:

- *CentralLimit[X]GaussianRng*
- *BoxMuller[X]GaussianRng*
- *MoroInvCumulative[X]GaussianRng*
- *InvCumulative[X]GaussianRng*

Where [X] refers to a uniform random number generator.

Specifically, there are 12 types of 4 types of generators:

- CentralLimitLecuyerGaussianRng
- CentralLimitKnuthGaussianRng
- CentralLimitMersenneTwisterGaussianRng
- BoxMullerLecuyerGaussianRng
- BoxMullerKnuthGaussianRng
- BoxMullerMersenneTwisterGaussianRng
- MoroInvCumulativeLecuyerGaussianRng
- MoroInvCumulativeKnuthGaussianRng
- MoroInvCumulativeMersenneTwisterGaussianRng
- InvCumulativeLecuyerGaussianRng
- InvCumulativeKnuthGaussianRng
- InvCumulativeMersenneTwisterGaussianRng

Constructor of random number generator:

`GaussianRandomNumberGenerator(RandomNumberGenerator)`

BoxMullerMersenneTwisterGaussianRng distributed random number generators accept a corresponding uniformly distributed random number generator as the source.

Taking *BoxMullerMersenneTwisterGaussianRng* as an example, you need to configure a *MersenneTwisterUniformRng* object as the source of random numbers, and use the classic Box-Muller algorithm to obtain normal distributed random numbers.

```
seed = 12324
unifMt = ql.MersenneTwisterUniformRng(seed)
bmGauss = ql.BoxMullerMersenneTwisterGaussianRng(unifMt)

for i in range(5):
    print(bmGauss.next().value())
```

Compared with the “pseudo” random numbers described earlier, another important type of random numbers in random simulations becomes “quasi” random numbers, also known as low-bias sequences. Because of better convergence, quasi-random numbers are often used in the simulation of high-dimensional random variables. There are two types of quasi-random numbers provided by quantlib-python,

- **HaltonRsg** : Halton sequence
- **SobolRsg** : Sobol sequence

7.5.1 HaltonRsg

`HaltonRsg(`*dimensionality*, *seed*, *randomStart*, *randomShift*`)`

where,

- dimensionality : integer, set the dimension;
- seed : an integer, with a default value of 0, used as a seed to initialize the corresponding deterministic sequence;
- randomStart : boolean, the default is True , whether to start randomly;
- randomShift : Boolean, default is False , whether to shift randomly.

Member function of *HaltonRsg*,

- **nextSequence()** : returns a SampleRealVector object as the result of the simulation;
- **lastSequence()** : returns a SampleRealVector object as the result of the previous simulation;
- **dimension()** : Returns the dimension.

7.5.2 SobolRsg

`SobolRsg(`*dimensionality*, *seed*, *directionIntegers=Jaeckel*`)`

where,

- dimensionality : integer, set the dimension;
- seed : an integer, with a default value of 0, used as a seed to initialize the corresponding deterministic sequence;
- directionIntegers : a built-in variable of quantlib-python. The default value is SobolRsg.Jaeckel , which is used to initialize Sobol sequences.

Member functions of *SobolRsg*,

- **nextSequence()** : returns a SampleRealVector object as the result of the simulation;
- **lastSequence()** : returns a SampleRealVector object as the result of the previous simulation;
- **dimension()** : Returns the dimension.
- **skipTo(n)** : n is an integer, skip to the nth dimension of the sampling result;
- **nextInt32Sequence()** : Returns an IntVector object.

7.6 Path Generators

QuantLib provides several wrapper functions to produce sample paths from a given stochastic process

7.6.1 GaussianMultiPathGenerator

Generate paths from an arbitrary stochastic process using pseudorandom numbers

`ql.GaussianMultiPathGenerator(`*stochasticProcess*, *times*, *sequenceGenerator*, *brownianBridge=False*`)`

```
timestep, length, numPaths = 24, 2, 2**15

today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))

v0, kappa, theta, rho, sigma = 0.005, 0.8, 0.008, 0.2, 0.1
hestonProcess = ql.HestonProcess(riskFreeTS, dividendTS, initialValue, v0, kappa, theta, sigma, rho)
```

(continues on next page)

(continued from previous page)

```
times = ql.TimeGrid(length, timestep)
dimension = hestonProcess.factors()

rng = ql.UniformRandomSequenceGenerator(dimension * timestep, ql.UniformRandomGenerator())
sequenceGenerator = ql.GaussianRandomSequenceGenerator(rng)
pathGenerator = ql.GaussianMultiPathGenerator(hestonProcess, list(times), sequenceGenerator, False)

# paths[0] will contain spot paths, paths[1] will contain vol paths
paths = [[] for i in range(dimension)]
for i in range(numPaths):
    samplePath = pathGenerator.next()
    values = samplePath.value()
    spot = values[0]

    for j in range(dimension):
        paths[j].append([x for x in values[j]])
```

7.6.2 GaussianSobolMultiPathGenerator

Generate paths from an arbitrary stochastic process using low discrepency numbers

`ql.GaussianSobolMultiPathGenerator`(*stochasticProcess*, *times*, *sequenceGenerator*, *brownianBridge=False*)

```
timestep, length, numPaths = 24, 2, 2**15

today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))

v0, kappa, theta, rho, sigma = 0.005, 0.8, 0.008, 0.2, 0.1
hestonProcess = ql.HestonProcess(riskFreeTS, dividendTS, initialValue, v0, kappa, theta, sigma, rho)

times = ql.TimeGrid(length, timestep)
dimension = hestonProcess.factors()

rng = ql.UniformLowDiscrepancySequenceGenerator(dimension * timestep)
sequenceGenerator = ql.GaussianLowDiscrepancySequenceGenerator(rng)
pathGenerator = ql.GaussianSobolMultiPathGenerator(hestonProcess, list(times), sequenceGenerator, False)

# paths[0] will contain spot paths, paths[1] will contain vol paths
paths = [[] for i in range(dimension)]
for i in range(numPaths):
    samplePath = pathGenerator.next()
    values = samplePath.value()
    spot = values[0]

    for j in range(dimension):
        paths[j].append([x for x in values[j]])
```

7.7 Statistics

7.8 Convention Calculators

7.8.1 BlackDeltaCalculator

A calculator class to calculate the relevant strike for FX-style delta-maturity-vol quotes (see Reiswich D., Wystup U., 2010. A Guide to FX Options Quoting Conventions)

`ql.BlackDeltaCalculator(`*optionType, deltaType, spot, domesticDcf, foreignDcf, volRootT*`)`

```
import numpy as np

today = ql.Date().todaysDate()
dayCounter = ql.Actual365Fixed()
spot = 100
rd, rf = 0.02, 0.05

ratesTs = ql.YieldTermStructureHandle(ql.FlatForward(today, rd, dayCounter))
dividendTs = ql.YieldTermStructureHandle(ql.FlatForward(today, rf, dayCounter))

# Details about the delta quote
optionType = ql.Option.Put
vol = 0.07
maturity = 1.0
deltaType = ql.DeltaVolQuote.Fwd      # Also supports: Spot, PaSpot, PaFwd

# Set up the calculator
localDcf, foreignDcf = ratesTs.discount(maturity), dividendTs.discount(maturity)
stdDev = np.sqrt(maturity) * vol
calc = ql.BlackDeltaCalculator(optionType, deltaType, spot, localDcf, foreignDcf, stdDev)
```

To calculate the strike for a given call/put delta (negative for put delta)

```
delta = -0.3
calc.strikeFromDelta(delta)
```

Or if this is an ATM quote, specify the ATM convention

```
atmType = ql.DeltaVolQuote.AtmFwd     # Also supports: AtmSpot, AtmDeltaNeutral, AtmVegaMax, AtmGammaMax,
↪ AtmPutCall50
calc.atmStrike(atmType)
```

Chapter 8

Pricing Engines

8.1 Bond Pricing Engines

8.1.1 DiscountingBondEngine

`ql.DiscountingBondEngine(`*discountCurve*`)`

```
crv = ql.FlatForward(ql.Date().todaysDate(),0.04875825,ql.Actual365Fixed())
yts = ql.YieldTermStructureHandle(crv)
engine = ql.DiscountingBondEngine(yts)
```

8.1.2 BlackCallableFixedRateBondEngine

`ql.BlackCallableFixedRateBondEngine(`*fwdYieldVol, discountCurve*`)`

```
crv = ql.FlatForward(ql.Date().todaysDate(),0.04875825,ql.Actual365Fixed())
yts = ql.YieldTermStructureHandle(crv)
vol = ql.QuoteHandle(ql.SimpleQuote(0.55))
engine = ql.BlackCallableFixedRateBondEngine(vol, yts)
```

8.1.3 TreeCallableFixedRateEngine

`ql.TreeCallableFixedRateBondEngine(`*shortRateModel, size, discountCurve*`)`

```
crv = ql.FlatForward(ql.Date().todaysDate(),0.04875825,ql.Actual365Fixed())
yts = ql.YieldTermStructureHandle(crv)
model = ql.Vasicek()
engine = ql.TreeCallableFixedRateBondEngine(model, 10, yts)
```

`ql.TreeCallableFixedRateBondEngine(`*shortRateModel, size*`)`

```
model = ql.Vasicek()
engine = ql.TreeCallableFixedRateBondEngine(model, 10)
```

`ql.TreeCallableFixedRateBondEngine(`*shortRateModel, TimeGrid, discountCurve*`)`

```
crv = ql.FlatForward(ql.Date().todaysDate(),0.04875825,ql.Actual365Fixed())
yts = ql.YieldTermStructureHandle(crv)
model = ql.Vasicek()
grid = ql.TimeGrid(5,10)

engine = ql.TreeCallableFixedRateBondEngine(model, grid, yts)
```

ql.TreeCallableFixedRateBondEngine(*shortRateModel*, *TimeGrid*)

```
crv = ql.FlatForward(ql.Date().todaysDate(),0.04875825,ql.Actual365Fixed())
yts = ql.YieldTermStructureHandle(crv)
model = ql.Vasicek()
grid = ql.TimeGrid(5,10)

engine = ql.TreeCallableFixedRateBondEngine(model, grid)
```

8.2 Cap Pricing Engines

8.2.1 BlackCapFloorEngine

ql.BlackCapFloorEngine(*yieldTermStructure*, *quoteHandle*)

```
vols = ql.QuoteHandle(ql.SimpleQuote(0.547295))
engine = ql.BlackCapFloorEngine(yts, vols)
cap.setPricingEngine(engine)
```

ql.BlackCapFloorEngine(*yieldTermStructure*, *OptionletVolatilityStructure*)

8.2.2 BachelierCapFloorEngine

ql.BachelierCapFloorEngine(*yieldTermStructure*, *quoteHandle*)

```
vols = ql.QuoteHandle(ql.SimpleQuote(0.00547295))
engine = ql.BachelierCapFloorEngine(yts, vols)
```

ql.BachelierCapFloorEngine(*yieldTermStructure*, *OptionletVolatilityStructure*)

8.2.3 AnalyticCapFloorEngine

ql.AnalyticCapFloorEngine(*OneFactorAffineModel*, *YieldTermStructure*)

ql.AnalyticCapFloorEngine(*OneFactorAffineModel*)

OneFactorAffineModel

- HullWhite : (termStructure, a=0.1, sigma=0.01)
- Vasicek : (r0=0.05, a=0.1, b=0.05, sigma=0.01, lambda=0.0)
- CoxIngersollRoss [NOT IMPLEMENTED]
- GeneralizedHullWhite [NOT IMPLEMENTED]

```
yts = ql.YieldTermStructureHandle(ql.FlatForward(ql.Date().todaysDate(), 0.0121, ql.Actual360()))
models = [
    ql.HullWhite (yts),
    ql.Vasicek(r0=0.008),
]
```

(continues on next page)

(continued from previous page)

```
for model in models:
    analyticEngine = ql.AnalyticCapFloorEngine(model, yts)
    cap.setPricingEngine(analyticEngine)
    print(f"Cap npv is: {cap.NPV():,.2f}")
```

8.2.4 TreeCapFloorEngine

`ql.TreeCapFloorEngine(`*ShortRateModel, Size, YieldTermStructure*`)`

`ql.TreeCapFloorEngine(`*ShortRateModel, Size*`)`

`ql.TreeCapFloorEngine(`*ShortRateModel, Size, TimeGrid, YieldTermStructure*`)`

`ql.TreeCapFloorEngine(`*ShortRateModel, Size, TimeGrid*`)`

Models

- HullWhite : (YieldTermStructure, a=0.1, sigma=0.01)
- BlackKarasinski : (YieldTermStructure, a=0.1, sigma=0.1)
- Vasicek : (r0=0.05, a=0.1, b=0.05, sigma=0.01, lambda=0.0)
- G2 : (termStructure, a=0.1, sigma=0.01, b=0.1, eta=0.01, rho=-0.75)
- GeneralizedHullWhite [NOT IMPLEMENTED]
- CoxIngersollRoss [NOT IMPLEMENTED]
- ExtendedCoxIngersollRoss [NOT IMPLEMENTED]

```
models = [
    ql.HullWhite (yts),
    ql.BlackKarasinski(yts),
    ql.Vasicek(0.0065560),
    ql.G2(yts)
]

for model in models:
    treeEngine = ql.TreeCapFloorEngine(model, 60, yts)
    cap.setPricingEngine(treeEngine)
    print(f"Cap npv is: {cap.NPV():,.2f}")
```

8.3 Swap Pricing Engines

8.3.1 DiscountingSwapEngine

`ql.DiscountingSwapEngine(`*YieldTermStructure*`)`

```
yts = ql.YieldTermStructureHandle(ql.FlatForward(2, ql.TARGET(), 0.5, ql.Actual360()))
engine = ql.DiscountingSwapEngine(yts)
```

8.4 Swaption Pricing Engines

8.4.1 BlackSwaptionEngine

`ql.BlackSwaptionEngine(`*yts*, *quote*`)`

`ql.BlackSwaptionEngine(`*yts*, *swaptionVolatilityStructure*`)`

`ql.BlackSwaptionEngine(`*yts*, *quote*, *dayCounter*`)`

`ql.BlackSwaptionEngine(`*yts*, *quote*, *dayCounter*, *displacement*`)`

```
blackEngine = ql.BlackSwaptionEngine(yts, ql.QuoteHandle(ql.SimpleQuote(0.55)))
blackEngine = ql.BlackSwaptionEngine(yts, ql.QuoteHandle(ql.SimpleQuote(0.55)), ql.ActualActual())
blackEngine = ql.BlackSwaptionEngine(yts, ql.QuoteHandle(ql.SimpleQuote(0.55)), ql.ActualActual(), 0.01)
```

8.4.2 BachelierSwaptionEngine

`ql.BachelierSwaptionEngine(`*yts*, *quote*`)`

`ql.BachelierSwaptionEngine(`*yts*, *swaptionVolatilityStructure*`)`

`ql.BachelierSwaptionEngine(`*yts*, *quote*, *dayCounter*`)`

```
bachelierEngine = ql.BachelierSwaptionEngine(yts, ql.QuoteHandle(ql.SimpleQuote(0.0055)))
swaption.setPricingEngine(bachelierEngine)
swaption.NPV()
```

8.4.3 FdHullWhiteSwaptionEngine

`ql.FdHullWhiteSwaptionEngine(`*model*, *range*, *interval*`)`

```
model = ql.HullWhite(yts)
engine = ql.FdHullWhiteSwaptionEngine(model)
swaption.setPricingEngine(engine)
swaption.NPV()
```

8.4.4 FdG2SwaptionEngine

`ql.FdG2SwaptionEngine(`*model*`)`

```
model = ql.G2(yts)
engine = ql.FdG2SwaptionEngine(model)
swaption.setPricingEngine(engine)
swaption.NPV()
```

8.4.5 G2SwaptionEngine

`ql.G2SwaptionEngine(`*model*, *range*, *interval*`)`

```
model = ql.G2(yts)
g2Engine = ql.G2SwaptionEngine(model, 4, 4)
swaption.setPricingEngine(g2Engine)
swaption.NPV()
```

8.4.6 JamshidianSwaptionEngine

ql.JamshidianSwaptionEngine(*OneFactorAffineModel*)

ql.JamshidianSwaptionEngine(*OneFactorAffineModel*, *YieldTermStructure*)

```
model = ql.HullWhite(yts)
engine = ql.JamshidianSwaptionEngine(model, yts)
swaption.setPricingEngine(g2Engine)
swaption.NPV()
```

8.4.7 TreeSwaptionEngine

ql.TreeSwaptionEngine(*ShortRateModel*, *Size*, *YieldTermStructure*)

ql.TreeSwaptionEngine(*ShortRateModel*, *Size*)

ql.TreeSwaptionEngine(*ShortRateModel*, *TimeGrid*, *YieldTermStructure*)

ql.TreeSwaptionEngine(*ShortRateModel*, *TimeGrid*)

```
model = ql.HullWhite(yts)
engine = ql.TreeSwaptionEngine(model, 10)
swaption.setPricingEngine(g2Engine)
swaption.NPV()
```

8.5 Credit Pricing Engines

8.5.1 IsdaCdsEngine

ql.IsdaCdsEngine(*defaultProbability*, *recoveryRate*, *yieldTermStructure*, *includeSettlementDateFlows=None*, *numericalFix=ql.IsdaCdsEngine.Taylor*, *AccrualBias accrualBias=ql.IsdaCdsEngine.HalfDayBias*, *forwardsInCouponPeriod=ql.IsdaCdsEngine.Piecewise*)

```
today = ql.Date().todaysDate()
defaultProbability = ql.DefaultProbabilityTermStructureHandle(
    ql.FlatHazardRate(today, ql.QuoteHandle(ql.SimpleQuote(0.01)), ql.Actual360())
)
yieldTermStructure = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual360()))

recoveryRate = 0.4
engine = ql.IsdaCdsEngine(defaultProbability, recoveryRate, yieldTermStructure)
```

8.5.2 MidPointCdsEngine

ql.MidPointCdsEngine(*defaultProbability*, *recoveryRate*, *yieldTermStructure*)

```
today = ql.Date().todaysDate()
defaultProbability = ql.DefaultProbabilityTermStructureHandle(
    ql.FlatHazardRate(today, ql.QuoteHandle(ql.SimpleQuote(0.01)), ql.Actual360())
)
yieldTermStructure = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual360()))

recoveryRate = 0.4
engine = ql.MidPointCdsEngine(defaultProbability, recoveryRate, yieldTermStructure)
```

8.5.3 IntegralCdsEngine

`ql.IntegralCdsEngine(`*integrationStep, probability, recoveryRate, discountCurve, includeSettlementDateFlows=False*`)`

```
today = ql.Date().todaysDate()
defaultProbability = ql.DefaultProbabilityTermStructureHandle(
    ql.FlatHazardRate(today, ql.QuoteHandle(ql.SimpleQuote(0.01)), ql.Actual360())
)
yieldTermStructure = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual360()))

integralStep = ql.Period('1d')
engine = ql.IntegralCdsEngine(integralStep, defaultProbability, 0.4, yieldTermStructure,␣
↪includeSettlementDateFlows=False)
```

8.5.4 BlackCdsOptionEngine

`ql.BlackCdsOptionEngine(`*defaultProbability, recoveryRate, yieldTermStructure, vol*`)`

```
today = ql.Date().todaysDate()
defaultProbability = ql.DefaultProbabilityTermStructureHandle(
    ql.FlatHazardRate(today, ql.QuoteHandle(ql.SimpleQuote(0.01)), ql.Actual360())
)
yieldTermStructure = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual360()))


vol = ql.QuoteHandle(ql.SimpleQuote(0.2))
engine = ql.BlackCdsOptionEngine(defaultProbability, 0.4, yieldTermStructure, vol)
```

8.6 Option Pricing Engines

8.6.1 Vanilla Options

8.6.1.1 AnalyticEuropeanEngine

`ql.AnalyticEuropeanEngine(`*GeneralizedBlackScholesProcess*`)`

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
volatility = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.NullCalendar(), 0.1, ql.
↪Actual365Fixed()))
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
process = ql.BlackScholesMertonProcess(initialValue, dividendTS, riskFreeTS, volatility)

engine = ql.AnalyticEuropeanEngine(process)
```

8.6.1.2 MCEuropeanEngine

`ql.MCEuropeanEngine(`*GeneralizedBlackScholesProcess, traits, timeSteps=None, timeStepsPerYear=None, brownianBridge=False, antitheticVariate=False, requiredSamples=None, requiredTolerance=None, maxSamples=None, seed=0*`)`

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
volatility = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.NullCalendar(), 0.1, ql.
↪Actual365Fixed()))
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
process = ql.BlackScholesMertonProcess(initialValue, dividendTS, riskFreeTS, volatility)

steps = 2
rng = "pseudorandom" # could use "lowdiscrepancy"
numPaths = 100000

engine = ql.MCEuropeanEngine(process, rng, steps, requiredSamples=numPaths)
```

8.6.1.3 FdBlackScholesVanillaEngine

Note that this engine is capable of pricing both European and American payoffs!

`ql.FdBlackScholesVanillaEngine(`*GeneralizedBlackScholesProcess, tGrid, xGrid, dampingSteps=0, schemeDesc=ql.FdmSchemeDesc.Douglas(), localVol=False, illegalLocalVolOverwrite=None*`)`

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
volatility = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.NullCalendar(), 0.1, ql.
↪Actual365Fixed()))
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
process = ql.BlackScholesMertonProcess(initialValue, dividendTS, riskFreeTS, volatility)

tGrid, xGrid = 2000, 200
engine = ql.FdBlackScholesVanillaEngine(process, tGrid, xGrid)
```

8.6.1.4 MCAmericanEngine

`ql.MCAmericanEngine(`*GeneralizedBlackScholesProcess, traits, timeSteps=None, timeStepsPerYear=None, antitheticVariate=False, controlVariate=False, requiredSamples=None, requiredTolerance=None, maxSamples=None, seed=0, polynomOrder=2, polynomType=0, nCalibrationSamples=2048, antitheticVariateCalibration=None, seedCalibration=None*`)`

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
volatility = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.NullCalendar(), 0.1, ql.
↪Actual365Fixed()))
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
process = ql.BlackScholesMertonProcess(initialValue, dividendTS, riskFreeTS, volatility)

steps = 200
rng = "pseudorandom" # could use "lowdiscrepancy"
numPaths = 100000

engine = ql.MCAmericanEngine(process, rng, steps, requiredSamples=numPaths)
```

8.6.1.5 MCDigitalEngine

This engine prices american (ie. knock-in) cash-or-nothing payoffs only

ql.MCDigitalEngine(*GeneralizedBlackScholesProcess, traits, timeSteps=None, timeStepsPerYear=None, brownianBridge=False, antitheticVariate=False, requiredSamples=None, requiredTolerance=None, maxSamples=None, seed=0*)

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
volatility = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.NullCalendar(), 0.1, ql.
→Actual365Fixed()))
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
process = ql.BlackScholesMertonProcess(initialValue, dividendTS, riskFreeTS, volatility)

steps = 200
rng = "pseudorandom" # could use "lowdiscrepancy"
numPaths = 100000

engine = ql.MCDigitalEngine(process, rng, steps, requiredSamples=numPaths)
```

8.6.1.6 AnalyticHestonEngine

ql.AnalyticHestonEngine(*HestonModel*)

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))

initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
v0 = 0.005
kappa = 0.8
theta = 0.008
rho = 0.2
sigma = 0.1

hestonProcess = ql.HestonProcess(riskFreeTS, dividendTS, initialValue, v0, kappa, theta, sigma, rho)
hestonModel = ql.HestonModel(hestonProcess)

engine = ql.AnalyticHestonEngine(hestonModel)
```

8.6.1.7 MCEuropeanHestonEngine

ql.MCEuropeanHestonEngine(*HestonProcess, traits, timeSteps=None, timeStepsPerYear=None, antitheticVariate=False, requiredSamples=None, requiredTolerance=None, maxSamples=None, seed=0*)

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))

initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
v0 = 0.005
kappa = 0.8
theta = 0.008
rho = 0.2
sigma = 0.1

hestonProcess = ql.HestonProcess(riskFreeTS, dividendTS, initialValue, v0, kappa, theta, sigma, rho)
```

(continues on next page)

(continued from previous page)

```
steps = 2
rng = "pseudorandom" # could use "lowdiscrepancy"
numPaths = 100000

engine = ql.MCEuropeanHestonEngine(hestonProcess, rng, steps, requiredSamples=numPaths)
```

8.6.1.8 FdHestonVanillaEngine

If a leverage function (and optional mixing factor) is passed in to this function, it prices using the Heston Stochastic Local Vol model

ql.FdHestonVanillaEngine(*HestonModel, tGrid=100, xGrid=100, vGrid=50, dampingSteps=0, FdmSchemeDesc=ql.FdmSchemeDesc.Hundsdorfer(), leverageFct=LocalVolTermStructure(), mixingFactor=1.0*)

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))

initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
v0 = 0.005
kappa = 0.8
theta = 0.008
rho = 0.2
sigma = 0.1

hestonProcess = ql.HestonProcess(riskFreeTS, dividendTS, initialValue, v0, kappa, theta, sigma, rho)
hestonModel = ql.HestonModel(hestonProcess)

tGrid, xGrid, vGrid = 100, 100, 50
dampingSteps = 0
fdScheme = ql.FdmSchemeDesc.ModifiedCraigSneyd()

engine = ql.FdHestonVanillaEngine(hestonModel, tGrid, xGrid, vGrid, dampingSteps, fdScheme)
```

8.6.1.9 AnalyticPTDHestonEngine

ql.AnalyticPTDHestonEngine(*PiecewiseTimeDependentHestonModel*)

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))

initialValue = ql.QuoteHandle(ql.SimpleQuote(100))

times = [1.0, 2.0, 3.0]
grid = ql.TimeGrid(times)

v0 = 0.005
theta = [0.010, 0.015, 0.02]
kappa = [0.600, 0.500, 0.400]
sigma = [0.400, 0.350, 0.300]
rho = [-0.15, -0.10, -0.00]

kappaTS = ql.PiecewiseConstantParameter(times[:-1], ql.PositiveConstraint())
thetaTS = ql.PiecewiseConstantParameter(times[:-1], ql.PositiveConstraint())
rhoTS = ql.PiecewiseConstantParameter(times[:-1], ql.BoundaryConstraint(-1.0, 1.0))
```

(continues on next page)

(continued from previous page)

```
sigmaTS = ql.PiecewiseConstantParameter(times[:-1], ql.PositiveConstraint())

for i, time in enumerate(times):
    kappaTS.setParam(i, kappa[i])
    thetaTS.setParam(i, theta[i])
    rhoTS.setParam(i, rho[i])
    sigmaTS.setParam(i, sigma[i])

hestonModelPTD = ql.PiecewiseTimeDependentHestonModel(riskFreeTS, dividendTS, initialValue, v0, thetaTS, kappaTS, sigmaTS, rhoTS, grid)
engine = ql.AnalyticPTDHestonEngine(hestonModelPTD)
```

8.6.2 Asian Options

8.6.2.1 AnalyticDiscreteGeometricAveragePriceAsianEngine

`ql.AnalyticDiscreteGeometricAveragePriceAsianEngine(`*GeneralizedBlackScholesProcess*`)`

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
volatility = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.NullCalendar(), 0.1, ql.Actual365Fixed()))
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
process = ql.BlackScholesMertonProcess(initialValue, dividendTS, riskFreeTS, volatility)

engine = ql.AnalyticDiscreteGeometricAveragePriceAsianEngine(process)
```

8.6.2.2 AnalyticContinuousGeometricAveragePriceAsianEngine

`ql.AnalyticContinuousGeometricAveragePriceAsianEngine(`*GeneralizedBlackScholesProcess*`)`

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
volatility = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.NullCalendar(), 0.1, ql.Actual365Fixed()))
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
process = ql.BlackScholesMertonProcess(initialValue, dividendTS, riskFreeTS, volatility)

engine = ql.AnalyticContinuousGeometricAveragePriceAsianEngine(process)
```

8.6.2.3 MCDiscreteGeometricAPEngine

`ql.MCDiscreteGeometricAPEngine(`*GeneralizedBlackScholesProcess, traits, brownianBridge=False, antitheticVariate=False, requiredSamples=None, requiredTolerance=None, maxSamples=None, seed=0*`)`

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
volatility = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.NullCalendar(), 0.1, ql.Actual365Fixed()))
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
process = ql.BlackScholesMertonProcess(initialValue, dividendTS, riskFreeTS, volatility)
```

(continues on next page)

(continued from previous page)

```
rng = "pseudorandom" # could use "lowdiscrepancy"
numPaths = 100000

engine = ql.MCDiscreteGeometricAPEngine(process, rng, requiredSamples=numPaths)
```

8.6.2.4 MCDiscreteArithmeticAPEngine

`ql.MCDiscreteArithmeticAPEngine`(*GeneralizedBlackScholesProcess, traits, brownianBridge=False, antitheticVariate=False, controlVariate=False, requiredSamples=None, requiredTolerance=None, maxSamples=None, seed=0*)

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
volatility = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.NullCalendar(), 0.1, ql.
↪Actual365Fixed()))
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
process = ql.BlackScholesMertonProcess(initialValue, dividendTS, riskFreeTS, volatility)

rng = "pseudorandom" # could use "lowdiscrepancy"
numPaths = 100000

engine = ql.MCDiscreteArithmeticAPEngine(process, rng, requiredSamples=numPaths)
```

8.6.2.5 FdBlackScholesAsianEngine

Note that this engine will throw an error if asked to price Geometric averaging options. It only prices Discrete Arithmetic Asians.

`ql.FdBlackScholesAsianEngine`(*GeneralizedBlackScholesProcess, tGrid=100, xGrid=100, aGrid=50*)

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
volatility = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.NullCalendar(), 0.1, ql.
↪Actual365Fixed()))
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
process = ql.BlackScholesMertonProcess(initialValue, dividendTS, riskFreeTS, volatility)

tGrid, xGrid, aGrid = 100, 100, 50
engine = ql.FdBlackScholesAsianEngine(process, tGrid=tGrid, xGrid=xGrid, aGrid=aGrid)
```

8.6.2.6 AnalyticDiscreteGeometricAveragePriceAsianHestonEngine

`ql.AnalyticDiscreteGeometricAveragePriceAsianHestonEngine`(*HestonProcess*)

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))

v0, kappa, theta, rho, sigma = 0.005, 0.8, 0.008, 0.2, 0.1
hestonProcess = ql.HestonProcess(riskFreeTS, dividendTS, initialValue, v0, kappa, theta, sigma, rho)

engine = ql.AnalyticDiscreteGeometricAveragePriceAsianHestonEngine(hestonProcess)
```

8.6.2.7 AnalyticContinuousGeometricAveragePriceAsianHestonEngine

`ql.AnalyticContinuousGeometricAveragePriceAsianHestonEngine(`*HestonProcess*`)`

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))

v0, kappa, theta, rho, sigma = 0.005, 0.8, 0.008, 0.2, 0.1
hestonProcess = ql.HestonProcess(riskFreeTS, dividendTS, initialValue, v0, kappa, theta, sigma, rho)

engine = ql.AnalyticContinuousGeometricAveragePriceAsianHestonEngine(hestonProcess)
```

8.6.2.8 MCDiscreteGeometricAPHestonEngine

`ql.MCDiscreteGeometricAPHestonEngine(`*HestonProcess, traits, antitheticVariate=False, requiredSamples=None, requiredTolerance=None, maxSamples=None, seed=0, timeSteps=None, timeStepsPerYear=None*`)`

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))

v0, kappa, theta, rho, sigma = 0.005, 0.8, 0.008, 0.2, 0.1
hestonProcess = ql.HestonProcess(riskFreeTS, dividendTS, initialValue, v0, kappa, theta, sigma, rho)

rng = "pseudorandom" # could use "lowdiscrepancy"
numPaths = 100000

engine = ql.MCDiscreteGeometricAPHestonEngine(hestonProcess, rng, requiredSamples=numPaths)
```

8.6.2.9 MCDiscreteArithmeticAPHestonEngine

`ql.MCDiscreteArithmeticAPHestonEngine(`*HestonProcess, traits, antitheticVariate=False, requiredSamples=None, requiredTolerance=None, maxSamples=None, seed=0, timeSteps=None, timeStepsPerYear=None, controlVariate=False*`)`

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))

v0, kappa, theta, rho, sigma = 0.005, 0.8, 0.008, 0.2, 0.1
hestonProcess = ql.HestonProcess(riskFreeTS, dividendTS, initialValue, v0, kappa, theta, sigma, rho)

rng = "pseudorandom" # could use "lowdiscrepancy"
numPaths = 100000

engine = ql.MCDiscreteArithmeticAPHestonEngine(hestonProcess, rng, requiredSamples=numPaths)
```

8.6.3 Barrier Options

8.6.3.1 BinomialBarrierEngine

`ql.BinomialBarrierEngine(`*process, type, steps*`)`

```
today = ql.Date().todaysDate()

spotHandle = ql.QuoteHandle(ql.SimpleQuote(100))
flatRateTs = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
flatVolTs = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.UnitedStates(), 0.2, ql.
↪Actual365Fixed()))
bsm = ql.BlackScholesProcess(spotHandle, flatRateTs, flatVolTs)

binomialBarrierEngine = ql.BinomialBarrierEngine(bsm, 'crr', 200)
```

8.6.3.2 AnalyticBarrierEngine

`ql.AnalyticBarrierEngine(`*process*`)`

```
today = ql.Date().todaysDate()

spotHandle = ql.QuoteHandle(ql.SimpleQuote(100))
flatRateTs = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
flatVolTs = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.UnitedStates(), 0.2, ql.
↪Actual365Fixed()))
bsm = ql.BlackScholesProcess(spotHandle, flatRateTs, flatVolTs)

analyticBarrierEngine = ql.AnalyticBarrierEngine(bsm)
```

8.6.3.3 FdBlackScholesBarrierEngine

`ql.FdBlackScholesBarrierEngine(`*process, tGrid=100, xGrid=100, dampingSteps=0, FdmSchemeDesc=ql.FdmSchemeDesc.Douglas(), localVol=False, illegalLocalVolOverwrite=None*`)`

```
today = ql.Date().todaysDate()

spotHandle = ql.QuoteHandle(ql.SimpleQuote(100))
flatRateTs = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
flatVolTs = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.UnitedStates(), 0.2, ql.
↪Actual365Fixed()))
bsm = ql.BlackScholesProcess(spotHandle, flatRateTs, flatVolTs)

fdBarrierEngine = ql.FdBlackScholesBarrierEngine(bsm)
```

8.6.3.4 FdBlackScholesRebateEngine

`ql.FdBlackScholesRebateEngine(`*process, tGrid=100, xGrid=100, dampingSteps=0, FdmSchemeDesc=ql.FdmSchemeDesc.Douglas(), localVol=False, illegalLocalVolOverwrite=None*`)`

```
today = ql.Date().todaysDate()

spotHandle = ql.QuoteHandle(ql.SimpleQuote(100))
flatRateTs = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
flatVolTs = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.UnitedStates(), 0.2, ql.
↪Actual365Fixed()))
```

(continues on next page)

(continued from previous page)

```
bsm = ql.BlackScholesProcess(spotHandle, flatRateTs, flatVolTs)

fdRebateEngine = ql.FdBlackScholesRebateEngine(bsm)
```

8.6.3.5 AnalyticBinaryBarrierEngine

`ql.AnalyticBinaryBarrierEngine(`*process*`)`

```
today = ql.Date().todaysDate()

spotHandle = ql.QuoteHandle(ql.SimpleQuote(100))
flatRateTs = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
flatVolTs = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.UnitedStates(), 0.2, ql.
↪Actual365Fixed()))
bsm = ql.BlackScholesProcess(spotHandle, flatRateTs, flatVolTs)

analyticBinaryBarrierEngine = ql.AnalyticBinaryBarrierEngine(bsm)
```

8.6.3.6 FdHestonBarrierEngine

If a leverage function (and optional mixing factor) is passed in to this function, it prices using the Heston Stochastic Local Vol model

`ql.FdHestonBarrierEngine(`*HestonModel, tGrid=100, xGrid=100, vGrid=50, dampingSteps=0, FdmSchemeDesc=ql.FdmSchemeDesc.Hundsdorfer(), leverageFct=LocalVolTermStructure(), mixingFactor=1.0*`)`

```
today = ql.Date().todaysDate()

spotHandle = ql.QuoteHandle(ql.SimpleQuote(100))
flatRateTs = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
flatDividendTs = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))

v0, kappa, theta, sigma, rho = 0.01, 2.0, 0.01, 0.01, 0.0
hestonProcess = ql.HestonProcess(flatRateTs, flatDividendTs, spotHandle, v0, kappa, theta, sigma, rho)
hestonModel = ql.HestonModel(hestonProcess)

hestonBarrierEngine = ql.FdHestonBarrierEngine(hestonModel)
```

8.6.3.7 AnalyticDoubleBarrierEngine

`ql.AnalyticDoubleBarrierEngine(`*process*`)`

```
today = ql.Date().todaysDate()

spotHandle = ql.QuoteHandle(ql.SimpleQuote(100))
flatRateTs = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
flatVolTs = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.UnitedStates(), 0.2, ql.
↪Actual365Fixed()))
bsm = ql.BlackScholesProcess(spotHandle, flatRateTs, flatVolTs)

analyticDoubleBarrierEngine = ql.AnalyticDoubleBarrierEngine(bsm)
```

8.6.3.8 AnalyticDoubleBarrierBinaryEngine

`ql.AnalyticDoubleBarrierBinaryEngine(`*process*`)`

```
today = ql.Date().todaysDate()

spotHandle = ql.QuoteHandle(ql.SimpleQuote(100))
flatRateTs = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
flatVolTs = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.UnitedStates(), 0.2, ql.
↪Actual365Fixed()))
bsm = ql.BlackScholesProcess(spotHandle, flatRateTs, flatVolTs)

analyticDoubleBinaryBarrierEngine = ql.AnalyticDoubleBarrierBinaryEngine(bsm)
```

8.6.3.9 FdHestonDoubleBarrierEngine

If a leverage function (and optional mixing factor) is passed in to this function, it prices using the Heston Stochastic Local Vol model

`ql.FdHestonDoubleBarrierEngine(`*HestonModel, tGrid=100, xGrid=100, vGrid=50, dampingSteps=0, FdmSchemeDesc=ql.FdmSchemeDesc.Hundsdorfer(), leverageFct=LocalVolTermStructure(), mixingFactor=1.0*`)`

```
today = ql.Date().todaysDate()

spotHandle = ql.QuoteHandle(ql.SimpleQuote(100))
flatRateTs = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
flatDividendTs = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))

v0, kappa, theta, sigma, rho = 0.01, 2.0, 0.01, 0.01, 0.0
hestonProcess = ql.HestonProcess(flatRateTs, flatDividendTs, spotHandle, v0, kappa, theta, sigma, rho)
hestonModel = ql.HestonModel(hestonProcess)

hestonDoubleBarrierEngine = ql.FdHestonDoubleBarrierEngine(hestonModel)
```

8.6.4 Basket Options

8.6.4.1 MCEuropeanBasketEngine

`ql.MCEuropeanBasketEngine(`*GeneralizedBlackScholesProcess, traits, timeSteps=None, timeStepsPerYear=None, brownianBridge=False, antitheticVariate=False, requiredSamples=None, requiredTolerance=None, maxSamples=None, seed=0*`)`

```
# Create a StochasticProcessArray for the various underlyings
underlying_spots = [100., 100., 100., 100., 100.]
underlying_vols = [0.1, 0.12, 0.13, 0.09, 0.11]
underlying_corr_mat = [[1, 0.1, -0.1, 0, 0], [0.1, 1, 0, 0, 0.2], [-0.1, 0, 1, 0, 0], [0, 0, 0, 1, 0.15],
↪ [0, 0.2, 0, 0.15, 1]]

today = ql.Date().todaysDate()
day_count = ql.Actual365Fixed()
calendar = ql.NullCalendar()

riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.0, day_count))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.0, day_count))

processes = [ql.BlackScholesMertonProcess(ql.QuoteHandle(ql.SimpleQuote(x)),
                                          dividendTS,
```

(continues on next page)

(continued from previous page)

```
                                                  riskFreeTS,
                                                  ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today,␣
↪calendar, y, day_count)))
              for x, y in zip(underlying_spots, underlying_vols)]

multiProcess = ql.StochasticProcessArray(processes, underlying_corr_mat)

# Create the pricing engine
rng = "pseudorandom"
numSteps = 500000
stepsPerYear = 1
seed = 43

engine = ql.MCEuropeanBasketEngine(multiProcess, rng, timeStepsPerYear=stepsPerYear,␣
↪requiredSamples=numSteps, seed=seed)
```

8.6.5 Cliquet Options

8.6.6 Forward Options

8.6.6.1 ForwardEuropeanEngine

This engine in python implements the C++ engine QuantLib::ForwardVanillaEngine (notice the subtle name change)

`ql.ForwardEuropeanEngine(`*process*`)`

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
volatility = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.NullCalendar(), 0.1, ql.
↪Actual365Fixed()))
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
process = ql.BlackScholesMertonProcess(initialValue, dividendTS, riskFreeTS, volatility)

engine = ql.ForwardEuropeanEngine(process)
```

8.6.6.2 MCForwardEuropeanBSEngine

`ql.MCForwardEuropeanBSEngine(`*process, traits, timeSteps=None, timeStepsPerYear=None, brownianBridge=False, antitheticVariate=False, requiredSamples=None, requiredTolerance=None, maxSamples=None, seed=0*`)`

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
volatility = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.NullCalendar(), 0.1, ql.
↪Actual365Fixed()))
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))

process = ql.BlackScholesMertonProcess(initialValue, dividendTS, riskFreeTS, volatility)

rng = "pseudorandom" # could use "lowdiscrepancy"
numPaths = 100000

engine = ql.MCForwardEuropeanBSEngine(process, rng, timeStepsPerYear=12, requiredSamples=numPaths)
```

8.6.6.3 AnalyticHestonForwardEuropeanEngine

`ql.AnalyticHestonForwardEuropeanEngine`(*process*, *integrationOrder=144*)

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))

v0, kappa, theta, rho, sigma = 0.005, 0.8, 0.008, 0.2, 0.2
hestonProcess = ql.HestonProcess(riskFreeTS, dividendTS, initialValue, v0, kappa, theta, sigma, rho)

engine = ql.AnalyticHestonForwardEuropeanEngine(hestonProcess)
```

8.6.6.4 MCForwardEuropeanHestonEngine

`ql.MCForwardEuropeanHestonEngine`(*hestonProcess*, *traits*, *timeSteps=None*, *timeStepsPerYear=None*, *antitheticVariate=False*, *requiredSamples=None*, *requiredTolerance=None*, *maxSamples=None*, *seed=0*, *controlVariate=False*)

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))

v0, kappa, theta, rho, sigma = 0.005, 0.8, 0.008, 0.2, 0.2
hestonProcess = ql.HestonProcess(riskFreeTS, dividendTS, initialValue, v0, kappa, theta, sigma, rho)

rng = "pseudorandom" # could use "lowdiscrepancy"
numPaths = 100000

engine = ql.MCForwardEuropeanHestonEngine(hestonProcess, rng, timeStepsPerYear=12, requiredSamples=numPaths)
```

8.6.7 Quanto Options

Chapter 9

Pricing Models

9.1 Equity

9.1.1 Heston

9.1.1.1 HestonModel

`ql.HestonModel(`*HestonProcess*`)`

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))

initialValue = ql.QuoteHandle(ql.SimpleQuote(100))

v0 = 0.005
theta = 0.010
kappa = 0.600
sigma = 0.400
rho = -0.15

hestonProcess = ql.HestonProcess(riskFreeTS, dividendTS, initialValue, v0, kappa, theta, sigma, rho)
hestonModel = ql.HestonModel(hestonProcess)
```

9.1.1.2 PiecewiseTimeDependentHestonModel

`ql.PiecewiseTimeDependentHestonModel(`*riskFreeRate, dividendYield, s0, v0, theta, kappa, sigma, rho, timeGrid*`)`

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))

initialValue = ql.QuoteHandle(ql.SimpleQuote(100))

times = [1.0, 2.0, 3.0]
grid = ql.TimeGrid(times)

v0 = 0.005
theta = [0.010, 0.015, 0.02]
kappa = [0.600, 0.500, 0.400]
sigma = [0.400, 0.350, 0.300]
```

(continues on next page)

(continued from previous page)

```
rho = [-0.15, -0.10, -0.00]

kappaTS = ql.PiecewiseConstantParameter(times[:-1], ql.PositiveConstraint())
thetaTS = ql.PiecewiseConstantParameter(times[:-1], ql.PositiveConstraint())
rhoTS = ql.PiecewiseConstantParameter(times[:-1], ql.BoundaryConstraint(-1.0, 1.0))
sigmaTS = ql.PiecewiseConstantParameter(times[:-1], ql.PositiveConstraint())

for i, time in enumerate(times):
    kappaTS.setParam(i, kappa[i])
    thetaTS.setParam(i, theta[i])
    rhoTS.setParam(i, rho[i])
    sigmaTS.setParam(i, sigma[i])

hestonModelPTD = ql.PiecewiseTimeDependentHestonModel(riskFreeTS, dividendTS, initialValue, v0, thetaTS,
↪kappaTS, sigmaTS, rhoTS, grid)
```

9.1.2 Bates

9.2 Short Rate Models

9.2.1 One Factor Models

- Vasicek: (r0=0.05, a=0.1, b=0.05, sigma=0.01, lambda=0.0)
- BlackKarasinski: (YieldTermStructure, a=0.1, sigma=0.1)
- HullWhite: (YieldTermStructure, a=0.1, sigma=0.01)
- Gsr()

9.2.1.1 Vasicek

`ql.Vasicek(`*r0=0.05*, *a=0.1*, *b=0.05*, *sigma=0.01*, *lambda=0.0*`)`

9.2.1.2 BlackKarasinski

`ql.BlackKarasinski(`*termStructure*, *a=0.1*, *sigma=0.1*`)`

9.2.1.3 HullWhite

`ql.HullWhite(`*termStructure*, *a=0.1*, *sigma=0.01*`)`

9.2.1.4 Gsr

One factor gsr model, formulation is in forward measure.

`ql.Gsr(`*termStrunctura*, *volstepdates*, *volatilities*, *reversions*`)`

9.2.2 Two Factor Models

9.2.2.1 G2

`ql.G2`(*termStructure*, *a=0.1*, *sigma=0.01*, *b=0.1*, *eta=0.01*, *rho=- 0.75*)

Chapter 10

Stochastic Processes

10.1 GeometricBrownianMotionProcess

`ql.GeometricBrownianMotionProcess(`*initialValue, mu, sigma*`)`

```
initialValue = 100
mu = 0.01
sigma = 0.2
process = ql.GeometricBrownianMotionProcess(initialValue, mu, sigma)
```

10.2 BlackScholesProcess

`ql.BlackScholesProcess(`*initialValue, riskFreeTS, volTS*`)`

```
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
sigma = 0.2
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
volTS = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.NullCalendar(), sigma, ql.
↪Actual365Fixed()))
process = ql.BlackScholesProcess(initialValue, riskFreeTS, volTS)
```

10.3 BlackScholesMertonProcess

`ql.BlackScholesMertonProcess(`*initialValue, dividendTS, riskFreeTS, volTS*`)`

```
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
sigma = 0.2
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
volTS = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.NullCalendar(), sigma, ql.
↪Actual365Fixed()))
process = ql.BlackScholesMertonProcess(initialValue, dividendTS, riskFreeTS, volTS)
```

10.4 GeneralizedBlackScholesProcess

`ql.GeneralizedBlackScholesProcess(`*initialValue, dividendTS, riskFreeTS, volTS*`)`

```
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
sigma = 0.2
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
volTS = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.NullCalendar(), sigma, ql.
↪Actual365Fixed()))
process = ql.GeneralizedBlackScholesProcess(initialValue, dividendTS, riskFreeTS, volTS)
```

10.5 ExtendedOrnsteinUhlenbeckProcess

`ql.ExtendedOrnsteinUhlenbeckProcess(`*speed, sigma, x0*`)`

```
x0 = 0.0
speed = 1.0
volatility = 0.1
process = ql.ExtendedOrnsteinUhlenbeckProcess(speed, volatility, x0, lambda x: x0)
```

10.6 ExtOUWithJumpsProcess

```
x0 = 0.0
x1 = 0.0
beta = 4.0
eta = 4.0
jumpIntensity = 1.0
speed = 1.0
volatility = 0.1
ouProcess = ql.ExtendedOrnsteinUhlenbeckProcess(speed, volatility, x0, lambda x: x0)
process = ql.ExtOUWithJumpsProcess(ouProcess, x1, beta, jumpIntensity, eta)
```

10.7 BlackProcess

`ql.BlackProcess(`*initialValue, riskFreeTS, volTS*`)`

```
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
sigma = 0.2
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
volTS = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.NullCalendar(), sigma, ql.
↪Actual365Fixed()))
process = ql.BlackProcess(initialValue, riskFreeTS, volTS)
```

10.8 Merton76Process

```
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
sigma = 0.2
today = ql.Date().todaysDate()
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))

riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
volTS = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.NullCalendar(), sigma, ql.
↪Actual365Fixed()))
process = ql.BlackProcess(initialValue, riskFreeTS, volTS)

jumpIntensity = ql.QuoteHandle(ql.SimpleQuote(1.0))
jumpVolatility = ql.QuoteHandle(ql.SimpleQuote(sigma * np.sqrt(0.25 / jumpIntensity.value())))
meanLogJump = ql.QuoteHandle(ql.SimpleQuote(-jumpVolatility.value()*jumpVolatility.value()))

process = ql.Merton76Process(initialValue, dividendTS, riskFreeTS, volTS, jumpIntensity, meanLogJump,␣
↪jumpVolatility)
```

10.9 VarianceGammaProcess

```
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))

sigma = 0.2
nu = 1
theta = 1
process = ql.VarianceGammaProcess(initialValue, dividendTS, riskFreeTS, sigma, nu, theta)
```

10.10 GarmanKohlagenProcess

`ql.GarmanKohlagenProcess(`*initialValue, foreignRiskFreeTS, domesticRiskFreeTS, volTS*`)`

```
initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
domesticRiskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.03, ql.Actual365Fixed()))
foreignRiskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))
volTS = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, ql.NullCalendar(), sigma, ql.
↪Actual365Fixed()))
process = ql.GarmanKohlagenProcess(initialValue, foreignRiskFreeTS, domesticRiskFreeTS, volTS)
```

10.11 HestonProcess

`ql.HestonProcess(`*riskFreeTS, dividendTS, initialValue, v0, kappa, theta, sigma, rho*`)`

```
today = ql.Date().todaysDate()
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))

initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
v0 = 0.005
kappa = 0.8
theta = 0.008
```

(continues on next page)

(continued from previous page)

```
rho = 0.2
sigma = 0.1

process = ql.HestonProcess(riskFreeTS, dividendTS, initialValue, v0, kappa, theta, sigma, rho)
```

10.12 HestonSLVProcess

ql.HestonSLVProcess(*hestonProcess*, *leverageFct*)

```
today = ql.Date().todaysDate()
endDate = today + ql.Period("2Y")

# Set up a Heston process
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.01, ql.Actual365Fixed()))

initialValue = ql.QuoteHandle(ql.SimpleQuote(100))
v0 = 0.005
kappa = 0.8
theta = 0.008
rho = 0.2
sigma = 0.1

hestonProcess = ql.HestonProcess(riskFreeTS, dividendTS, initialValue, v0, kappa, theta, sigma, rho)

# Set up a local vol surface
periods = [ql.Period("3M"), ql.Period("6M"), ql.Period("12M"), ql.Period("24M")]
expirationDates = [today + period for period in periods]
strikes = [90, 95, 100, 105, 110]

data = [[0.075, 0.076, 0.078, 0.080], [0.071, 0.072, 0.074, 0.078], [0.071, 0.072, 0.073, 0.077], [0.075,
↪ 0.075, 0.075, 0.077], [0.081, 0.080, 0.078, 0.078]]
impliedVols = ql.Matrix(data)

localVolSurface = ql.BlackVarianceSurface(today, ql.NullCalendar(), expirationDates, strikes,␣
↪impliedVols, ql.Actual365Fixed())
localVolHandle = ql.BlackVolTermStructureHandle(localVolSurface)
localVol = ql.LocalVolSurface(localVolHandle, riskFreeTS, dividendTS, initialValue)
localVol.enableExtrapolation()

# Calibrate Leverage Function to the Local Vol and Heston Model via Monte-Carlo
timeStepsPerYear = 365
nBins = 201
calibrationPaths = 2**15
mandatoryDates = []
mixingFactor = 0.9

generatorFactory = ql.MTBrownianGeneratorFactory()

hestonModel = ql.HestonModel(hestonProcess)
stochLocalMcModel = ql.HestonSLVMCModel(localVol, hestonModel, generatorFactory, endDate,␣
↪timeStepsPerYear, nBins, calibrationPaths, mandatoryDates, mixingFactor)
leverageFct = stochLocalMcModel.leverageFunction()

process = ql.HestonSLVProcess(hestonProcess, leverageFct, mixingFactor)
```

10.13 BatesProcess

10.14 HullWhiteProcess

ql.HullWhiteProcess(*riskFreeTS*, *a*, *sigma*)

```
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
a = 0.001
sigma = 0.1
process = ql.HullWhiteProcess(riskFreeTS, a, sigma)
```

10.15 HullWhiteForwardProcess

ql.HullWhiteForwardProcess(*riskFreeTS*, *a*, *sigma*)

```
riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.05, ql.Actual365Fixed()))
a = 0.001
sigma = 0.1
process = ql.HullWhiteForwardProcess(riskFreeTS, a, sigma)
```

10.16 GSR Process

```
today = ql.Date().todaysDate()
dates = [ql.TARGET().advance(today, ql.Period(i, ql.Days)) for i in range(1,10)]
times = [i for i in range(1,10)]
sigmas = [0.01 for i in range(0, len(dates)+1)]
reversion = 0.01
reversions = [reversion]
process = ql.GsrProcess(times, sigmas, reversions)
```

10.17 G2Process

10.18 G2ForwardProcess

10.19 Multiple Processes

ql.StochasticProcessArray(*processes*, *correlationMatrix*)

```
underlyingSpots = [100., 100., 100., 100., 100.]
underlyingVols = [0.1, 0.12, 0.13, 0.09, 0.11]
underlyingCorrMatrix = [[1, 0.1, -0.1, 0, 0], [0.1, 1, 0, 0, 0.2], [-0.1, 0, 1, 0, 0], [0, 0, 0, 1, 0.
↪15], [0, 0.2, 0, 0.15, 1]]

today = ql.Date().todaysDate()
dayCount = ql.Actual365Fixed()
calendar = ql.NullCalendar()

riskFreeTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.0, dayCount))
dividendTS = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.0, dayCount))
```

(continues on next page)

(continued from previous page)

```
processes = [ql.BlackScholesMertonProcess(ql.QuoteHandle(ql.SimpleQuote(x)),
                                          dividendTS,
                                          riskFreeTS,
                                          ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today,␣
↪calendar, y, dayCount)))
             for x, y in zip(underlyingSpots, underlyingVols)]

multiProcess = ql.StochasticProcessArray(processes, underlyingCorrMatrix)
```

Chapter 11

Term Structures

11.1 Yield Term Structures

11.1.1 FlatForward

Flat interest-rate curve.

`ql.FlatForward(`*date, quote, dayCounter, compounding, frequency*`)`

`ql.FlatForward(`*integer, Calendar, quote, dayCounter, compounding, frequency*`)`

`ql.FlatForward(`*integer, rate, dayCounter*`)`

Examples:

```
ql.FlatForward(ql.Date(15,6,2020), ql.QuoteHandle(ql.SimpleQuote(0.05)), ql.Actual360(), ql.Compounded,␣
↪ql.Annual)
ql.FlatForward(ql.Date(15,6,2020), ql.QuoteHandle(ql.SimpleQuote(0.05)), ql.Actual360(), ql.Compounded)
ql.FlatForward(ql.Date(15,6,2020), ql.QuoteHandle(ql.SimpleQuote(0.05)), ql.Actual360())
ql.FlatForward(2, ql.TARGET(), ql.QuoteHandle(ql.SimpleQuote(0.05)), ql.Actual360())
ql.FlatForward(2, ql.TARGET(), 0.05, ql.Actual360())
```

11.1.2 DiscountCurve

Term structure based on log-linear interpolation of discount factors.

`ql.DiscountCurve(`*dates, dfs, dayCounter, cal=ql.NullCalendar()*`)`

Example:

```
dates = [ql.Date(7,5,2019), ql.Date(7,5,2020), ql.Date(7,5,2021)]
dfs = [1, 0.99, 0.98]
dayCounter = ql.Actual360()
curve = ql.DiscountCurve(dates, dfs, dayCounter)
```

11.1.3 ZeroCurve

- ZeroCurve
- LogLinearZeroCurve
- CubicZeroCurve
- NaturalCubicZeroCurve
- LogCubicZeroCurve
- MonotonicCubicZeroCurve

`ql.ZeroCurve(`*dates*, *yields*, *dayCounter*, *cal*, *i*, *comp*, *freq*`)`

Dates	The date sequence, the maturity date corresponding to the zero interest rate. Note: The first date must be the base date of the curve, such as a date with a yield of 0.0.
yields	a sequence of floating point numbers, zero coupon yield
dayCounter	DayCounter object, number of days calculation rule
cal	Calendar object, calendar
i	Linear object, linear interpolation method
comp and freq	are preset integers indicating the way and frequency of payment

```
dates = [ql.Date(31,12,2019),  ql.Date(31,12,2020),  ql.Date(31,12,2021)]
zeros = [0.01, 0.02, 0.03]

ql.ZeroCurve(dates, zeros, ql.ActualActual(), ql.TARGET())
ql.LogLinearZeroCurve(dates, zeros, ql.ActualActual(), ql.TARGET())
ql.CubicZeroCurve(dates, zeros, ql.ActualActual(), ql.TARGET())
ql.NaturalCubicZeroCurve(dates, zeros, ql.ActualActual(), ql.TARGET())
ql.LogCubicZeroCurve(dates, zeros, ql.ActualActual(), ql.TARGET())
ql.MonotonicCubicZeroCurve(dates, zeros, ql.ActualActual(), ql.TARGET())
```

11.1.4 ForwardCurve

Term structure based on flat interpolation of forward rates.

`ql.ForwardCurve(`*dates*, *rates*, *dayCounter*`)`

`ql.ForwardCurve(`*dates*, *rates*, *dayCounter*, *calendar*, *BackwardFlat*`)`

`ql.ForwardCurve(`*dates*, *date*, *rates*, *rate*, *dayCounter*, *calendar*`)`

`ql.ForwardCurve(`*dates*, *date*, *rates*, *rate*, *dayCounter*`)`

```
dates = [ql.Date(15,6,2020), ql.Date(15,6,2022), ql.Date(15,6,2023)]
rates = [0.02, 0.03, 0.04]
ql.ForwardCurve(dates, rates, ql.Actual360(), ql.TARGET())
ql.ForwardCurve(dates, rates, ql.Actual360())
```

11.1.5 Piecewise

Piecewise yield term structure. This term structure is bootstrapped on a number of interest rate instruments which are passed as a vector of RateHelper instances. Their maturities mark the boundaries of the interpolated segments.

Each segment is determined sequentially starting from the earliest period to the latest and is chosen so that the instrument whose maturity marks the end of such segment is correctly repriced on the curve.

- PiecewiseLogLinearDiscount
- PiecewiseLogCubicDiscount
- PiecewiseLinearZero

- PiecewiseCubicZero
- PiecewiseLinearForward
- PiecewiseSplineCubicDiscount

`ql.Piecewise`(*referenceDate*, *helpers*, *dayCounter*)

```
helpers = []
helpers.append( ql.DepositRateHelper(0.05, ql.Euribor6M()) )
helpers.append(
    ql.SwapRateHelper(0.06, ql.EuriborSwapIsdaFixA(ql.Period('1y')))
)
curve = ql.PiecewiseLogLinearDiscount(ql.Date(15,6,2020), helpers, ql.Actual360())
```

`ql.PiecewiseYieldCurve`(*referenceDate*, *instruments*, *dayCounter*, *jumps*, *jumpDate*, *i=Interpolator()*, *bootstrap=bootstrap_type()*)

```
referenceDate = ql.Date(15,6,2020)
ql.PiecewiseLogLinearDiscount(referenceDate, helpers, ql.ActualActual())

jumps = [ql.QuoteHandle(ql.SimpleQuote(0.01))]
ql.PiecewiseLogLinearDiscount(referenceDate, helpers, ql.ActualActual(), jumps)

jumpDates = [ql.Date(15,9,2020)]
ql.PiecewiseLogLinearDiscount(referenceDate, helpers, ql.ActualActual(), jumps, jumpDates)
```

```
import pandas as pd
pgbs = pd.DataFrame(
    {'maturity': ['15-06-2020', '15-04-2021', '17-10-2022', '25-10-2023',
                  '15-02-2024', '15-10-2025', '21-07-2026', '14-04-2027',
                  '17-10-2028', '15-06-2029', '15-02-2030', '18-04-2034',
                  '15-04-2037', '15-02-2045'],
    'coupon': [4.8, 3.85, 2.2, 4.95,  5.65, 2.875, 2.875, 4.125,
                2.125, 1.95, 3.875, 2.25, 4.1, 4.1],
    'px': [102.532, 105.839, 107.247, 119.824, 124.005, 116.215, 117.708,
            128.027, 115.301, 114.261, 133.621, 119.879, 149.427, 159.177]})

calendar = ql.TARGET()
today = calendar.adjust(ql.Date(19, 12, 2019))
ql.Settings.instance().evaluationDate = today

bondSettlementDays = 2
bondSettlementDate = calendar.advance(
    today,
    ql.Period(bondSettlementDays, ql.Days))
frequency = ql.Annual
dc = ql.ActualActual(ql.ActualActual.ISMA)
accrualConvention = ql.ModifiedFollowing
convention = ql.ModifiedFollowing
redemption = 100.0

instruments = []
for idx, row in pgbs.iterrows():
    maturity = ql.Date(row.maturity, '%d-%m-%Y')
    schedule = ql.Schedule(
        bondSettlementDate,
        maturity,
        ql.Period(frequency),
        calendar,
        accrualConvention,
        accrualConvention,
        ql.DateGeneration.Backward,
        False)
```

(continues on next page)

(continued from previous page)

```
    helper = ql.FixedRateBondHelper(
            ql.QuoteHandle(ql.SimpleQuote(row.px)),
            bondSettlementDays,
            100.0,
            schedule,
            [row.coupon / 100],
            dc,
            convention,
            redemption)

    instruments.append(helper)

params = [bondSettlementDate, instruments, dc]

piecewiseMethods = {
    'logLinearDiscount': ql.PiecewiseLogLinearDiscount(*params),
    'logCubicDiscount': ql.PiecewiseLogCubicDiscount(*params),
    'linearZero': ql.PiecewiseLinearZero(*params),
    'cubicZero': ql.PiecewiseCubicZero(*params),
    'linearForward': ql.PiecewiseLinearForward(*params),
    'splineCubicDiscount': ql.PiecewiseSplineCubicDiscount(*params),
}
```

11.1.6 ImpliedTermStructure

Implied term structure at a given date in the future

ql.ImpliedTermStructure(*YieldTermStructure*, *date*)

```
crv = ql.FlatForward(ql.Date(10,1,2020),0.04875825,ql.Actual365Fixed())
yts = ql.YieldTermStructureHandle(crv)
ql.ImpliedTermStructure(yts, ql.Date(20,9,2020))
```

11.1.7 ForwardSpreadedTermStructure

Term structure with added spread on the instantaneous forward rate.

ql.ForwardSpreadedTermStructure(*YieldTermStructure*, *spread*)

```
crv = ql.FlatForward(ql.Date(10,1,2020),0.04875825,ql.Actual365Fixed())
yts = ql.YieldTermStructureHandle(crv)
spread = ql.QuoteHandle(ql.SimpleQuote(0.005))
ql.ForwardSpreadedTermStructure(yts, spread)
```

11.1.8 ZeroSpreadedTermStructure

Term structure with an added spread on the zero yield rate

ql.ZeroSpreadedTermStructure(*YieldTermStructure*, *spread*)

```
crv = ql.FlatForward(ql.Date(10,1,2020),0.04875825,ql.Actual365Fixed())
yts = ql.YieldTermStructureHandle(crv)
spread = ql.QuoteHandle(ql.SimpleQuote(0.005))
ql.ZeroSpreadedTermStructure(yts, spread)
```

11.1.9 SpreadedLinearZeroInterpolatedTermStructure

`ql.SpreadedLinearZeroInterpolatedTermStructure(`*YieldTermStructure, quotes, dates, compounding, frequency, dayCounter, linear*`)`

```
crv = ql.FlatForward(settlement,0.04875825,ql.Actual365Fixed())
yts = ql.YieldTermStructureHandle(crv)

calendar = ql.TARGET()
spread21 = ql.SimpleQuote(0.0050)
spread22 = ql.SimpleQuote(0.0050)
startDate = ql.Date().todaysDate()
endDate = calendar.advance(startDate, ql.Period(50, ql.Years))

tsSpread = ql.SpreadedLinearZeroInterpolatedTermStructure(
    yts,
    [ql.QuoteHandle(spread21), ql.QuoteHandle(spread22)],
    [startDate, endDate]
)
```

11.1.10 FittedBondCurve

`ql.FittedBondDiscountCurve(`*bondSettlementDate, helpers, dc, method, accuracy=1e-10, maxEvaluations=10000, guess=Array(), simplexLambda=1.0*`)`

Methods:

- CubicBSplinesFitting
- ExponentialSplinesFitting
- NelsonSiegelFitting
- SimplePolynomialFitting
- SvenssonFitting

```
pgbs = pd.DataFrame(
    {'maturity': ['15-06-2020', '15-04-2021', '17-10-2022', '25-10-2023',
                  '15-02-2024', '15-10-2025', '21-07-2026', '14-04-2027',
                  '17-10-2028', '15-06-2029', '15-02-2030', '18-04-2034',
                  '15-04-2037', '15-02-2045'],
    'coupon': [4.8, 3.85, 2.2, 4.95,  5.65, 2.875, 2.875, 4.125,
                2.125, 1.95, 3.875, 2.25, 4.1, 4.1],
    'px': [102.532, 105.839, 107.247, 119.824, 124.005, 116.215, 117.708,
            128.027, 115.301, 114.261, 133.621, 119.879, 149.427, 159.177]})

calendar = ql.TARGET()
today = calendar.adjust(ql.Date(19, 12, 2019))
ql.Settings.instance().evaluationDate = today

bondSettlementDays = 2
bondSettlementDate = calendar.advance(
    today,
    ql.Period(bondSettlementDays, ql.Days))
frequency = ql.Annual
dc = ql.ActualActual(ql.ActualActual.ISMA)
accrualConvention = ql.ModifiedFollowing
convention = ql.ModifiedFollowing
redemption = 100.0

instruments = []
for idx, row in pgbs.iterrows():
    maturity = ql.Date(row.maturity, '%d-%m-%Y')
    schedule = ql.Schedule(
```

(continues on next page)

(continued from previous page)

```
        bondSettlementDate,
        maturity,
        ql.Period(frequency),
        calendar,
        accrualConvention,
        accrualConvention,
        ql.DateGeneration.Backward,
        False)
    helper = ql.FixedRateBondHelper(
            ql.QuoteHandle(ql.SimpleQuote(row.px)),
            bondSettlementDays,
            100.0,
            schedule,
            [row.coupon / 100],
            dc,
            convention,
            redemption)

    instruments.append(helper)

params = [bondSettlementDate, instruments, dc]

cubicNots = [-30.0, -20.0, 0.0, 5.0, 10.0, 15.0,20.0, 25.0, 30.0, 40.0, 50.0]
fittingMethods = {
    'NelsonSiegelFitting': ql.NelsonSiegelFitting(),
    'SvenssonFitting': ql.SvenssonFitting(),
    'SimplePolynomialFitting': ql.SimplePolynomialFitting(2),
    'ExponentialSplinesFitting': ql.ExponentialSplinesFitting(),
    'CubicBSplinesFitting': ql.CubicBSplinesFitting(cubicNots),
}

fittedBondCurveMethods = {
    label: ql.FittedBondDiscountCurve(*params, method)
    for label, method in fittingMethods.items()
}

curve = fittedBondCurveMethods.get('NelsonSiegelFitting')
```

11.1.11 FXImpliedCurve

11.2 Volatility

11.2.1 BlackConstantVol

`ql.BlackConstantVol(`*date, calendar, volatility, dayCounter*`)`

`ql.BlackConstantVol(`*date, calendar, volatilityHandle, dayCounter*`)`

`ql.BlackConstantVol(`*days, calendar, volatility, dayCounter*`)`

`ql.BlackConstantVol(`*days, calendar, volatilityHandle, dayCounter*`)`

```
date = ql.Date().todaysDate()
settlementDays = 2
calendar = ql.TARGET()
volatility = 0.2
volHandle = ql.QuoteHandle(ql.SimpleQuote(volatility))
```

(continues on next page)

(continued from previous page)

```
dayCounter = ql.Actual360()

ql.BlackConstantVol(date, calendar, volatility, dayCounter)
ql.BlackConstantVol(date, calendar, volHandle, dayCounter)
ql.BlackConstantVol(settlementDays, calendar, volatility, dayCounter)
ql.BlackConstantVol(settlementDays, calendar, volHandle, dayCounter)
```

11.2.2 BlackVarianceCurve

`ql.BlackVarianceCurve`(*referenceDate*, *expirations*, *volatilities*, *dayCounter*)

```
referenceDate = ql.Date(30, 9, 2013)
expirations = [ql.Date(20, 12, 2013), ql.Date(17, 1, 2014), ql.Date(21, 3, 2014)]
volatilities = [.145, .156, .165]
volatilityCurve = ql.BlackVarianceCurve(referenceDate, expirations, volatilities, ql.Actual360())
```

11.2.3 BlackVarianceSurface

`ql.BlackVarianceSurface`(*referenceDate*, *calendar*, *expirations*, *strikes*, *volMatrix*, *dayCounter*)

```
referenceDate = ql.Date(30, 9, 2013)
ql.Settings.instance().evaluationDate = referenceDate;
calendar = ql.TARGET()
dayCounter = ql.ActualActual()

strikes = [1650.0, 1660.0, 1670.0]
expirations = [ql.Date(20, 12, 2013), ql.Date(17, 1, 2014), ql.Date(21, 3, 2014)]

volMatrix = ql.Matrix(len(strikes), len(expirations))

#1650 - Dec, Jan, Mar
volMatrix[0][0] = .15640; volMatrix[0][1] = .15433; volMatrix[0][2] = .16079;
#1660 - Dec, Jan, Mar
volMatrix[1][0] = .15343; volMatrix[1][1] = .15240; volMatrix[1][2] = .15804;
#1670 - Dec, Jan, Mar
volMatrix[2][0] = .15128; volMatrix[2][1] = .14888; volMatrix[2][2] = .15512;

volatilitySurface = ql.BlackVarianceSurface(
    referenceDate,
    calendar,
    expirations,
    strikes,
    volMatrix,
    dayCounter
)
volatilitySurface.enableExtrapolation()
```

11.2.4 HestonBlackVolSurface

`ql.HestonBlackVolSurface(`*hestonModelHandle*`)`

```
flatTs = ql.YieldTermStructureHandle(
  ql.FlatForward(ql.Date().todaysDate(), 0.05, ql.Actual365Fixed())
)
dividendTs = ql.YieldTermStructureHandle(
  ql.FlatForward(ql.Date().todaysDate(), 0.02, ql.Actual365Fixed())
)

v0 = 0.01; kappa = 0.01; theta = 0.01; rho = 0.0; sigma = 0.01
spot = 100
process = ql.HestonProcess(flatTs, dividendTs,
                           ql.QuoteHandle(ql.SimpleQuote(spot)),
                           v0, kappa, theta, sigma, rho
                           )

hestonModel = ql.HestonModel(process)
hestonHandle = ql.HestonModelHandle(hestonModel)
hestonVolSurface = ql.HestonBlackVolSurface(hestonHandle)
```

11.2.5 AndreasenHugeVolatilityAdapter

An implementation of the arb-free Andreasen-Huge vol interpolation described in "Andreasen J., Huge B., 2010. Volatility Interpolation" (https://ssrn.com/abstract=1694972). An advantage of this method is that it can take a non-rectangular grid of option quotes.

`ql.AndreasenHugeVolatilityAdapter(`*AndreasenHugeVolatilityInterpl*`)`

```
today = ql.Date().todaysDate()
calendar = ql.NullCalendar()
dayCounter = ql.Actual365Fixed()
spot = 100
r, q = 0.02, 0.05

spotQuote = ql.QuoteHandle(ql.SimpleQuote(spot))
ratesTs = ql.YieldTermStructureHandle(ql.FlatForward(today, r, dayCounter))
dividendTs = ql.YieldTermStructureHandle(ql.FlatForward(today, q, dayCounter))

# Market options price quotes
optionStrikes = [95, 97.5, 100, 102.5, 105, 90, 95, 100, 105, 110, 80, 90, 100, 110, 120]
optionMaturities = ["3M", "3M", "3M", "3M", "3M", "6M", "6M", "6M", "6M", "6M", "1Y", "1Y", "1Y", "1Y",
↪"1Y"]
optionQuotedVols = [0.11, 0.105, 0.1, 0.095, 0.095, 0.12, 0.11, 0.105, 0.1, 0.105, 0.12, 0.115, 0.11, 0.
↪11, 0.115]

calibrationSet = ql.CalibrationSet()

for strike, expiry, impliedVol in zip(optionStrikes, optionMaturities, optionQuotedVols):
  payoff = ql.PlainVanillaPayoff(ql.Option.Call, strike)
  exercise = ql.EuropeanExercise(calendar.advance(today, ql.Period(expiry)))

  calibrationSet.push_back((ql.VanillaOption(payoff, exercise), ql.SimpleQuote(impliedVol)))

ahInterpolation = ql.AndreasenHugeVolatilityInterpl(calibrationSet, spotQuote, ratesTs, dividendTs)
ahSurface = ql.AndreasenHugeVolatilityAdapter(ahInterpolation)
```

11.2.6 BlackVolTermStructureHandle

ql.BlackVolTermStructureHandle(*blackVolTermStructure*)

```
ql.BlackVolTermStructureHandle(constantVol)
ql.BlackVolTermStructureHandle(volatilityCurve)
ql.BlackVolTermStructureHandle(volatilitySurface)
```

11.2.7 RelinkableBlackVolTermStructureHandle

ql.RelinkableBlackVolTermStructureHandle()

ql.RelinkableBlackVolTermStructureHandle(*blackVolTermStructure*)

```
blackTSHandle = ql.RelinkableBlackVolTermStructureHandle(volatilitySurface)

blackTSHandle = ql.RelinkableBlackVolTermStructureHandle()
blackTSHandle.linkTo(volatilitySurface)
```

11.2.8 LocalConstantVol

ql.LocalConstantVol(*date*, *volatility*, *dayCounter*)

```
date = ql.Date().todaysDate()
volatility = 0.2
dayCounter = ql.Actual360()

ql.LocalConstantVol(date, volatility, dayCounter)
```

11.2.9 LocalVolSurface

ql.LocalVolSurface(*blackVolTs*, *ratesTs*, *dividendsTs*, *spot*)

```
today = ql.Date().todaysDate()
calendar = ql.NullCalendar()
dayCounter = ql.Actual365Fixed()
volatility = 0.2
r, q = 0.02, 0.05

blackVolTs = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, calendar, volatility, dayCounter))
ratesTs = ql.YieldTermStructureHandle(ql.FlatForward(today, r, dayCounter))
dividendTs = ql.YieldTermStructureHandle(ql.FlatForward(today, q, dayCounter))
spot = 100

ql.LocalVolSurface(blackVolTs, ratesTs, dividendTs, spot)
```

11.2.10 NoExceptLocalVolSurface

This powerful but dangerous surface will swallow any exceptions and return the specified override value when they occur. If your vol surface is well-calibrated, this protects you from crashes due to very far illiquid points on the local vol surface. But if your vol surface is not good, it could suppress genuine errors. Caution recommended.

`ql.NoExceptLocalVolSurface(`*blackVolTs*, *ratesTs*, *dividendsTs*, *spot*, *illegalVolOverride*`)`

```
today = ql.Date().todaysDate()
calendar = ql.NullCalendar()
dayCounter = ql.Actual365Fixed()
r, q = 0.02, 0.05
volatility = 0.2
illegalVolOverride = 0.25

blackVolTs = ql.BlackVolTermStructureHandle(ql.BlackConstantVol(today, calendar, volatility, dayCounter))
ratesTs = ql.YieldTermStructureHandle(ql.FlatForward(today, r, dayCounter))
dividendTs = ql.YieldTermStructureHandle(ql.FlatForward(today, q, dayCounter))
spot = 100

ql.NoExceptLocalVolSurface(blackVolTs, ratesTs, dividendTs, spot, illegalVolOverride)
```

11.2.11 AndreasenHugeLocalVolAdapter

`ql.AndreasenHugeLocalVolAdapter(`*AndreasenHugeVolatilityInterpl*`)`

```
today = ql.Date().todaysDate()
calendar = ql.NullCalendar()
dayCounter = ql.Actual365Fixed()
spot = 100
r, q = 0.02, 0.05

spotQuote = ql.QuoteHandle(ql.SimpleQuote(spot))
ratesTs = ql.YieldTermStructureHandle(ql.FlatForward(today, r, dayCounter))
dividendTs = ql.YieldTermStructureHandle(ql.FlatForward(today, q, dayCounter))

# Market options price quotes
optionStrikes = [95, 97.5, 100, 102.5, 105, 90, 95, 100, 105, 110, 80, 90, 100, 110, 120]
optionMaturities = ["3M", "3M", "3M", "3M", "3M", "6M", "6M", "6M", "6M", "6M", "1Y", "1Y", "1Y", "1Y",
↪"1Y"]
optionQuotedVols = [0.11, 0.105, 0.1, 0.095, 0.095, 0.12, 0.11, 0.105, 0.1, 0.105, 0.12, 0.115, 0.11, 0.
↪11, 0.115]

calibrationSet = ql.CalibrationSet()

for strike, expiry, impliedVol in zip(optionStrikes, optionMaturities, optionQuotedVols):
  payoff = ql.PlainVanillaPayoff(ql.Option.Call, strike)
  exercise = ql.EuropeanExercise(calendar.advance(today, ql.Period(expiry)))

  calibrationSet.push_back((ql.VanillaOption(payoff, exercise), ql.SimpleQuote(impliedVol)))

ahInterpolation = ql.AndreasenHugeVolatilityInterpl(calibrationSet, spotQuote, ratesTs, dividendTs)
ahLocalSurface = ql.AndreasenHugeLocalVolAdapter(ahInterpolation)
```

11.2.12 LocalVolTermStructureHandle

11.3 Cap Volatility

11.3.1 ConstantOptionletVolatility

floating reference date, floating market data

`ql.ConstantOptionletVolatility(`*settlementDays, cal, bdc, volatility (Quote), dc, type=ShiftedLognormal, displacement=0.0*`)`

fixed reference date, floating market data

`ql.ConstantOptionletVolatility(`*settlementDate, cal, bdc, volatility (Quote), dc, type=ShiftedLognormal, displacement=0.0*`)`

floating reference date, fixed market data

`ql.ConstantOptionletVolatility(`*settlementDays, cal, bdc, volatility (value), dc, type=ShiftedLognormal, displacement=0.0*`)`

fixed reference date, fixed market data

`ql.ConstantOptionletVolatility(`*settlementDate, cal, bdc, volatility (value), dc, type=ShiftedLognormal, displacement=0.0*`)`

```
settlementDays = 2
settlementDate = ql.Date().todaysDate()
cal = ql.TARGET()
bdc = ql.ModifiedFollowing
volatility = 0.55
vol_quote = ql.QuoteHandle(ql.SimpleQuote(volatility))
dc = ql.Actual365Fixed()

#floating reference date, floating market data
c1 = ql.ConstantOptionletVolatility(settlementDays, cal, bdc, vol_quote, dc, ql.Normal)

#fixed reference date, floating market data
c2 = ql.ConstantOptionletVolatility(settlementDate, cal, bdc, vol_quote, dc)

#floating reference date, fixed market data
c3 = ql.ConstantOptionletVolatility(settlementDays, cal, bdc, volatility, dc)

#fixed reference date, fixed market data
c4 = ql.ConstantOptionletVolatility(settlementDate, cal, bdc, volatility, dc)
```

11.3.2 CapFloorTermVolCurve

Cap/floor at-the-money term-volatility vector.

floating reference date, floating market data

`ql.CapFloorTermVolCurve(`*settlementDays, calendar, bdc, optionTenors, vols (Quotes), dc=Actual365Fixed*`)`

fixed reference date, floating market data

`ql.CapFloorTermVolCurve(`*settlementDate, calendar, bdc, optionTenors, vols (Quotes), dc=Actual365Fixed*`)`

fixed reference date, fixed market data

`ql.CapFloorTermVolCurve(`*settlementDate, calendar, bdc, optionTenors, vols (vector), dc=Actual365Fixed*`)`

floating reference date, fixed market data

`ql.CapFloorTermVolCurve(`*settlementDays, calendar, bdc, optionTenors, vols (vector), dc=Actual365Fixed*`)`

```
settlementDate = ql.Date().todaysDate()
settlementDays = 2
calendar = ql.TARGET()
bdc = ql.ModifiedFollowing
optionTenors  = [ql.Period('1y'), ql.Period('2y'), ql.Period('3y')]
vols = [0.55, 0.60, 0.65]

# fixed reference date, fixed market data
c3 = ql.CapFloorTermVolCurve(settlementDate, calendar, bdc, optionTenors, vols)

# floating reference date, fixed market data
c4 = ql.CapFloorTermVolCurve(settlementDays, calendar, bdc, optionTenors, vols)
```

11.3.3 CapFloorTermVolSurface

floating reference date, floating market data

`ql.CapFloorTermVolSurface(`*settlementDays, calendar, bdc, expiries, strikes, vol_data (Handle), daycount=ql.Actual365Fixed*`)`

fixed reference date, floating market data

`ql.CapFloorTermVolSurface(`*settlementDate, calendar, bdc, expiries, strikes, vol_data (Handle), daycount=ql.Actual365Fixed*`)`

fixed reference date, fixed market data

`ql.CapFloorTermVolSurface(`*settlementDate, calendar, bdc, expiries, strikes, vol_data (Matrix), daycount=ql.Actual365Fixed*`)`

floating reference date, fixed market data

`ql.CapFloorTermVolSurface(`*settlementDays, calendar, bdc, expiries, strikes, vol_data (Matrix), daycount=ql.Actual365Fixed*`)`

```
settlementDate = ql.Date().todaysDate()
settlementDays = 2
calendar = ql.TARGET()
bdc = ql.ModifiedFollowing
expiries  = [ql.Period('9y'), ql.Period('10y'), ql.Period('12y')]
strikes = [0.015, 0.02, 0.025]

black_vols = [
    [1.    , 0.792 , 0.6873],
    [0.9301, 0.7401, 0.6403],
    [0.7926, 0.6424, 0.5602]]


# fixed reference date, fixed market data
s3 = ql.CapFloorTermVolSurface(settlementDate, calendar, bdc, expiries, strikes, black_vols)

# floating reference date, fixed market data
s4 = ql.CapFloorTermVolSurface(settlementDays, calendar, bdc, expiries, strikes, black_vols)
```

11.3.4 OptionletStripper1

`ql.OptionletStripper1(`*CapFloorTermVolSurface, index, switchStrikes=Null, accuracy=1e-06, maxIter=100, discount=YieldTermStructure, type=ShiftedLognormal, displacement=0.0, dontThrow=false*`)`

```
index = ql.Euribor6M()
optionlet_surf = ql.OptionletStripper1(s3, index, type=ql.Normal)
```

11.3.5 StrippedOptionletAdapter

`ql.StrippedOptionletAdapter(`*StrippedOptionletBase*`)`

11.3.6 OptionletVolatilityStructureHandle

`ql.OptionletVolatilityStructureHandle(`*OptionletVolatilityStructure*`)`

```
ovs_handle = ql.OptionletVolatilityStructureHandle(
    ql.StrippedOptionletAdapter(optionlet_surf)
)
```

11.3.7 RelinkableOptionletVolatilityStructureHandle

`ql.RelinkableOptionletVolatilityStructureHandle()`

```
ovs_handle = ql.RelinkableOptionletVolatilityStructureHandle()
ovs_handle.linkTo(ql.StrippedOptionletAdapter(optionlet_surf))
```

11.4 Swaption Volatility

11.4.1 ConstantSwaptionVolatility

Constant swaption volatility, no time-strike dependence.

floating reference date, floating market data

`ql.ConstantSwaptionVolatility(`*settlementDays, cal, bdc, volatility, dc, type=ql.ShiftedLognormal, shift=0.0*`)`

fixed reference date, floating market data

`ql.ConstantSwaptionVolatility(`*settlementDate, cal, bdc, volatility, dc, type=ql.ShiftedLognormal, shift=0.0*`)`

floating reference date, fixed market data

`ql.ConstantSwaptionVolatility(`*settlementDays, cal, bdc, volatilityQuote, dc, type=ql.ShiftedLognormal, shift=0.0*`)`

fixed reference date, fixed market data

`ql.ConstantSwaptionVolatility(`*settlementDate, cal, bdc, volatilityQuote, dc, type=ql.ShiftedLognormal, shift=0.0*`)`

```
constantSwaptionVol = ql.ConstantSwaptionVolatility(2, ql.TARGET(), ql.ModifiedFollowing, ql.
↪QuoteHandle(ql.SimpleQuote(0.55)), ql.ActualActual())
```

11.4.2 SwaptionVolatilityMatrix

At-the-money swaption-volatility matrix.

floating reference date, floating market data

`ql.SwaptionVolatilityMatrix(`*calendar, bdc, optionTenors, swapTenors, vols (Handles), dayCounter, flatExtrapolation=false, type=ShiftedLognormal, shifts (vector)*`)`

fixed reference date, floating market data

`ql.SwaptionVolatilityMatrix(`*referenceDate, calendar, bdc, optionTenors, swapTenors, vols (Handles), dayCounter, flatExtrapolation=false, type=ShiftedLognormal, shifts (vector)*`)`

floating reference date, fixed market data

`ql.SwaptionVolatilityMatrix(`*calendar, bdc, optionTenors, swapTenors, vols (matrix), dayCounter, flatExtrapolation=false, type=ShiftedLognormal, shifts (matrix)*`)`

fixed reference date, fixed market data

`ql.SwaptionVolatilityMatrix(`*referenceDate, calendar, bdc, optionTenors, swapTenors, vols (matrix), dayCounter, flatExtrapolation=false, type=ShiftedLognormal, shifts (matrix)*`)`

fixed reference date and fixed market data, option dates

`ql.SwaptionVolatilityMatrix(`*referenceDate, calendar, bdc, optionDates, swapTenors, vols (matrix), dayCounter, flatExtrapolation=false, type=ShiftedLognormal, shifts (matrix)*`)`

```
# market Data 07.01.2020

swapTenors = [
    '1Y', '2Y', '3Y', '4Y', '5Y',
    '6Y', '7Y', '8Y', '9Y', '10Y',
    '15Y', '20Y', '25Y', '30Y']

optionTenors = [
    '1M', '2M', '3M', '6M', '9M', '1Y',
    '18M', '2Y', '3Y', '4Y', '5Y', '7Y',
    '10Y', '15Y', '20Y', '25Y', '30Y']

normal_vols = [
    [8.6, 12.8, 19.5, 26.9, 32.7, 36.1, 38.7, 40.9, 42.7, 44.3, 48.8, 50.4, 50.8, 50.4],
    [9.2, 13.4, 19.7, 26.4, 31.9, 35.2, 38.3, 40.2, 41.9, 43.1, 47.8, 49.9, 50.7, 50.3],
    [11.2, 15.3, 21.0, 27.6, 32.7, 35.3, 38.4, 40.8, 42.6, 44.5, 48.6, 50.5, 50.9, 51.0],
    [12.9, 17.1, 22.6, 28.8, 33.5, 36.0, 38.8, 41.0, 43.0, 44.6, 48.7, 50.6, 51.1, 51.0],
    [14.6, 18.7, 24.6, 30.1, 34.2, 36.9, 39.3, 41.3, 43.2, 44.9, 48.9, 51.0, 51.3, 51.5],
    [16.5, 20.9, 26.3, 31.3, 35.0, 37.6, 40.0, 42.0, 43.7, 45.3, 48.8, 50.9, 51.4, 51.7],
    [20.9, 25.3, 30.0, 34.0, 37.0, 39.5, 41.9, 43.4, 45.0, 46.4, 49.3, 51.0, 51.3, 51.9],
    [25.1, 28.9, 33.2, 36.2, 39.2, 41.2, 43.2, 44.7, 46.0, 47.3, 49.6, 51.0, 51.3, 51.6],
    [34.0, 36.6, 39.2, 41.1, 43.2, 44.5, 46.1, 47.2, 48.0, 49.0, 50.3, 51.3, 51.3, 51.2],
    [40.3, 41.8, 43.6, 44.9, 46.1, 47.1, 48.2, 49.2, 49.9, 50.5, 51.2, 51.3, 50.9, 50.7],
    [44.0, 44.8, 46.0, 47.1, 48.4, 49.1, 49.9, 50.7, 51.4, 51.9, 51.6, 51.4, 50.6, 50.2],
    [49.6, 49.7, 50.4, 51.2, 51.8, 52.2, 52.6, 52.9, 53.3, 53.8, 52.6, 51.7, 50.4, 49.6],
    [53.9, 53.7, 54.0, 54.2, 54.4, 54.5, 54.5, 54.4, 54.4, 54.9, 53.1, 51.8, 50.1, 49.1],
    [54.0, 53.7, 53.8, 53.7, 53.5, 53.6, 53.5, 53.3, 53.5, 53.7, 51.4, 49.8, 47.9, 46.6],
    [52.8, 52.4, 52.6, 52.3, 52.2, 52.3, 52.0, 51.9, 51.8, 51.8, 49.5, 47.4, 45.4, 43.8],
    [51.4, 51.2, 51.3, 51.0, 50.8, 50.7, 50.3, 49.9, 49.8, 49.7, 47.6, 45.3, 43.1, 41.4],
    [49.6, 49.6, 49.7, 49.5, 49.5, 49.2, 48.6, 47.9, 47.4, 47.1, 45.1, 42.9, 40.8, 39.2]
]

swapTenors = [ql.Period(tenor) for tenor in swapTenors]
optionTenors = [ql.Period(tenor) for tenor in optionTenors]
normal_vols = [[vol / 10000 for vol in row] for row in normal_vols]
```

(continues on next page)

(continued from previous page)

```
calendar = ql.TARGET()
bdc = ql.ModifiedFollowing
dayCounter = ql.ActualActual()
swaptionVolMatrix = ql.SwaptionVolatilityMatrix(
    calendar, bdc,
    optionTenors, swapTenors, ql.Matrix(normal_vols),
    dayCounter, False, ql.Normal)
```

11.4.3 SwaptionVolCube1

11.4.4 SwaptionVolCube2

`ql.SwaptionVolCube2(`*atmVolStructure, optionTenors, swapTenors, strikeSpreads, volSpreads, swapIndex, shortSwapIndex, vegaWeightedSmileFit*`)`

```
optionTenors = ['1y', '2y', '3y']
swapTenors = [ '5Y', '10Y']
strikeSpreads = [ -0.01, 0.0, 0.01]
volSpreads = [
    [0.5, 0.55, 0.6],
    [0.5, 0.55, 0.6],
    [0.5, 0.55, 0.6],
    [0.5, 0.55, 0.6],
    [0.5, 0.55, 0.6],
    [0.5, 0.55, 0.6],
]


optionTenors = [ql.Period(tenor) for tenor in optionTenors]
swapTenors = [ql.Period(tenor) for tenor in swapTenors]
volSpreads = [[ql.QuoteHandle(ql.SimpleQuote(v)) for v in row] for row in volSpreads]

swapIndexBase = ql.EuriborSwapIsdaFixA(ql.Period(1, ql.Years), e6m_yts, ois_yts)
shortSwapIndexBase = ql.EuriborSwapIsdaFixA(ql.Period(1, ql.Years), e6m_yts, ois_yts)
vegaWeightedSmileFit = False

volCube = ql.SwaptionVolatilityStructureHandle(
    ql.SwaptionVolCube2(
        ql.SwaptionVolatilityStructureHandle(swaptionVolMatrix),
        optionTenors,
        swapTenors,
        strikeSpreads,
        volSpreads,
        swapIndexBase,
        shortSwapIndexBase,
        vegaWeightedSmileFit)
)
volCube.enableExtrapolation()
```

11.4.5 SwaptionVolatilityStructureHandle

ql.SwaptionVolatilityStructureHandle(*swaptionVolStructure*)

```
swaptionVolHandle = ql.SwaptionVolatilityStructureHandle(swaptionVolMatrix)
```

11.4.6 RelinkableSwaptionVolatilityStructureHandle

ql.RelinkableSwaptionVolatilityStructureHandle()

```
handle = ql.RelinkableSwaptionVolatilityStructureHandle()
handle.linkTo(swaptionVolMatrix)
```

11.5 SABR

11.5.1 SabrSmileSection

ql.SabrSmileSection(*date*, *fwd*[, *alpha*, *beta*, *nu*, *rho*], *dayCounter*, *Real*)

ql.SabrSmileSection(*time*, *fwd*[, *alpha*, *beta*, *nu*, *rho*], *dayCounter*, *Real*)

```
alpha = 1.63
beta = 0.6
nu = 3.3
rho = 0.00002

ql.SabrSmileSection(17/365, 120, [alpha, beta, nu, rho])
```

11.5.2 sabrVolatility

ql.sabrVolatility(*strike*, *forward*, *expiryTime*, *alpha*, *beta*, *nu*, *rho*)

```
alpha = 1.63
beta = 0.6
nu = 3.3
rho = 0.00002
ql.sabrVolatility(106, 120, 17/365, alpha, beta, nu, rho)
```

11.5.3 shiftedSabrVolatility

ql.shiftedSabrVolatility(*strike*, *forward*, *expiryTime*, *alpha*, *beta*, *nu*, *rho*, *shift*)

```
alpha = 1.63
beta = 0.6
nu = 3.3
rho = 0.00002
shift = 50

ql.shiftedSabrVolatility(106, 120, 17/365, alpha, beta, nu, rho, shift)
```

11.5.4 sabrFlochKennedyVolatility

ql.sabrFlochKennedyVolatility(*strike*, *forward*, *expiryTime*, *alpha*, *beta*, *nu*, *rho*)

```
alpha = 0.01
beta = 0.01
nu = 0.01
rho = 0.01

ql.sabrFlochKennedyVolatility(0.01,0.01, 5, alpha, beta, nu, rho)
```

11.6 Credit Term Structures

11.6.1 FlatHazardRate

Flat hazard-rate curve.

ql.FlatHazardRate(*settlementDays*, *calendar*, *Quote*, *dayCounter*)

ql.FlatHazardRate(*settelementDate*, *Quote*, *dayCounter*)

```
pd_curve = ql.FlatHazardRate(2, ql.TARGET(), ql.QuoteHandle(ql.SimpleQuote(0.05)), ql.Actual360())
pd_curve = ql.FlatHazardRate(ql.Date().todaysDate(), ql.QuoteHandle(ql.SimpleQuote(0.05)), ql.
↪Actual360())
```

11.6.2 PiecewiseFlatHazardRate

Piecewise default-probability term structure.

ql.PiecewiseFlatHazardRate(*settlementDate*, *helpers*, *dayCounter*)

```
recoveryRate = 0.4
settlementDate = ql.Date().todaysDate()
yts = ql.FlatForward(2, ql.TARGET(), 0.05, ql.Actual360())

CDS_tenors = [ql.Period(6, ql.Months), ql.Period(1, ql.Years), ql.Period(2, ql.Years), ql.Period(3, ql.
↪Years), \
    ql.Period(4, ql.Years), ql.Period(5, ql.Years), ql.Period(7, ql.Years), ql.Period(10, ql.Years), ql.
↪Period(50, ql.Years)]
CDS_ctpy = [26.65, 37.22, 53.17, 65.79, 77.39, 91.14, 116.84, 136.67, 136.67]

CDSHelpers_ctpy = [ql.SpreadCdsHelper((CDS_spread / 10000.0), CDS_tenor, 0, ql.TARGET(), ql.Quarterly,␣
↪ql.Following, \
    ql.DateGeneration.TwentiethIMM, ql.Actual360(), recoveryRate, ql.YieldTermStructureHandle(yts))
for CDS_spread, CDS_tenor in zip(CDS_ctpy, CDS_tenors)]

pd_curve = ql.PiecewiseFlatHazardRate(settlementDate, CDSHelpers_ctpy, ql.Thirty360())
```

11.6.3 SurvivalProbabilityCurve

ql.SurvivalProbabilityCurve(*dates*, *survivalProbabilities*, *dayCounter*, *calendar*)

```
today = ql.Date().todaysDate()
dates = [today + ql.Period(n , ql.Years) for n in range(11)]
sp = [1.0, 0.9941, 0.9826, 0.9674, 0.9488, 0.9246, 0.8945, 0.8645, 0.83484, 0.80614, 0.7784]
crv = ql.SurvivalProbabilityCurve(dates, sp, ql.Actual360(), ql.TARGET())
crv.enableExtrapolation()
```

11.7 Inflation Term Structures

11.7.1 ZeroInflationCurve

ql.PiecewiseZeroInflation(*referenceDate*, *calendar*, *dayCounter*, *observationLag*, *frequency*, *bool indexIsInterpolated*, *baseZeroRate*, *nominalTS*, *helpers*, *accuracy=1.0e-12*, *interpolator=ql.Linear()*)

11.7.2 YoYInflationCurve

11.7.3 PiecewiseZeroInflation

Chapter 12

Helpers

12.1 Interest Rate

12.1.1 DepositRateHelper

`ql.DepositRateHelper(`*quote, tenor, fixingDays, calendar, convention, endOfMonth, dayCounter*`)`

```
quote = ql.QuoteHandle(ql.SimpleQuote(0.05))
tenor = ql.Period('6M')
fixingDays = 2
calendar = ql.TARGET()
convention = ql.ModifiedFollowing
endOfMonth = False
dayCounter = ql.Actual360()
ql.DepositRateHelper(quote, tenor, fixingDays, calendar, convention, endOfMonth, dayCounter)
```

`ql.DepositRateHelper(`*rate, tenor, fixingDays, calendar, convention, endOfMonth, dayCounter*`)`

```
ql.DepositRateHelper(0.05, ql.Period('6M'), 2, ql.TARGET(), ql.ModifiedFollowing, False, ql.Actual360())
```

`ql.DepositRateHelper(`*quote, index*`)`

```
ql.DepositRateHelper(ql.QuoteHandle(ql.SimpleQuote(0.05)), ql.Euribor6M())
```

`ql.DepositRateHelper(`*rate, index*`)`

```
ql.DepositRateHelper(0.05, ql.Euribor6M());
```

12.1.2 FraRateHelper

12.1.2.1 from months with quote

`ql.FraRateHelper(`*quote, monthsToStart, monthsToEnd, fixingDays, calendar, convention, endOfMonth, dayCounter, pillar=ql.Pillar.LastRelevantDate, customPillarDate=ql.Date(), useIndexedCoupon=True*`)`

```
quote = ql.QuoteHandle(ql.SimpleQuote(0.05))
monthsToStart = 1
monthsToEnd = 7
fixingDays = 2
calendar = ql.TARGET()
convention = ql.ModifiedFollowing
```

(continues on next page)

(continued from previous page)

```
endOfMonth = False
dayCounter = ql.Actual360()
ql.FraRateHelper (quote, monthsToStart, monthsToEnd, fixingDays, calendar, convention, endOfMonth,␣
↪dayCounter)
ql.FraRateHelper (quote, monthsToStart, monthsToEnd, fixingDays, calendar, convention, endOfMonth,␣
↪dayCounter, ql.Pillar.LastRelevantDate)
ql.FraRateHelper (quote, monthsToStart, monthsToEnd, fixingDays, calendar, convention, endOfMonth,␣
↪dayCounter, ql.Pillar.LastRelevantDate, ql.Date())
ql.FraRateHelper (quote, monthsToStart, monthsToEnd, fixingDays, calendar, convention, endOfMonth,␣
↪dayCounter, ql.Pillar.LastRelevantDate, ql.Date(), True)
```

12.1.2.2 from months with rate

`ql.FraRateHelper`(*rate, monthsToStart, monthsToEnd, fixingDays, calendar, convention, endOfMonth, dayCounter, pillar=ql.Pillar.LastRelevantDate, customPillarDate=ql.Date(), useIndexedCoupon=True*)

```
rate = 0.05
monthsToStart = 1
monthsToEnd = 7
fixingDays = 2
calendar = ql.TARGET()
convention = ql.ModifiedFollowing
endOfMonth = False
dayCounter = ql.Actual360()
ql.FraRateHelper (rate, monthsToStart, monthsToEnd, fixingDays, calendar, convention, endOfMonth,␣
↪dayCounter)
ql.FraRateHelper (rate, monthsToStart, monthsToEnd, fixingDays, calendar, convention, endOfMonth,␣
↪dayCounter, ql.Pillar.LastRelevantDate)
ql.FraRateHelper (rate, monthsToStart, monthsToEnd, fixingDays, calendar, convention, endOfMonth,␣
↪dayCounter, ql.Pillar.LastRelevantDate, ql.Date())
ql.FraRateHelper (rate, monthsToStart, monthsToEnd, fixingDays, calendar, convention, endOfMonth,␣
↪dayCounter, ql.Pillar.LastRelevantDate, ql.Date(), True)
```

12.1.2.3 from quote, monthsToStart and index

`ql.FraRateHelper`(*quote, monthsToStart, index, pillar=ql.Pillar.LastRelevantDate, customPillarDate=ql.Date(), useIndexedCoupon=True*)

```
quote = ql.QuoteHandle(ql.SimpleQuote(0.05))
monthsToStart = 1
index = ql.Euribor6M()
ql.FraRateHelper(quote, monthsToStart, index)
ql.FraRateHelper(quote, monthsToStart, index, ql.Pillar.LastRelevantDate)
ql.FraRateHelper(quote, monthsToStart, index, ql.Pillar.LastRelevantDate, ql.Date())
ql.FraRateHelper(quote, monthsToStart, index, ql.Pillar.LastRelevantDate, ql.Date(), True)
```

12.1.2.4 from price, monthsToStart and index

`ql.FraRateHelper(`*rate, monthsToStart, index, pillar=ql.Pillar.LastRelevantDate, customPillarDate=ql.Date(), useIndexedCoupon=True*`)`

```
rate = 0.05
monthsToStart = 1
index = ql.Euribor6M()
h = ql.FraRateHelper(rate, monthsToStart, index)
ql.FraRateHelper(rate, monthsToStart, index, ql.Pillar.LastRelevantDate)
ql.FraRateHelper(rate, monthsToStart, index, ql.Pillar.LastRelevantDate, ql.Date())
ql.FraRateHelper(rate, monthsToStart, index, ql.Pillar.LastRelevantDate, ql.Date(), True)
```

12.1.2.5 from quote, immOffsets and index

`ql.FraRateHelper(`*quote, immOffsetStart, immOffsetEnd, index, pillar=ql.Pillar.LastRelevantDate, customPillarDate=ql.Date(), useIndexedCoupon=True*`)`

```
quote = ql.QuoteHandle(ql.SimpleQuote(0.05))
immOffsetStart = 1
immOffsetEnd = 2
index = ql.Euribor6M()
ql.FraRateHelper(quote, immOffsetStart, immOffsetEnd, index)
ql.FraRateHelper(quote, immOffsetStart, immOffsetEnd, index, ql.Pillar.LastRelevantDate)
ql.FraRateHelper(quote, immOffsetStart, immOffsetEnd, index, ql.Pillar.LastRelevantDate, ql.Date())
ql.FraRateHelper(quote, immOffsetStart, immOffsetEnd, index, ql.Pillar.LastRelevantDate, ql.Date(), True)
```

12.1.3 Futures

12.1.3.1 FuturesRateHelper

`ql.FuturesRateHelper(`*price, iborStartDate, iborIndex, convexityAdjustment=0.0, type=ql.Futures.IMM*`)`

```
price = 100
index = ql.Euribor3M()
iborStartDate = ql.Date(17,6,2020)
ql.FuturesRateHelper(price, iborStartDate, index)
ql.FuturesRateHelper(price, iborStartDate, index, 0.01)
ql.FuturesRateHelper(price, iborStartDate, index, 0.01, ql.Futures.IMM)
ql.FuturesRateHelper(price, ql.Date(8,5,2020), index, 0.01, ql.Futures.ASX)
```

`ql.FuturesRateHelper(`*quote, iborStartDate, iborIndex, convexityAdjustment=ql.QuoteHandle(), type=ql.Futures.IMM*`)`

```
quote = ql.QuoteHandle(ql.SimpleQuote(100))
index = ql.Euribor3M()
iborStartDate = ql.Date(17,6,2020)
convexityAdjustment = ql.QuoteHandle(ql.SimpleQuote(0.01))
ql.FuturesRateHelper(quote, iborStartDate, index)
ql.FuturesRateHelper(quote, iborStartDate, index, convexityAdjustment)
ql.FuturesRateHelper(quote, iborStartDate, index, convexityAdjustment, ql.Futures.IMM)
ql.FuturesRateHelper(quote, ql.Date(8,5,2020), index, convexityAdjustment, ql.Futures.ASX)
```

`ql.FuturesRateHelper(`*price, iborStartDate, lengthInMonths, calendar, convention, endOfMonth, dayCounter, convexityAdjustment=0.0, type=ql.Futures.IMM*`)`

```
price = 100
iborStartDate = ql.Date(17,6,2020)
lengthInMonths = 3
```

(continues on next page)

(continued from previous page)

```
calendar = ql.TARGET()
convention = ql.Following
endOfMonth = False
dayCounter = ql.Actual360()
ql.FuturesRateHelper (price, iborStartDate, lengthInMonths, calendar, convention, endOfMonth, dayCounter)
ql.FuturesRateHelper (price, iborStartDate, lengthInMonths, calendar, convention, endOfMonth, dayCounter,
↪ 0.01)
ql.FuturesRateHelper (price, iborStartDate, lengthInMonths, calendar, convention, endOfMonth, dayCounter,
↪ 0.01, ql.Futures.IMM)
```

`ql.FuturesRateHelper`(*price, iborStartDate, lengthInMonths, calendar, convention, endOfMonth, dayCounter, convexityAdjustment=0.0, type=ql.Futures.IMM*)

```
quote = ql.QuoteHandle(ql.SimpleQuote(100))
iborStartDate = ql.Date(17,6,2020)
lengthInMonths = 3
calendar = ql.TARGET()
convention = ql.Following
endOfMonth = False
dayCounter = ql.Actual360()
convexityAdjustment = ql.QuoteHandle(ql.SimpleQuote(0.01))
ql.FuturesRateHelper (quote, iborStartDate, lengthInMonths, calendar, convention, endOfMonth, dayCounter)
ql.FuturesRateHelper (quote, iborStartDate, lengthInMonths, calendar, convention, endOfMonth, dayCounter,
↪ convexityAdjustment)
ql.FuturesRateHelper (quote, iborStartDate, lengthInMonths, calendar, convention, endOfMonth, dayCounter,
↪ convexityAdjustment, ql.Futures.IMM)
```

`ql.FuturesRateHelper`(*price, iborStartDate, iborEndDate, dayCounter, convexityAdjustment=0.0, ql.Futures.IMM*)

```
price = 100
iborStartDate = ql.Date(17,6,2020)
iborEndDate = ql.Date(17,9,2020)
dayCounter = ql.Actual360()
ql.FuturesRateHelper (price, iborStartDate, iborEndDate, dayCounter)
ql.FuturesRateHelper (price, iborStartDate, iborEndDate, dayCounter, 0.01)
ql.FuturesRateHelper (price, iborStartDate, iborEndDate, dayCounter, 0.01, ql.Futures.IMM)
```

`ql.FuturesRateHelper`(*quote, iborStartDate, iborEndDate, dayCounter, convexityAdjustment=ql.QuoteHandle(), ql.Futures.IMM*)

```
quote = ql.QuoteHandle(ql.SimpleQuote(100))
iborStartDate = ql.Date(17,6,2020)
iborEndDate = ql.Date(17,9,2020)
dayCounter = ql.Actual360()
convexityAdjustment = ql.QuoteHandle(ql.SimpleQuote(0.01))
ql.FuturesRateHelper (quote, iborStartDate, iborEndDate, dayCounter)
ql.FuturesRateHelper (quote, iborStartDate, iborEndDate, dayCounter, convexityAdjustment)
ql.FuturesRateHelper (quote, iborStartDate, iborEndDate, dayCounter, convexityAdjustment, ql.Futures.IMM)
```

12.1.3.2 OvernightIndexFutureRateHelper

`ql.OvernightIndexFutureRateHelper(`*quote, valueDate, maturityDate, overnightIndex, convexityAdjustmentQuote=ql.QuoteHandle(), nettingType=ql.OvernightIndexFuture.Compounding*`)`

Netting Types:

- Averaging
- Compounding

```
overnightIndex = ql.FedFunds()
priceQuote = ql.QuoteHandle(ql.SimpleQuote(99.92))
valueDate = ql.Date(3, 7, 2017)
maturityDate = ql.Date(30, 6, 2020 )
convexityAdjustment = ql.QuoteHandle()
netting = ql.OvernightIndexFuture.Averaging
future = ql.OvernightIndexFutureRateHelper(priceQuote, valueDate, maturityDate, overnightIndex)
future = ql.OvernightIndexFutureRateHelper(priceQuote, valueDate, maturityDate, overnightIndex,␣
↪convexityAdjustment, netting)
```

12.1.3.3 SofrFutureRateHelper

`ql.SofrFutureRateHelper(`*price, month, year, frequency, index*`)`

`ql.SofrFutureRateHelper(`*priceQuote, month, year, frequency, index*`)`

```
price = 99.915
ql.SofrFutureRateHelper(price, 3, 2020, ql.Quarterly, ql.Sofr())

priceQuote = ql.QuoteHandle(ql.SimpleQuote(price))
ql.SofrFutureRateHelper(priceQuote, 3, 2020, ql.Quarterly, ql.Sofr())
```

`ql.SofrFutureRateHelper(`*price, month, year, frequency, index, convexityAdjustment=0*`)`

`ql.SofrFutureRateHelper(`*priceQuote, month, year, frequency, index, convexityAdjustmentQuote=ql.QuoteHandle()*`)`

```
price = 99.915
convexityAdjustment = 0.004
ql.SofrFutureRateHelper(price, 3, 2020, ql.Quarterly, ql.Sofr(), convexityAdjustment)

priceQuote = ql.QuoteHandle(ql.SimpleQuote(price))
convexityAdjustmentQuote = ql.QuoteHandle(ql.SimpleQuote(convexityAdjustment))
ql.SofrFutureRateHelper(priceQuote, 3, 2020, ql.Quarterly, ql.Sofr(), convexityAdjustmentQuote)
```

`ql.SofrFutureRateHelper(`*price, month, year, frequency, index, convexityAdjustment=0, nettingType=ql.OvernightIndexFuture.Compounding*`)`

`ql.SofrFutureRateHelper(`*priceQuote, month, year, frequency, index, convexityAdjustment=0, nettingType=ql.OvernightIndexFuture.Compounding*`)`

Netting Types:

- Averaging
- Compounding

```
price = 99.915
convexityAdjustment = 0.004
ql.SofrFutureRateHelper(price, 3, 2020, ql.Quarterly, ql.Sofr(), 0.004, ql.OvernightIndexFuture.
↪Averaging)

priceQuote = ql.QuoteHandle(ql.SimpleQuote(price))
```

(continues on next page)

(continued from previous page)

```
convexityAdjustmentQuote = ql.QuoteHandle(ql.SimpleQuote(convexityAdjustment))

ql.SofrFutureRateHelper(priceQuote,3,2020,ql.Quarterly, ql.Sofr(), convexityAdjustmentQuote, ql.
↪OvernightIndexFuture.Compounding)
```

12.1.3.4 IMM

(Not a helper)

`ql.IMM.date(codeString, date=ql.Date())`

```
ql.IMM.date('M0')
ql.IMM.date('M0', ql.Date(20,6,2020))
```

ql.IMM.code(*immDate*)

```
immDate = ql.Date(16,12,2020)
ql.IMM.code(immDate)
```

ql.IMM.isIMMcode(*codeString*, *mainCycle=True*)

```
ql.IMM.isIMMcode('M0')
ql.IMM.isIMMcode('H0', True)
```

ql.IMM.isIMMdate(*date*, *mainCycle=True*)

```
dt = ql.Date(15,1,2020)
ql.IMM.isIMMdate(dt, True)
```

```
dates = ql.MakeSchedule(ql.Date(16,3,2020), ql.Date(16,12,2020), ql.Period('1M'))
list(map(ql.IMM.isIMMdate, dates))
```

ql.IMM.nextCode()

ql.IMM.nextCode(*date*)

ql.IMM.nextCode(*date*, *mainCycle=True*)

ql.IMM.nextCode(*codeString*)

ql.IMM.nextCode(*codeString*, *mainCycle=True*)

ql.IMM.nextCode(*codeString*, *mainCycle=True*, *refDate*)

```
ql.IMM.nextCode()
ql.IMM.nextCode(ql.Date(7,5,2020))
ql.IMM.nextCode(ql.Date(7,5,2020), False)
ql.IMM.nextCode('K0')
ql.IMM.nextCode('K0', False)
ql.IMM.nextCode('M9', False, ql.Date(16,8,2019))
```

ql.IMM.nextDate()

ql.IMM.nextDate(*date*)

ql.IMM.nextDate(*date*, *mainCycle=True*)

ql.IMM.nextDate(*codeString*)

ql.IMM.nextDate(*codeString*, *mainCycle=True*)

ql.IMM.nextDate(*codeString*, *mainCycle=True*, *refDate*)

```
ql.IMM.nextDate()
ql.IMM.nextDate(ql.Date(7,5,2020))
ql.IMM.nextDate(ql.Date(7,5,2020), False)
ql.IMM.nextDate('K0')
ql.IMM.nextDate('K0', False)
ql.IMM.nextDate('M9', False, ql.Date(16,8,2019))
```

12.1.4 SwapRateHelper

`ql.SwapRateHelper(`*rate, swapIndex, spread=0, fwdStart=ql.Period(), discountingCurve=ql.YieldTermStructureHandle, pillar=ql.Pillar.LastRelevantDate, customPillarDate=ql.Date(), endOfMonth=Dalse*`)`

```
rate = 0.05
swapIndex = ql.EuriborSwapIsdaFixA(ql.Period('1y'))
spread = ql.QuoteHandle(ql.SimpleQuote(0.0))
ql.SwapRateHelper(rate, ql.EuriborSwapIsdaFixA(ql.Period('1y')))
ql.SwapRateHelper(rate, ql.EuriborSwapIsdaFixA(ql.Period('1y')), spread)
ql.SwapRateHelper(rate, ql.EuriborSwapIsdaFixA(ql.Period('1y')), spread, ql.Period('1M'))
discountCurve = ql.YieldTermStructureHandle(ql.FlatForward(2, ql.TARGET(), 0.05, ql.Actual360()))
ql.SwapRateHelper(rate, ql.EuriborSwapIsdaFixA(ql.Period('1y')), spread, ql.Period(), discountCurve)
```

`ql.SwapRateHelper(`*quote, tenor, calendar, fixedFrequency, fixedConvention, fixedDayCount, iborIndex, spread=ql.QuoteHandle(), fwdStart=ql.Period(), discountingCurve=ql.YieldTermStructureHandle(), settlementDays=Null< Natural >(), pillar=ql.Pillar.LastRelevantDate, customPillarDate=ql.Date(), endOfMonth=False*`)`

```
quote = ql.QuoteHandle(ql.SimpleQuote(0.05))
swapIndex = ql.EuriborSwapIsdaFixA(ql.Period('1y'))
spread = ql.QuoteHandle(ql.SimpleQuote(0.0))
ql.SwapRateHelper(quote, ql.EuriborSwapIsdaFixA(ql.Period('1y')))
ql.SwapRateHelper(quote, ql.EuriborSwapIsdaFixA(ql.Period('1y')), spread)
ql.SwapRateHelper(quote, ql.EuriborSwapIsdaFixA(ql.Period('1y')), spread, ql.Period('1M'))
discountCurve = ql.YieldTermStructureHandle(ql.FlatForward(2, ql.TARGET(), 0.05, ql.Actual360()))
ql.SwapRateHelper(quote, ql.EuriborSwapIsdaFixA(ql.Period('1y')), spread, ql.Period(), discountCurve)
```

`ql.SwapRateHelper(`*quote, tenor, calendar, fixedFrequency, fixedConvention, fixedDayCount, iborIndex, spread=ql.QuoteHandle(), fwdStart=ql.Period(), discountingCurve=ql.YieldTermStructureHandle(), settlementDays, pillar=ql.Pillar.LastRelevantDate, customPillarDate=ql.Date(), endOfMonth=False*`)`

```
rate = ql.QuoteHandle(ql.SimpleQuote(0.05))
tenor = ql.Period('5Y')
fixedFrequency = ql.Annual
fixedConvention = ql.Following
fixedDayCount = ql.Thirty360()
iborIndex = ql.Euribor6M()
ql.SwapRateHelper(rate, tenor, calendar, fixedFrequency, fixedConvention, fixedDayCount, iborIndex)
```

`ql.SwapRateHelper(`*rate, tenor, calendar, fixedFrequency, fixedConvention, fixedDayCount, iborIndex, spread=ql.QuoteHandle(), fwdStart=ql.Period(), discountingCurve=ql.YieldTermStructureHandle(), settlementDays, pillar=ql.Pillar.LastRelevantDate, customPillarDate=ql.Date(), endOfMonth=False*`)`

```
rate = 0.05
tenor = ql.Period('5Y')
fixedFrequency = ql.Annual
fixedConvention = ql.Following
fixedDayCount = ql.Thirty360()
```

(continues on next page)

(continued from previous page)

```
iborIndex = ql.Euribor6M()
ql.SwapRateHelper(rate, tenor, calendar, fixedFrequency, fixedConvention, fixedDayCount, iborIndex)
```

12.1.5 OISRateHelper

`ql.OISRateHelper(`*settlementDays, tenor, fixedRate, overnightIndex, discountingCurve=ql.YieldTermStructureHandle(), telescopicValueDates=False, paymentLag=0, paymentConvention=ql.Following, paymentFrequency=ql.Annual, paymentCalendar=ql.Calendar(), forwardStart=ql.Period(), overnightSpread=0.0, pillar=ql.Pillar.LastRelevantDate, customPillarDate=qlDate()*`)`

```
forward6mLevel = 0.025
forward6mQuote = ql.QuoteHandle(ql.SimpleQuote(forward6mLevel))
yts6m = ql.FlatForward(0, ql.TARGET(), forward6mQuote, ql.Actual365Fixed() )
yts6mh = ql.YieldTermStructureHandle(yts6m)
oishelper = ql.OISRateHelper(2,ql.Period("3M"), ql.QuoteHandle(ql.SimpleQuote(0.01)), ql.Eonia(yts6mh),
↪yts6mh, True)
```

12.1.6 DatedOISRateHelper

`ql.DatedOISRateHelper(`*startDate, endDate, fixedRate, overnightIndex, discountingCurve=ql.YieldTermStructureHandle(), telescopicValueDates=False*`)`

```
startDate = ql.Date(15,6,2020)
endDate = ql.Date(15,6,2021)
fixedRate = ql.QuoteHandle(ql.SimpleQuote(0.05))
overnightIndex = ql.Eonia()
ql.DatedOISRateHelper(startDate, endDate, fixedRate, overnightIndex)
```

12.1.7 FxSwapRateHelper

`ql.FxSwapRateHelper(`*fwdPoint, spotFx, tenor, fixingDays, calendar, convention, endOfMonth, isFxBaseCurrencyCollateralCurrency, collateralCurve*`)`

`ql.FxSwapRateHelper(`*fwdPoint, spotFx, tenor, fixingDays, calendar, convention, endOfMonth, isFxBaseCurrencyCollateralCurrency, collateralCurve, tradingCalendar=Calendar()*`)`

```
yts = ql.YieldTermStructureHandle(ql.FlatForward(2, ql.TARGET(), 0.02, ql.Actual360()))
spot = ql.QuoteHandle(ql.SimpleQuote(1.10))
fwdPoints = ql.QuoteHandle(ql.SimpleQuote(122.29))
ql.FxSwapRateHelper(fwdPoints, spot, ql.Period('6M'), 2, ql.TARGET(), ql.Following, False, True, yts)
```

12.1.8 CrossCurrencyBasisSwapRateHelper

`ql.CrossCurrencyBasisSwapRateHelper(`*basis, tenor, fixingDays, calendar, convention, endOfMonth, baseCurrencyIndex, quoteCurrencyIndex, collateralCurve, isFxBaseCurrencyCollateralCurrency, isBasisOnFxBaseCurrencyLeg*`)`

```
eur_curve = ql.YieldTermStructureHandle(ql.FlatForward(2, ql.TARGET(), 0.01, ql.Actual360()))
usd_curve = ql.YieldTermStructureHandle(ql.FlatForward(2, ql.TARGET(), 0.02, ql.Actual360()))

basis = ql.QuoteHandle(ql.SimpleQuote(0.005))
tenor = ql.Period('1Y')
fixingDays = 2
calendar = ql.TARGET()
```

(continues on next page)

(continued from previous page)

```
convention = ql.ModifiedFollowing
endOfMonth = True
baseCurrencyIndex = ql.USDLibor(ql.Period('3M'), usd_curve)
quoteCurrencyIndex = ql.Euribor3M(eur_curve)
collateralCurve = ql.YieldTermStructureHandle(ql.FlatForward(2, ql.TARGET(), 0.05, ql.Actual360()))
isFxBaseCurrencyCollateralCurrency = False
isBasisOnFxBaseCurrencyLeg = False

helper = ql.CrossCurrencyBasisSwapRateHelper(
    basis, tenor, fixingDays, calendar, convention, endOfMonth,
    baseCurrencyIndex, quoteCurrencyIndex, collateralCurve,
    isFxBaseCurrencyCollateralCurrency, isBasisOnFxBaseCurrencyLeg)
```

12.1.9 FixedRateBondHelper

`ql.FixedRateBondHelper(`*price, settlementDays, faceAmount, schedule, coupons, dayCounter, paymentConv=Following, redemption=100.0, issueDate=Date(), paymentCalendar=Calendar(), exCouponPeriod=Period(), exCouponCalendar=Calendar(), exCouponConvention=Unadjusted, exCouponEndOfMonth=False, useCleanPrice=True*`)`

```
quote = ql.QuoteHandle(ql.SimpleQuote(115.5))
settlementDays = 2
faceAmount = 100
schedule = ql.MakeSchedule(ql.Date(15,6,2020), ql.Date(15,6,2021), ql.Period('1y'))
coupons = [0.0195]
dayCounter = ql.ActualActual()
helper = ql.FixedRateBondHelper(quote, settlementDays, faceAmount, schedule, coupons, dayCounter)
```

12.1.10 BondHelper

`ql.BondHelper(`*cleanPrice, bond, useCleanPrice=True*`)`

```
bond = ql.FixedRateBond(
    2, ql.TARGET(), 100.0, ql.Date(15,12,2019), ql.Date(15,12,2024),
    ql.Period('1Y'), [0.05], ql.ActualActual())

cleanPrice = ql.QuoteHandle(ql.SimpleQuote(115))
ql.BondHelper(cleanPrice, bond)
```

12.1.11 BondHelperVector

`ql.BondHelperVector()`

```
bond_helpers = ql.BondHelperVector()
bond_helpers.append(bond_helper)
```

12.1.12 RateHelperVector

ql.RateHelperVector()

```
helpers = ql.RateHelperVector()
helpers.append(ql.DepositRateHelper(0.05, ql.Euribor6M()))
```

12.2 Volatility

12.2.1 CapHelper

ql.CapHelper(*period*, *quote*, *index*, *frequency*, *dayCounter*, *includeFirstOptionlet (bool)*, *YieldTermStructure*, *errorType=BlackCalibrationHelper.RelativePriceError*)

```
period = ql.Period('2y')
quote = ql.QuoteHandle(ql.SimpleQuote(0.55))
today = ql.Date().todaysDate()
yts = ql.YieldTermStructureHandle(ql.FlatForward(today, 0.02, ql.Actual360()))
index = ql.Euribor6M(yts)

helper = ql.CapHelper(period, quote, index, ql.Semiannual, ql.Actual360(), False, yts)
```

12.2.2 SwaptionHelper

ql.SwaptionHelper(*maturity*, *length*, *volatility*, *index*, *fixedLegTenor*, *fixedLegDayCounter*, *floatingLegDayCounter*, *termStructure*, *errorType=ql.BlackCalibrationHelper.RelativePriceError*, *strike=Null< Real >()*, *nominal=1.0*, *type=ql.ShiftedLognormal*, *shift=0.0*)

```
maturity = ql.Period('5Y')
length = ql.Period('5Y')
volatility = ql.QuoteHandle(ql.SimpleQuote(0.0055))
index = ql.Euribor6M()
fixedLegTenor = ql.Period('1Y')
fixedLegDayCounter = ql.Thirty360()
floatingLegDayCounter = ql.Actual360()

crv = ql.FlatForward(2, ql.TARGET(), 0.05, ql.Actual360())
yts = ql.YieldTermStructureHandle(crv)

ql.SwaptionHelper(
  maturity, length, volatility, index, fixedLegTenor,
  fixedLegDayCounter, floatingLegDayCounter, yts
)
```

ql.SwaptionHelper(*exerciseDate*, *length*, *volatility*, *index*, *fixedLegTenor*, *fixedLegDayCounter*, *floatingLegDayCounter*, *termStructure*, *errorType=ql.BlackCalibrationHelper.RelativePriceError*, *strike=Null< Real >()*, *nominal=1.0*, *type=ql.ShiftedLognormal*, *shift=0.0*)

```
exerciseDate = ql.Date(15,6,2020)
length = ql.Period('5Y')
volatility = ql.QuoteHandle(ql.SimpleQuote(0.0055))
index = ql.Euribor6M()
fixedLegTenor = ql.Period('1Y')
fixedLegDayCounter = ql.Thirty360()
floatingLegDayCounter = ql.Actual360()
```

(continues on next page)

(continued from previous page)

```
crv = ql.FlatForward(2, ql.TARGET(), 0.05, ql.Actual360())
yts = ql.YieldTermStructureHandle(crv)

ql.SwaptionHelper(
  exerciseDate, length, volatility, index, fixedLegTenor,
  fixedLegDayCounter, floatingLegDayCounter, yts
)
```

`ql.SwaptionHelper(`*exerciseDate, endDate, volatility, index, fixedLegTenor, fixedLegDayCounter, floatingLegDayCounter, termStructure, errorType=ql.BlackCalibrationHelper.RelativePriceError, strike=Null< Real >(), nominal=1.0, type=ql.ShiftedLognormal, shift=0.0*`)`

```
exerciseDate = ql.Date(15,6,2020)
endDate = ql.Date(15,6,2025)
volatility = ql.QuoteHandle(ql.SimpleQuote(0.0055))
index = ql.Euribor6M()
fixedLegTenor = ql.Period('1Y')
fixedLegDayCounter = ql.Thirty360()
floatingLegDayCounter = ql.Actual360()
blackCalibrationHelper = ql.BlackCalibrationHelper.RelativePriceError
strike = ql.nullDouble()
nominal = 1.0

crv = ql.FlatForward(2, ql.TARGET(), 0.05, ql.Actual360())
yts = ql.YieldTermStructureHandle(crv)

ql.SwaptionHelper(
  exerciseDate, endDate, volatility, index, fixedLegTenor,
  fixedLegDayCounter, floatingLegDayCounter, yts,
  blackCalibrationHelper, strike, nominal
)
```

12.2.3 HestonModelHelper

`ql.HestonModelHelper(`*tenor, calendar, spot, strike, volQuote, riskFreeCurveHandle, dividendCurveHandle, errorType=ql.BlackCalibrationHelper.RelativePriceError*`)`

```
spot, strike = 100, 110

tenor = ql.Period("3M")
calendar = ql.NullCalendar()
dayCount = ql.Actual365Fixed()
volQuote = ql.QuoteHandle(ql.SimpleQuote(0.22))

today = ql.Date().todaysDate()
riskFreeCurve = ql.FlatForward(today, 0.04, dayCount)
dividendCurve = ql.FlatForward(today, 0.0, dayCount)
riskFreeHandle = ql.YieldTermStructureHandle(riskFreeCurve)
dividendHandle = ql.YieldTermStructureHandle(dividendCurve)

ql.HestonModelHelper(tenor, calendar, spot, strike, volQuote, riskFreeHandle, dividendHandle)
```

12.3 Credit

12.3.1 SpreadCdsHelper

ql.SpreadCdsHelper(*runningSpread, tenor, settlementDays, calendar, frequency, paymentConvention, rule, dayCounter, recoveryRate, discountCurve, settlesAccrual=True, paysAtDefaultTime=True, startDate=ql.Date(), lastPeriodDayCounter=ql.DayCounter(), rebatesAccrual=True, model=ql.CreditDefaultSwap.Midpoint*)

```
runningSpread = ql.QuoteHandle(ql.SimpleQuote(0.005))
tenor = ql.Period('5Y')
settlementDays = 2
calendar = ql.TARGET()
frequency = ql.Annual
paymentConvention = ql.Following
rule = ql.DateGeneration.TwentiethIMM
dayCounter = ql.Actual365Fixed()
recoveryRate = 0.4

crv = ql.FlatForward(2, ql.TARGET(), 0.05, ql.Actual360())
yts = ql.YieldTermStructureHandle(crv)

ql.SpreadCdsHelper(
  runningSpread, tenor, settlementDays, calendar, frequency,
  paymentConvention, rule, dayCounter, recoveryRate, yts
)
```

12.4 Inflation

ql.ZeroCouponInflationSwapHelper(*quote, period, date, calendar, convention, daycounter, index*)

ql.ZeroCouponInflationSwapHelper(*quote, period, date, calendar, convention, daycounter, index, yieldTermStructure*)

```
quote = ql.QuoteHandle(ql.SimpleQuote(0.02))
period = ql.Period('6M')
date = ql.Date(15,6,2020)
calendar = ql.TARGET()
convention = ql.ModifiedFollowing
daycounter = ql.Actual360()
index = ql.EUHICPXT(True)

helper = ql.ZeroCouponInflationSwapHelper(quote, period, date, calendar, convention, daycounter, index)
```

Chapter 13

Fixed Income

13.1 Gearing in swaps

[2]:
```
import QuantLib as ql
```

Create a relinkable yield term structure handle and build a curve

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

yts = ql.YieldTermStructureHandle(ql.FlatForward(2, ql.TARGET(), 0.05, ql.Actual360()))
engine = ql.DiscountingSwapEngine(yts)
index = ql.USDLibor(ql.Period('6M'), yts)

schedule = ql.MakeSchedule(ql.Date(15,6,2021), ql.Date(15,6,2023), ql.Period('6M'))
nominal = [10e6]


fixedLeg = ql.FixedRateLeg(schedule, index.dayCounter(), nominal, [0.05])
floatingLeg = ql.IborLeg(nominal, schedule, index)
swap = ql.Swap(fixedLeg, floatingLeg)
swap.setPricingEngine(engine)

print(f"Floating leg NPV: {swap.legNPV(1):,.2f}\n")
pd.DataFrame([{
    'fixingDate': cf.fixingDate().ISO(),
    'accrualStart': cf.accrualStartDate().ISO(),
    'accrualEnd': cf.accrualEndDate().ISO(),
    "paymentDate": cf.date().ISO(),
    'gearing': cf.gearing(),
    'forward': cf.indexFixing(),
    'rate': cf.rate(),
    "amount": cf.amount()
} for cf in map(ql.as_floating_rate_coupon, swap.leg(1))])
```

```
Floating leg NPV: 933,741.01
```

[1]:
```
   fixingDate accrualStart  accrualEnd paymentDate  gearing   forward  \
0  2021-06-11   2021-06-15  2021-12-15  2021-12-15      1.0  0.050641
1  2021-12-13   2021-12-15  2022-06-15  2022-06-15      1.0  0.050637
2  2022-06-13   2022-06-15  2022-12-15  2022-12-15      1.0  0.050641
3  2022-12-13   2022-12-15  2023-06-15  2023-06-15      1.0  0.050637

       rate         amount
```

(continues on next page)

(continued from previous page)

```
0  0.050641  257424.241734
1  0.050637  255999.698407
2  0.050641  257424.241734
3  0.050637  255999.698407
```

[2]:
```
floatingLeg = ql.IborLeg(nominal, schedule, index, gearings=[0.7])
swap = ql.Swap(fixedLeg, floatingLeg)
swap.setPricingEngine(engine)

print(f"Floating leg NPV: {swap.legNPV(1):,.2f}\n")
pd.DataFrame([{
    'fixingDate': cf.fixingDate().ISO(),
    'accrualStart': cf.accrualStartDate().ISO(),
    'accrualEnd': cf.accrualEndDate().ISO(),
    "paymentDate": cf.date().ISO(),
    'gearing': cf.gearing(),
    'forward': cf.indexFixing(),
    'rate': cf.rate(),
    "amount": cf.amount()
} for cf in map(ql.as_floating_rate_coupon, swap.leg(1))])
```

```
Floating leg NPV: 653,618.71
```

[2]:
```
   fixingDate accrualStart  accrualEnd paymentDate  gearing   forward  \
0  2021-06-11   2021-06-15  2021-12-15  2021-12-15      0.7  0.050641
1  2021-12-13   2021-12-15  2022-06-15  2022-06-15      0.7  0.050637
2  2022-06-13   2022-06-15  2022-12-15  2022-12-15      0.7  0.050641
3  2022-12-13   2022-12-15  2023-06-15  2023-06-15      0.7  0.050637

       rate         amount
0  0.035449  180196.969214
1  0.035446  179199.788885
2  0.035449  180196.969214
3  0.035446  179199.788885
```

[4]:
```
swapType = ql.VanillaSwap.Payer
numDates = (len(schedule)-1)
gearing = [0.7] * numDates
spread = [0.0] * numDates
fixedRateArray = [0.05] * numDates
nominalArray = nominal * numDates
nsSwap = ql.NonstandardSwap(
    swapType, nominalArray, nominalArray,
    schedule, fixedRateArray, index.dayCounter(),
    schedule, index, gearing, spread, index.dayCounter())

nsSwap.setPricingEngine(engine)
print(f"Floating leg NPV: {nsSwap.legNPV(1):,.2f}\n")
pd.DataFrame([{
    'fixingDate': cf.fixingDate().ISO(),
    'accrualStart': cf.accrualStartDate().ISO(),
    'accrualEnd': cf.accrualEndDate().ISO(),
    "paymentDate": cf.date().ISO(),
    'gearing': cf.gearing(),
    'forward': cf.indexFixing(),
    'rate': cf.rate(),
    "amount": cf.amount()
} for cf in map(ql.as_floating_rate_coupon, swap.leg(1))])
```

```
Floating leg NPV: 653,618.71
```

```
[4]:    fixingDate accrualStart  accrualEnd paymentDate  gearing   forward  \
0  2021-06-11   2021-06-15  2021-12-15  2021-12-15      1.0  0.050641
1  2021-12-13   2021-12-15  2022-06-15  2022-06-15      1.0  0.050637
2  2022-06-13   2022-06-15  2022-12-15  2022-12-15      1.0  0.050641
3  2022-12-13   2022-12-15  2023-06-15  2023-06-15      1.0  0.050637

       rate         amount
0  0.050641  180196.969214
1  0.050637  179199.788885
2  0.050641  180196.969214
3  0.050637  179199.788885
```

```
[ ]:
```

13.2 Vanilla Swap

```
[2]: import QuantLib as ql
```

Create a relinkable yield term structure handle and build a curve

```
[3]: yts = ql.RelinkableYieldTermStructureHandle()

instruments = [
    ('depo', '6M', 0.025),
    ('fra', '6M', 0.03),
    ('swap', '1Y', 0.031),
    ('swap', '2Y', 0.032),
    ('swap', '3Y', 0.035)
]

helpers = ql.RateHelperVector()
index = ql.Euribor6M(yts)
for instrument, tenor, rate in instruments:
    if instrument == 'depo':
        helpers.append( ql.DepositRateHelper(rate, index) )
    if instrument == 'fra':
        monthsToStart = ql.Period(tenor).length()
        helpers.append( ql.FraRateHelper(rate, monthsToStart, index) )
    if instrument == 'swap':
        swapIndex = ql.EuriborSwapIsdaFixA(ql.Period(tenor))
        helpers.append( ql.SwapRateHelper(rate, swapIndex))
curve = ql.PiecewiseLogCubicDiscount(2, ql.TARGET(), helpers, ql.ActualActual())
```

Link the built curve to the relinkable yield term structure handle and build a swap pricing engine

```
[4]: yts.linkTo(curve)
engine = ql.DiscountingSwapEngine(yts)
```

Build a vanilla swap and provide a pricing engine

```
[5]: tenor = ql.Period('2y')
fixedRate = 0.05
forwardStart = ql.Period("2D")

swap = ql.MakeVanillaSwap(tenor, index, fixedRate, forwardStart, Nominal=10e6, pricingEngine=engine)
```

Get the fair rate and NPV

[6]:
```
fairRate = swap.fairRate()
npv = swap.NPV()

print(f"Fair swap rate: {fairRate:.3%}")
print(f"Swap NPV: {npv:,.3f}")
```

```
Fair swap rate: 3.232%
Swap NPV: -337,608.498
```

[10]:
```
import pandas as pd
pd.options.display.float_format = "{:,.2f}".format

cashflows = pd.DataFrame({
    'date': cf.date(),
    'amount': cf.amount()
    } for cf in swap.leg(1))
display(cashflows)
```

	date	amount
0	April 6th, 2021	129,166.12
1	October 5th, 2021	166,342.17
2	April 5th, 2022	-100,976.64
3	October 5th, 2022	455,178.07

[23]:
```
cashflows = pd.DataFrame({
    'nominal': cf.nominal(),
    'accrualStartDate': cf.accrualStartDate().ISO(),
    'accrualEndDate': cf.accrualEndDate().ISO(),
    'rate': cf.rate(),
    'amount': cf.amount()
    } for cf in map(ql.as_coupon, swap.leg(1)))
display(cashflows)
```

	nominal	accrualStartDate	accrualEndDate	rate	amount
0	10,000,000.00	2020-10-05	2021-04-06	0.03	129,166.12
1	10,000,000.00	2021-04-06	2021-10-05	0.03	166,342.17
2	10,000,000.00	2021-10-05	2022-04-05	-0.02	-100,976.64
3	10,000,000.00	2022-04-05	2022-10-05	0.09	455,178.07

13.3 Yield Curve

[5]:
```
import QuantLib as ql
import matplotlib.pyplot as plt
```

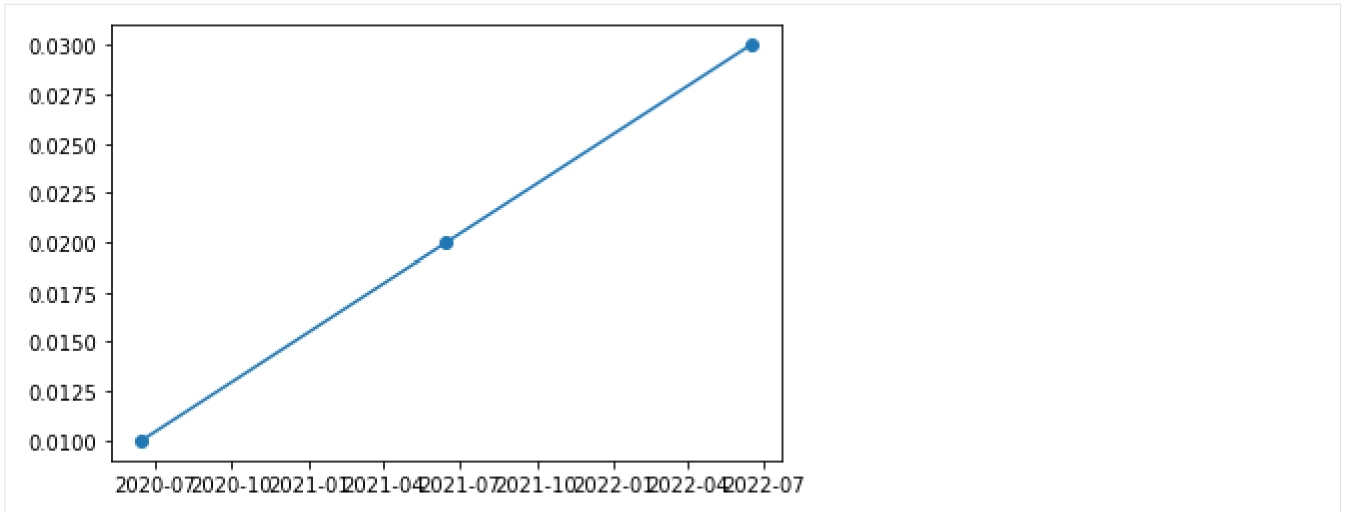
[3]:
```
dates = [ql.Date(15,6,2020), ql.Date(15,6,2021), ql.Date(15,6,2022)]
zeros = [0.01, 0.02, 0.03]
curve = ql.ZeroCurve(dates, zeros, ql.ActualActual(), ql.TARGET())
```

[4]:
```
curve.nodes()
```

[4]:
```
((Date(15,6,2020), 0.01), (Date(15,6,2021), 0.02), (Date(15,6,2022), 0.03))
```

[15]:
```
plt.plot(*list(zip(*[(dt.to_date(), rate) for dt,rate in curve.nodes()])), marker='o')
```

[15]:
```
[<matplotlib.lines.Line2D at 0x7f6933e5ddf0>]
```


0.0300
0.0275
0.0250
0.0225
0.0200
0.0175
0.0150
0.0125
0.0100
2020-07
2020-10
2021-01
2021-04
2021-07
2021-10
2022-01
2022-04
2022-07

```
[ ]:
```

Chapter 14

Inflation

Chapter 15

Credit

Chapter 16

Equity

Chapter 17

Indices and tables

- genindex
- modindex
- search

Index

B

bondYield() (*ql.Bond. method*), 32
built-in function
    GaussianRandomNumberGenerator(), 51
    HaltonRsg(), 52
    ql.AmortizingFixedRateBond(), 33
    ql.AmortizingPayment(), 8
    ql.AnalyticBarrierEngine(), 67
    ql.AnalyticBinaryBarrierEngine(), 68
    ql.AnalyticCapFloorEngine(), 56
    ql.AnalyticContinuousGeometricAveragePriceAsianEngine(), 64
    ql.AnalyticContinuousGeometricAveragePriceAsianHestonEngine(), 66
    ql.AnalyticDiscreteGeometricAveragePriceAsianEngine(), 64
    ql.AnalyticDiscreteGeometricAveragePriceAsianHestonEngine(), 65
    ql.AnalyticDoubleBarrierBinaryEngine(), 69
    ql.AnalyticDoubleBarrierEngine(), 68
    ql.AnalyticEuropeanEngine(), 60
    ql.AnalyticHestonEngine(), 62
    ql.AnalyticHestonForwardEuropeanEngine(), 71
    ql.AnalyticPTDHestonEngine(), 63
    ql.AndreasenHugeLocalVolAdapter(), 92
    ql.AndreasenHugeVolatilityAdapter(), 90
    ql.Array(), 3
    ql.AssetSwap(), 37
    ql.BachelierCapFloorEngine(), 56
    ql.BachelierSwaptionEngine(), 58
    ql.BarrierOption(), 44
    ql.BasketOption(), 45
    ql.BinomialBarrierEngine(), 67
    ql.BlackCallableFixedRateBondEngine(), 55
    ql.BlackCapFloorEngine(), 56
    ql.BlackCdsOptionEngine(), 60
    ql.BlackConstantVol(), 88
    ql.BlackDeltaCalculator(), 54
    ql.BlackIborCouponPricer(), 12
    ql.BlackKarasinski(), 74
    ql.BlackProcess(), 78
    ql.BlackScholesMertonProcess(), 77
    ql.BlackScholesProcess(), 77
    ql.BlackSwaptionEngine(), 58
    ql.BlackVarianceCurve(), 89
    ql.BlackVarianceSurface(), 89
    ql.BlackVolTermStructureHandle(), 91
    ql.BondHelper(), 109
    ql.BondHelperVector(), 109
    ql.Calendar.holidayList(), 23
    ql.CallableFixedRateBond(), 34
    ql.Cap(), 39
    ql.CapFloorTermVolCurve(), 93, 94
    ql.CapFloorTermVolSurface(), 94
    ql.CapHelper(), 110
    ql.CappedFlooredCmsCoupon(), 10
    ql.CappedFlooredCoupon(), 9
    ql.CashFlows.atmRate(), 13
    ql.CashFlows.basisPointValue(), 14
    ql.CashFlows.bps(), 13, 14
    ql.CashFlows.convexity(), 14
    ql.CashFlows.duration(), 14
    ql.CashFlows.maturityDate(), 13
    ql.CashFlows.nextCashFlowDate(), 13
    ql.CashFlows.npv(), 13, 14
    ql.CashFlows.previousCashFlowDate(), 13
    ql.CashFlows.startDate(), 13
    ql.CashFlows.yieldRate(), 14
    ql.CashFlows.zSpread(), 15
    ql.CdsOption(), 42
    ql.CmsCoupon(), 9
    ql.CmsRateBond(), 34
    ql.CmsSpreadCoupon(), 10
    ql.Collar(), 40
    ql.ConstantOptionletVolatility(), 93
    ql.ConstantSwaptionVolatility(), 95
    ql.CPIBond(), 40
    ql.CreditDefaultSwap(), 42
    ql.CrossCurrencyBasisSwapRateHelper(), 108
    ql.Date(), 20
    ql.DatedOISRateHelper(), 108
    ql.DeltaVolQuote(), 5
    ql.DepositRateHelper(), 101
    ql.DerivedQuote(), 5
    ql.DiscountCurve(), 83
    ql.DiscountingBondEngine(), 55
    ql.DiscountingSwapEngine(), 57

ql.DiscreteAveragingAsianOption(), 43
ql.DoubleBarrierOption(), 44
ql.Euribor(), 27
ql.EuriborSwapIsdaFixA(), 28
ql.ExtendedOrnsteinUhlenbeckProcess(), 78
ql.FdBlackScholesAsianEngine(), 65
ql.FdBlackScholesBarrierEngine(), 67
ql.FdBlackScholesRebateEngine(), 67
ql.FdBlackScholesVanillaEngine(), 61
ql.FdG2SwaptionEngine(), 58
ql.FdHestonBarrierEngine(), 68
ql.FdHestonDoubleBarrierEngine(), 69
ql.FdHestonVanillaEngine(), 63
ql.FdHullWhiteSwaptionEngine(), 58
ql.FittedBondDiscountCurve(), 87
ql.FixedRateBond(), 33
ql.FixedRateBondHelper(), 109
ql.FixedRateCoupon(), 8
ql.FixedRateLeg(), 11
ql.FlatForward(), 83
ql.FlatHazardRate(), 99
ql.FloatingRateBond(), 33
ql.Floor(), 40
ql.ForwardCurve(), 84
ql.ForwardEuropeanEngine(), 70
ql.ForwardSpreadedTermStructure(), 86
ql.ForwardVanillaOption(), 45
ql.FraRateHelper(), 101–103
ql.FuturesRateHelper(), 103, 104
ql.FxSwapRateHelper(), 108
ql.G2(), 75
ql.G2SwaptionEngine(), 58
ql.GarmanKohlagenProcess(), 79
ql.GaussianMultiPathGenerator(), 52
ql.GaussianSobolMultiPathGenerator(), 53
ql.GeneralizedBlackScholesProcess(), 78
ql.GeometricBrownianMotionProcess(), 77
ql.Gsr(), 74
ql.HestonBlackVolSurface(), 90
ql.HestonModel(), 73
ql.HestonModelHelper(), 111
ql.HestonProcess(), 79
ql.HestonSLVProcess(), 80
ql.HullWhite(), 74
ql.HullWhiteForwardProcess(), 81
ql.HullWhiteProcess(), 81
ql.IborCoupon(), 9
ql.IborIndex(), 27
ql.IborLeg(), 11
ql.IMM.code(), 106
ql.IMM.isIMMcode(), 106
ql.IMM.isIMMdate(), 106
ql.IMM.nextCode(), 106
ql.IMM.nextDate(), 106
ql.ImpliedTermStructure(), 86
ql.IntegralCdsEngine(), 60
ql.InterestRate(), 7
ql.IsdaCdsEngine(), 59
ql.JamshidianSwaptionEngine(), 59
ql.JointCalendar(), 24
ql.LinearTsrPricer(), 12
ql.LocalConstantVol(), 91
ql.LocalVolSurface(), 91
ql.MakeOIS(), 38
ql.MakeSchedule(), 26
ql.MakeVanillaSwap(), 36
ql.Matrix(), 4
ql.MCAmericanEngine(), 61
ql.MCDigitalEngine(), 62
ql.MCDiscreteArithmeticAPEngine(), 65
ql.MCDiscreteArithmeticAPHestonEngine(), 66
ql.MCDiscreteGeometricAPEngine(), 64
ql.MCDiscreteGeometricAPHestonEngine(), 66
ql.MCEuropeanBasketEngine(), 69
ql.MCEuropeanEngine(), 61
ql.MCEuropeanHestonEngine(), 62
ql.MCForwardEuropeanBSEngine(), 70
ql.MCForwardEuropeanHestonEngine(), 71
ql.MidPointCdsEngine(), 59
ql.Money(), 17
ql.NoExceptLocalVolSurface(), 92
ql.NonstandardSwap(), 38
ql.OISRateHelper(), 108
ql.OptionletStripper1(), 95
ql.OptionletVolatilityStructureHandle(), 95
ql.OvernightIndex(), 27
ql.OvernightIndexedCoupon(), 9
ql.OvernightIndexedSwap(), 37
ql.OvernightIndexFutureRateHelper(), 105
ql.OvernightLeg(), 12
ql.Period(), 21
ql.Piecewise(), 85
ql.PiecewiseFlatHazardRate(), 99
ql.PiecewiseTimeDependentHestonModel(), 73
ql.PiecewiseYieldCurve(), 85
ql.PiecewiseZeroInflation(), 100
ql.RateHelperVector(), 110
ql.Redemption(), 8
ql.RelinkableBlackVolTermStructureHandle(), 91
ql.RelinkableOptionletVolatilityStructureHandle(), 95
ql.RelinkableSwaptionVolatilityStructureHandle(), 98
ql.sabrFlochKennedyVolatility(), 99
ql.SabrSmileSection(), 98
ql.sabrVolatility(), 98
ql.shiftedSabrVolatility(), 98
ql.SimpleCashFlow(), 8
ql.SimpleQuote(), 5
ql.SofrFutureRateHelper(), 105
ql.SpreadCdsHelper(), 112
ql.SpreadedLinearZeroInterpolatedTermStructure(), 87
ql.StochasticProcessArray(), 81
ql.StrippedOptionletAdapter(), 95

ql.SurvivalProbabilityCurve(), 100
ql.Swap(), 36
ql.SwapIndex(), 28
ql.SwapRateHelper(), 107
ql.Swaption(), 39
ql.SwaptionHelper(), 110, 111
ql.SwaptionVolatilityMatrix(), 96
ql.SwaptionVolatilityStructureHandle(), 98
ql.SwaptionVolCube2(), 97
ql.TimeGrid(), 26
ql.TreeCallableFixedRateBondEngine(), 55, 56
ql.TreeCapFloorEngine(), 57
ql.TreeSwaptionEngine(), 59
ql.VanillaOption(), 43
ql.VanillaSwap(), 35
ql.Vasicek(), 74
ql.YearOnYearInflationSwap(), 42
ql.ZeroCouponBond(), 33
ql.ZeroCouponInflationSwap(), 41
ql.ZeroCouponInflationSwapHelper(), 112
ql.ZeroCurve(), 84
ql.ZeroSpreadedTermStructure(), 86
RandomNumberGenerator(), 50
Schedule(), 25
SobolRsg(), 52
SwapSpreadIndex(), 28
XXXInterpolation(), 49
businessDayConvention() (*ql.ForwardRateAgreement. method*), 31

C

calendar() (*ql.ForwardRateAgreement. method*), 31
chain(r1, r2), **18**

D

dayCounter() (*ql.ForwardRateAgreement. method*), 31
dirtyPrice() (*ql.Bond. method*), 32
discountCurve() (*ql.ForwardRateAgreement. method*), 31

E

exchange(amount), **18**
ExchangeRate.Derived, **18**
ExchangeRate.Direct, **18**

F

fixingDate() (*ql.ForwardRateAgreement. method*), 31
forwardRate() (*ql.ForwardRateAgreement. method*), 31
forwardValue() (*ql.ForwardRateAgreement. method*), 31

G

GaussianRandomNumberGenerator()
built-in function, 51

H

HaltonRsg()
built-in function, 52

I

impliedYield() (*ql.ForwardRateAgreement. method*), 31
incomeDiscountCurve() (*ql.ForwardRateAgreement. method*), 31
isExpired() (*ql.ForwardRateAgreement. method*), 31

N

NPV() (*ql.ForwardRateAgreement. method*), 31

Q

ql.AmortizingFixedRateBond()
built-in function, 33
ql.AmortizingPayment()
built-in function, 8
ql.AnalyticBarrierEngine()
built-in function, 67
ql.AnalyticBinaryBarrierEngine()
built-in function, 68
ql.AnalyticCapFloorEngine()
built-in function, 56
ql.AnalyticContinuousGeometricAveragePriceAsianEngine()
built-in function, 64
ql.AnalyticContinuousGeometricAveragePriceAsianHestonEng
built-in function, 66
ql.AnalyticDiscreteGeometricAveragePriceAsianEngine()
built-in function, 64
ql.AnalyticDiscreteGeometricAveragePriceAsianHestonEngin
built-in function, 65
ql.AnalyticDoubleBarrierBinaryEngine()
built-in function, 69
ql.AnalyticDoubleBarrierEngine()
built-in function, 68
ql.AnalyticEuropeanEngine()
built-in function, 60
ql.AnalyticHestonEngine()
built-in function, 62
ql.AnalyticHestonForwardEuropeanEngine()
built-in function, 71
ql.AnalyticPTDHestonEngine()
built-in function, 63
ql.AndreasenHugeLocalVolAdapter()
built-in function, 92
ql.AndreasenHugeVolatilityAdapter()
built-in function, 90
ql.Array()
built-in function, 3
ql.AssetSwap()
built-in function, 37
ql.BachelierCapFloorEngine()
built-in function, 56
ql.BachelierSwaptionEngine()
built-in function, 58
ql.BarrierOption()
built-in function, 44
ql.BasketOption()

built-in function, 45
ql.BinomialBarrierEngine()
built-in function, 67
ql.BlackCallableFixedRateBondEngine()
built-in function, 55
ql.BlackCapFloorEngine()
built-in function, 56
ql.BlackCdsOptionEngine()
built-in function, 60
ql.BlackConstantVol()
built-in function, 88
ql.BlackDeltaCalculator()
built-in function, 54
ql.BlackIborCouponPricer()
built-in function, 12
ql.BlackKarasinski()
built-in function, 74
ql.BlackProcess()
built-in function, 78
ql.BlackScholesMertonProcess()
built-in function, 77
ql.BlackScholesProcess()
built-in function, 77
ql.BlackSwaptionEngine()
built-in function, 58
ql.BlackVarianceCurve()
built-in function, 89
ql.BlackVarianceSurface()
built-in function, 89
ql.BlackVolTermStructureHandle()
built-in function, 91
ql.Bond (*built-in class*), 32
ql.BondHelper()
built-in function, 109
ql.BondHelperVector()
built-in function, 109
ql.Calendar.holidayList()
built-in function, 23
ql.CallableFixedRateBond()
built-in function, 34
ql.Cap()
built-in function, 39
ql.CapFloorTermVolCurve()
built-in function, 93, 94
ql.CapFloorTermVolSurface()
built-in function, 94
ql.CapHelper()
built-in function, 110
ql.CappedFlooredCmsCoupon()
built-in function, 10
ql.CappedFlooredCoupon()
built-in function, 9
ql.CashFlows.atmRate()
built-in function, 13
ql.CashFlows.basisPointValue()
built-in function, 14
ql.CashFlows.bps()
built-in function, 13, 14
ql.CashFlows.convexity()
built-in function, 14
ql.CashFlows.duration()
built-in function, 14
ql.CashFlows.maturityDate()
built-in function, 13
ql.CashFlows.nextCashFlowDate()
built-in function, 13
ql.CashFlows.npv()
built-in function, 13, 14
ql.CashFlows.previousCashFlowDate()
built-in function, 13
ql.CashFlows.startDate()
built-in function, 13
ql.CashFlows.yieldRate()
built-in function, 14
ql.CashFlows.zSpread()
built-in function, 15
ql.CdsOption()
built-in function, 42
ql.CmsCoupon()
built-in function, 9
ql.CmsRateBond()
built-in function, 34
ql.CmsSpreadCoupon()
built-in function, 10
ql.Collar()
built-in function, 40
ql.ConstantOptionletVolatility()
built-in function, 93
ql.ConstantSwaptionVolatility()
built-in function, 95
ql.CPIBond()
built-in function, 40
ql.CreditDefaultSwap()
built-in function, 42
ql.CrossCurrencyBasisSwapRateHelper()
built-in function, 108
ql.Date()
built-in function, 20
ql.DatedOISRateHelper()
built-in function, 108
ql.DeltaVolQuote()
built-in function, 5
ql.DepositRateHelper()
built-in function, 101
ql.DerivedQuote()
built-in function, 5
ql.DiscountCurve()
built-in function, 83
ql.DiscountingBondEngine()
built-in function, 55
ql.DiscountingSwapEngine()
built-in function, 57
ql.DiscreteAveragingAsianOption()
built-in function, 43
ql.DoubleBarrierOption()
built-in function, 44

ql.Euribor()
built-in function, 27
ql.EuriborSwapIsdaFixA()
built-in function, 28
ql.ExtendedOrnsteinUhlenbeckProcess()
built-in function, 78
ql.FdBlackScholesAsianEngine()
built-in function, 65
ql.FdBlackScholesBarrierEngine()
built-in function, 67
ql.FdBlackScholesRebateEngine()
built-in function, 67
ql.FdBlackScholesVanillaEngine()
built-in function, 61
ql.FdG2SwaptionEngine()
built-in function, 58
ql.FdHestonBarrierEngine()
built-in function, 68
ql.FdHestonDoubleBarrierEngine()
built-in function, 69
ql.FdHestonVanillaEngine()
built-in function, 63
ql.FdHullWhiteSwaptionEngine()
built-in function, 58
ql.FittedBondDiscountCurve()
built-in function, 87
ql.FixedRateBond()
built-in function, 33
ql.FixedRateBondForward (*built-in class*), 32
ql.FixedRateBondHelper()
built-in function, 109
ql.FixedRateCoupon()
built-in function, 8
ql.FixedRateLeg()
built-in function, 11
ql.FlatForward()
built-in function, 83
ql.FlatHazardRate()
built-in function, 99
ql.FloatingRateBond()
built-in function, 33
ql.Floor()
built-in function, 40
ql.ForwardCurve()
built-in function, 84
ql.ForwardEuropeanEngine()
built-in function, 70
ql.ForwardRateAgreement (*built-in class*), 31
ql.ForwardSpreadedTermStructure()
built-in function, 86
ql.ForwardVanillaOption()
built-in function, 45
ql.FraRateHelper()
built-in function, 101–103
ql.FuturesRateHelper()
built-in function, 103, 104
ql.FxSwapRateHelper()
built-in function, 108
ql.G2()
built-in function, 75
ql.G2SwaptionEngine()
built-in function, 58
ql.GarmanKohlagenProcess()
built-in function, 79
ql.GaussianMultiPathGenerator()
built-in function, 52
ql.GaussianSobolMultiPathGenerator()
built-in function, 53
ql.GeneralizedBlackScholesProcess()
built-in function, 78
ql.GeometricBrownianMotionProcess()
built-in function, 77
ql.Gsr()
built-in function, 74
ql.HestonBlackVolSurface()
built-in function, 90
ql.HestonModel()
built-in function, 73
ql.HestonModelHelper()
built-in function, 111
ql.HestonProcess()
built-in function, 79
ql.HestonSLVProcess()
built-in function, 80
ql.HullWhite()
built-in function, 74
ql.HullWhiteForwardProcess()
built-in function, 81
ql.HullWhiteProcess()
built-in function, 81
ql.IborCoupon()
built-in function, 9
ql.IborIndex()
built-in function, 27
ql.IborLeg()
built-in function, 11
ql.IMM.code()
built-in function, 106
ql.IMM.isIMMcode()
built-in function, 106
ql.IMM.isIMMdate()
built-in function, 106
ql.IMM.nextCode()
built-in function, 106
ql.IMM.nextDate()
built-in function, 106
ql.ImpliedTermStructure()
built-in function, 86
ql.IntegralCdsEngine()
built-in function, 60
ql.InterestRate()
built-in function, 7
ql.IsdaCdsEngine()
built-in function, 59
ql.JamshidianSwaptionEngine()
built-in function, 59

ql.JointCalendar()
built-in function, 24
ql.LinearTsrPricer()
built-in function, 12
ql.LocalConstantVol()
built-in function, 91
ql.LocalVolSurface()
built-in function, 91
ql.MakeOIS()
built-in function, 38
ql.MakeSchedule()
built-in function, 26
ql.MakeVanillaSwap()
built-in function, 36
ql.Matrix()
built-in function, 4
ql.MCAmericanEngine()
built-in function, 61
ql.MCDigitalEngine()
built-in function, 62
ql.MCDiscreteArithmeticAPEngine()
built-in function, 65
ql.MCDiscreteArithmeticAPHestonEngine()
built-in function, 66
ql.MCDiscreteGeometricAPEngine()
built-in function, 64
ql.MCDiscreteGeometricAPHestonEngine()
built-in function, 66
ql.MCEuropeanBasketEngine()
built-in function, 69
ql.MCEuropeanEngine()
built-in function, 61
ql.MCEuropeanHestonEngine()
built-in function, 62
ql.MCForwardEuropeanBSEngine()
built-in function, 70
ql.MCForwardEuropeanHestonEngine()
built-in function, 71
ql.MidPointCdsEngine()
built-in function, 59
ql.Money()
built-in function, 17
ql.NoExceptLocalVolSurface()
built-in function, 92
ql.NonstandardSwap()
built-in function, 38
ql.OISRateHelper()
built-in function, 108
ql.OptionletStripper1()
built-in function, 95
ql.OptionletVolatilityStructureHandle()
built-in function, 95
ql.OvernightIndex()
built-in function, 27
ql.OvernightIndexedCoupon()
built-in function, 9
ql.OvernightIndexedSwap()
built-in function, 37
ql.OvernightIndexFutureRateHelper()
built-in function, 105
ql.OvernightLeg()
built-in function, 12
ql.Period()
built-in function, 21
ql.Piecewise()
built-in function, 85
ql.PiecewiseFlatHazardRate()
built-in function, 99
ql.PiecewiseTimeDependentHestonModel()
built-in function, 73
ql.PiecewiseYieldCurve()
built-in function, 85
ql.PiecewiseZeroInflation()
built-in function, 100
ql.RateHelperVector()
built-in function, 110
ql.Redemption()
built-in function, 8
ql.RelinkableBlackVolTermStructureHandle()
built-in function, 91
ql.RelinkableOptionletVolatilityStructureHandle()
built-in function, 95
ql.RelinkableSwaptionVolatilityStructureHandle()
built-in function, 98
ql.sabrFlochKennedyVolatility()
built-in function, 99
ql.SabrSmileSection()
built-in function, 98
ql.sabrVolatility()
built-in function, 98
ql.shiftedSabrVolatility()
built-in function, 98
ql.SimpleCashFlow()
built-in function, 8
ql.SimpleQuote()
built-in function, 5
ql.SofrFutureRateHelper()
built-in function, 105
ql.SpreadCdsHelper()
built-in function, 112
ql.SpreadedLinearZeroInterpolatedTermStructure()
built-in function, 87
ql.StochasticProcessArray()
built-in function, 81
ql.StrippedOptionletAdapter()
built-in function, 95
ql.SurvivalProbabilityCurve()
built-in function, 100
ql.Swap()
built-in function, 36
ql.SwapIndex()
built-in function, 28
ql.SwapRateHelper()
built-in function, 107
ql.Swaption()
built-in function, 39

ql.SwaptionHelper()
built-in function, 110, 111
ql.SwaptionVolatilityMatrix()
built-in function, 96
ql.SwaptionVolatilityStructureHandle()
built-in function, 98
ql.SwaptionVolCube2()
built-in function, 97
ql.TimeGrid()
built-in function, 26
ql.TreeCallableFixedRateBondEngine()
built-in function, 55, 56
ql.TreeCapFloorEngine()
built-in function, 57
ql.TreeSwaptionEngine()
built-in function, 59
ql.VanillaOption()
built-in function, 43
ql.VanillaSwap()
built-in function, 35
ql.Vasicek()
built-in function, 74
ql.YearOnYearInflationSwap()
built-in function, 42
ql.ZeroCouponBond()
built-in function, 33
ql.ZeroCouponInflationSwap()
built-in function, 41
ql.ZeroCouponInflationSwapHelper()
built-in function, 112
ql.ZeroCurve()
built-in function, 84
ql.ZeroSpreadedTermStructure()
built-in function, 86

R

RandomNumberGenerator()
built-in function, 50
rate, **18**
rate(), **18**

S

Schedule()
built-in function, 25
settlementDate() (*ql.ForwardRateAgreement. method*), 31
SobolRsg()
built-in function, 52
source, **18**
source(), **18**
spotIncome() (*ql.ForwardRateAgreement. method*), 31
spotValue() (*ql.ForwardRateAgreement. method*), 32
SwapSpreadIndex()
built-in function, 28

T

target, **18**
target(), **18**
type(), **18**

X

XXXInterpolation()
built-in function, 49