QuantLib.jl Documentation

Release 0.0.1

Chris Alexander

Nov 12, 2018

Contents

QuantLib.jl is a Julia package that provides a pure Julia version of the popular open-source quantitative finance library QuantLib.

This documentation is largely derived from QuantLib's documentation, with some alterations based on the Julia implementation.

Contents:

CHAPTER 1

Getting Started

Let's look at a few examples!

First, start off by importing QuantLib.jl

```
using QuantLib
```

1.1 Price a fixed rate bond

First, set up the global environment.

```
settlement_date = Date(2008, 9, 18) # construct settlement date
# settings is a global singleton that contains global settings
set_eval_date!(settings, settlement_date - Dates.Day(3))
```

Then, construct settings we will need to construct the yield curve

```
freq = QuantLib.Time.Semiannual()
tenor = QuantLib.Time.TenorPeriod(freq)
conv = QuantLib.Time.Unadjusted()
conv_depo = QuantLib.Time.ModifiedFollowing()
rule = QuantLib.Time.DateGenerationBackwards()
calendar = QuantLib.Time.USGovernmentBondCalendar()
dc_depo = QuantLib.Time.Actual365()
dc = QuantLib.Time.ISDAActualActual()
dc_bond = QuantLib.Time.ISMAActualActual()
fixing_days = 3
```

Build the Deposit and Bond helpers we will use to bootstrap the curve.

First we will set up vectors of the deposit rates and tenors, and bond issue dates and maturity dates (these could come from some other market data source)

```
# build depos
depo_rates = [0.0096, 0.0145, 0.0194]
depo_tens = [Base.Dates.Month(3), Base.Dates.Month(6), Base.Dates.Month(12)]

# build bonds
issue_dates = [Date(2005, 3, 15), Date(2005, 6, 15), Date(2006, 6, 30), Date(2002, 11,
↪ 15), Date(1987, 5, 15)]
mat_dates = [Date(2010, 8, 31), Date(2011, 8, 31), Date(2013, 8, 31), Date(2018, 8,␣
↪15), Date(2038, 5, 15)]
```

Now, we'll use some static coupon rates and market quotes for the bond helpers

```
coupon_rates = [0.02375, 0.04625, 0.03125, 0.04000, 0.04500]
market_quotes = [100.390625, 106.21875, 100.59375, 101.6875, 102.140625]
```

With this data, we can now construct our helpers

```
# construct the deposit and fixed rate bond helpers
insts = Vector{BootstrapHelper}(length(depo_rates) + length(issue_dates))
for i = eachindex(depo_rates)
  depo_quote = Quote(depo_rates[i])
  depo_tenor = QuantLib.Time.TenorPeriod(depo_tens[i])
  depo = DepositRateHelper(depo_quote, depo_tenor, fixing_days, calendar, conv_depo,␣
↪true, dc_depo)
  insts[i] = depo
end

for i = eachindex(coupon_rates)
  term_date = mat_dates[i]
  rate = coupon_rates[i]
  issue_date = issue_dates[i]
  market_quote = market_quotes[i]
  sched = QuantLib.Time.Schedule(issue_date, term_date, tenor, conv, conv, rule, true)
  bond = FixedRateBondHelper(Quote(market_quote), FixedRateBond(3, 100.0, sched, rate,
↪ dc_bond, conv,
                             100.0, issue_date, calendar, DiscountingBondEngine()))
  insts[i + length(depo_rates)] = bond
end
```

With our helpers created, we can start to construct the yield curve which we will bootstrap

```
interp = QuantLib.Math.LogLinear()
trait = Discount()
bootstrap = IterativeBootstrap()
yts = PiecewiseYieldCurve(settlement_date, insts, dc, interp, trait, 0.00000000001,␣
↪bootstrap)
```

Now, we can trigger the bootstrapping calculation (this can be triggered by a number of events, but for now we will just directly trigger calculation)

```
calculate!(yts)
```

Let's now create our fixed rate bond, by generating a coupon schedule and giving it a pricing engine

```
settlement_days = 3
face_amount = 100.0
```

(continues on next page)

(continued from previous page)

```
fixed_schedule = QuantLib.Time.Schedule(Date(2007, 5, 15), Date(2017, 5, 15),
            QuantLib.Time.TenorPeriod(QuantLib.Time.Semiannual()), QuantLib.Time.
↪Unadjusted(),
            QuantLib.Time.Unadjusted(), QuantLib.Time.DateGenerationBackwards(),␣
↪false,
            QuantLib.Time.USGovernmentBondCalendar())

pe = DiscountingBondEngine(yts)

fixedrate_bond = FixedRateBond(settlement_days, face_amount, fixed_schedule, 0.045,
              QuantLib.Time.ISMAActualActual(), QuantLib.Time.ModifiedFollowing(),␣
↪100.0,
              Date(2007, 5, 15), fixed_schedule.cal, pe)
```

Finally, we can request for the bond's NPV!

```
npv(fixedrate_bond) # 107.66828913260542
```

1.2 Calculate the Survival Probability of a Credit Default Swap

First, let's set up the environment

```
cal = QuantLib.Time.TargetCalendar()
todays_date = Date(2007, 5, 15)
settlementDate = todays_date
set_eval_date!(settings, todays_date)
```

Now, let's generate a flat-forward term structure for use with our CDS Helpers (which are used to generate the Hazard Rate Curve)

```
flatRate = Quote(0.01)

tsCurve = FlatForwardTermStructure(settlementDate, cal, flatRate, QuantLib.Time.
↪Actual365())
```

To bootstrap the hazard rate curve that we will use for survival probability (and inversely, default probability), we need to build CDS helpers. To begin, we'll set a recovery rate, and quote spreads, tenors, and maturity dates for 4 CDS helpers

```
recoveryRate = 0.5
quoteSpreads = [0.0150, 0.0150, 0.0150, 0.0150]
tenors = [Dates.Month(3), Dates.Month(6), Dates.Year(1), Dates.Year(2)]

maturities = [QuantLib.Time.adjust(cal, QuantLib.Following(), todays_date + ten) for␣
↪ten in tenors]
```

Let's build our CDS helpers

```
insts = SpreadCDSHelper[SpreadCDSHelper(Quote(quoteSpreads[i]), tenors[i], 0, cal,␣
↪QuantLib.Time.Quarterly(),
        QuantLib.Time.Following(), QuantLib.Time.DateGenerationTwentieth(), QuantLib.
↪Time.Actual365(),
        recoveryRate, tsCurve) for i in eachindex(tenors)]
```

With our helpers constructed, now we can build the hazard rate curve.

```
hazardRateStructure = PiecewiseDefaultCurve(todays_date, insts, QuantLib.Time.
↪Actual365(),
                      QuantLib.Math.BackwardFlatInterpolation(), HazardRate(), 1.0e-
↪12)
```

By requested for the curve nodes, we will trigger the bootstrap calculation

```
hr_curve_data = nodes(hazardRateStructure)
```

Now we can output the 1Y and 2Y survival probabilities

```
println(@sprintf("1Y Survival Probability: %.6f %%", survival_
↪probability(hazardRateStructure,
        todays_date + Dates.Year(1)) * 100.0))
println(@sprintf("2Y Survival Probability: %.6f %%", survival_
↪probability(hazardRateStructure,
        todays_date + Dates.Year(2)) * 100.0))
```

1.3 Price a Swaption Using a G2 Calibrated Model

Set up our environment

```
cal = QuantLib.Time.TargetCalendar()
settlementDate = Date(2002, 2, 19)
todays_date = Date(2002, 2, 15)
set_eval_date!(settings, todays_date)
```

Gather appropriate market data

```
swaptionMats = [Dates.Year(1), Dates.Year(2), Dates.Year(3), Dates.Year(4), Dates.
↪Year(5)]
swaptionVols = [0.1490, 0.1340, 0.1228, 0.1189, 0.1148, 0.1290, 0.1201, 0.1146, 0.
↪1108,
                0.1040, 0.1149, 0.1112, 0.1070, 0.1010, 0.0957, 0.1047, 0.1021, 0.
↪0980, 0.0951,
                0.1270, 0.1000, 0.0950, 0.0900, 0.1230, 0.1160]
swaptionLengths = [Dates.Year(1), Dates.Year(2), Dates.Year(3), Dates.Year(4), Dates.
↪Year(5)]
```

Generate a flat-forward term structure implying a 1x5 swap at 5%

```
flat_rate = Quote(0.04875825)
rhTermStructure = FlatForwardTermStructure(settlementDate, cal, flat_rate, QuantLib.
↪Time.Actual365())
```

Build an ATM swap

```
fixedLegFrequency = QuantLib.Time.Annual()
fixedLegConvention = QuantLib.Time.Unadjusted()
floatingLegConvention = QuantLib.Time.ModifiedFollowing()
fixedLegDayCounter = QuantLib.Time.EuroThirty360()
floatingLegFrequency = QuantLib.Time.Semiannual()
```

(continues on next page)

(continued from previous page)

```
swapType = Payer()
dummyFixedRate = 0.03
indexSixMonths = euribor_index(QuantLib.Time.TenorPeriod(Dates.Month(6)),␣
↪rhTermStructure)

startDate = QuantLib.Time.advance(Dates.Year(1), cal, settlementDate,␣
↪floatingLegConvention)
maturity = QuantLib.Time.advance(Dates.Year(5), cal, startDate, floatingLegConvention)

fixedSchedule = QuantLib.Time.Schedule(startDate, maturity, QuantLib.Time.
↪TenorPeriod(fixedLegFrequency),
                fixedLegConvention, fixedLegConvention, QuantLib.Time.
↪DateGenerationForwards(), false, cal)
floatSchedule = QuantLib.Time.Schedule(startDate, maturity, QuantLib.Time.
↪TenorPeriod(floatingLegFrequency),
                floatingLegConvention, floatingLegConvention, QuantLib.Time.
↪DateGenerationForwards(), false, cal)

swap = VanillaSwap(swapType, 1000.0, fixedSchedule, dummyFixedRate,␣
↪fixedLegDayCounter,
      indexSixMonths, 0.0, floatSchedule, indexSixMonths.dc,␣
↪DiscountingSwapEngine(rhTermStructure))

fixedATMRate = fair_rate(swap)

atmSwap = VanillaSwap(swapType, 1000.0, fixedSchedule, fixedATMRate,␣
↪fixedLegDayCounter, indexSixMonths,
          0.0, floatSchedule, indexSixMonths.dc,␣
↪DiscountingSwapEngine(rhTermStructure))
```

Construct our model

```
modelG2 = G2(rhTermStructure)
```

Build our calibration helpers

```
numRows = 5
numCols = 5

times = zeros(0)
swaptions = Vector{SwaptionHelper}(numRows)

for i = 1:numRows
  j = numCols - (i - 1)
  k = (i - 1) * numCols + j

  sh = SwaptionHelper(swaptionMats[i], swaptionLengths[j], Quote(swaptionVols[k]),␣
↪indexSixMonths,
        indexSixMonths.tenor, indexSixMonths.dc, indexSixMonths.dc, rhTermStructure,
        G2SwaptionEngine(modelG2, 6.0, 16))

  times = add_times_to!(sh, times)
  swaptions[i] = sh
end

tg = QuantLib.Time.TimeGrid(times, 30)
```

Calibrate our model

```
om = QuantLib.Math.LevenbergMarquardt()
calibrate!(modelG2, swaptions, om, QuantLib.Math.EndCriteria(400, 100, 1.0e-8, 1.0e-8,
↪ 1.0e-8))

for i=1:numRows
  j = numCols - (i - 1)
  k = (i - 1) * numCols + j

  npv = model_value!(swaptions[i])
  implied = implied_volatility!(swaptions[i], npv, 1e-4, 1000, 0.05, 0.50)
  diff = implied - swaptionVols[k]

  println(@sprintf("%i x %i: model %.5f%%, market: %.5f%% (%.5f%%)", i,␣
↪Int(swaptionLengths[j]), implied * 100,
          swaptionVols[k] * 100, diff * 100))
end

println("calibrated to: ")
println(@sprintf("a = %.6f, sigma = %.6f", get_params(modelG2)[1], get_
↪params(modelG2)[2]))
println(@sprintf("b = %.6f, eta = %.6f", get_params(modelG2)[3], get_
↪params(modelG2)[4]))
println(@sprintf("rho = %.6f", get_params(modelG2)[5]))
```

Build a Bermudan swaption for pricing

```
swapLeg = swap.legs[1] # Fixed Leg

bermudanDates = Vector{Date}(length(swapLeg.coupons))
for i=1:length(swapLeg.coupons)
bermudanDates[i]  = accrual_start_date(swapLeg.coupons[i])
end

bermudanExercise = BermudanExercise(bermudanDates)

bermudanSwaption = Swaption(atmSwap, bermudanExercise)
```

Use a tree swaption engine to price the swaption with our G2 model

```
bermudanSwaption = update_pricing_engine(bermudanSwaption, TreeSwaptionEngine(modelG2,
↪ 50))

println(@sprintf("G2 (tree):       %.6f", npv(bermudanSwaption)))
```

Use a finite-differences swaption engine to price the swaption with our G2 model

```
bermudanSwaption = update_pricing_engine(bermudanSwaption,␣
↪FdG2SwaptionEngine(modelG2))

println(@sprintf("G2 (fdm):       %.6f", npv(bermudanSwaption)))
```

CHAPTER 2

CashFlow Types and Functions

Types and methods to build and work with cashflows.

2.1 Basic CashFlow and Coupon Types and Methods

These are common methods for all coupons in QuantLib.jl. Coupon is an abstract type from which all other coupons are derived. A coupon itself is a type of cash flow.

2.1.1 Accessor methods

accrual_start_date(*c::Coupon*)
Returns the accrual start date of the coupon

accrual_end_date(*c::Coupon*)
Returns the accrual end date of the coupon

date_accrual_end(*c::Coupon*)
Returns the accrual end date of the coupon

ref_period_start(*c::Coupon*)
Returns the reference period start date of the coupon

ref_period_end(*c::Coupon*)
Returns the reference period end date of the coupon

get_dc(*c::Coupon*)
Returns the day counter of the coupon

accrual_period!(c::Coupon, val::Float64)
Sets the accrual period for a coupon

accrual_period(*c::Coupon*)
Returns the accrual period for the coupon

date (*c::Coupon*)
Returns the coupon's payment date

2.1.2 Simple Cash Flow

This is a basic cash flow type, typically used for redemptions

```
type SimpleCashFlow <: CashFlow
  amount::Float64
  date::Date
end
```

amount (*cf::SimpleCashFlow*)
Returns the simple cash flow amount

date (*cf::SimpleCashFlow*)
Returns the simple cash flow date

date_accrual_end (*cf::SimpleCashFlow*)
Returns the accrual end date of the simple cash flow (which is the date)

2.1.3 Dividend

This type is for any dividend cash flow

```
type Dividend <: CashFlow
  amount::Float64
  date::Date
end
```

amount (*cf::Dividend*)
Returns the dividend amount

date (*cf::Dividend*)
Returns the dividend date

date_accrual_end (*cf::Dividend*)
Returns the accrual end date of the dividend (which is the date)

2.1.4 Fixed Rate Coupon

```
type FixedRateCoupon{DC <: DayCount} <: Coupon
  couponMixin::CouponMixin{DC}
  paymentDate::Date
  nominal::Float64
  rate::InterestRate
end
```

This is a coupon for fixed rate cash flows

FixedRateCoupon{DC <: DayCount}(paymentDate::Date, faceAmount::Float64, rate::InterestRate,
Constructor for FixedRateCoupon

amount (*coup::FixedRateCoupon*)
Calculates value of coupon

calc_rate(coup::FixedRateCoupon) = coup.rate.rate
Returns the coupon rate

get_pay_dates (*coups::Vector{Union{FixedRateCoupon, SimpleCashFlow}}*)
Returns the pay dates as a vector of a vector of fixed rate coupons and simple cash flows (usually a redemption)

get_reset_dates (*coups::Vector{Union{FixedRateCoupon, SimpleCashFlow}}*)
Returns the reset dates as a vector of a vector of fixed rate coupons and simple cash flows (usually a redemption)

accrued_amount (*coup::FixedRateCoupon*, *settlement_date::Date*)
Calculates the accrued amount of a fixed rate coupon given a settlement date

2.1.5 Ibor Coupon

```
type IborCoupon{DC <: DayCount, X <: InterestRateIndex, ICP <: IborCouponPricer} <:
↪Coupon
  couponMixin::CouponMixin{DC}
  paymentDate::Date
  nominal::Float64
  fixingDate::Date
  fixingValueDate::Date
  fixingEndDate::Date
  fixingDays::Int
  iborIndex::X
  gearing::Float64
  spread::Float64
  isInArrears::Bool
  spanningTime::Float64
  pricer::ICP
end
```

IborCoupon (*paymentDate::Date*, *nominal::Float64*, *startDate::Date*, *endDate::Date*, *fixingDays::I*, *iborIndex::InterestRateIndex*, *gearing::Float64*, *spread::Float64*, *refPeriodStart::Date*, *refPeriodEnd::Date*, *dc::DayCount*, *isInArrears::Bool*, *pricer::IborCouponPricer*)
Constructor for IborCoupon

amount (*coup::IborCoupon*)
Calculates the Ibor coupon amount

accrued_amount (*coup::IborCoupon*, *settlement_date::Date*)
Calculates the accrued amount of the coupon given a settlement date

2.2 Cash Flow Legs

Cash Flow legs are basically holders of cash flows and coupons

2.2.1 General CashFlow Leg methods

npv (*leg::Leg*, *yts::YieldTermStructure*, *settlement_date::Date*, *npv_date::Date*)
Calculates the npv of a cash flow leg given a term structure, settlement date, and npv date

npv (*leg::Leg*, *y::InterestRate*, *include_settlement_cf::Bool*, *settlement_date::Date*, *npv_date::Date*)
Calculates the npv of a cash flow leg given an interest rate, settlement date, and npv date

npvbps (*leg::Leg*, *yts::YieldTermStructure*, *settlement_date::Date*, *npv_date::Date*, *includeSettlementDateFlows::Bool = true*)
Calculates the npv and bps of a cash flow leg given a term structure, settlement date, and npv date

duration (*::ModifiedDuration*, *leg::Leg*, *y::InterestRate*, *dc::DayCount*, *include_settlement_cf::Bool*, *settlement_date::Date*, *npv_date::Date = Date()*)
Calculates the modified duration of a cash flow leg given an interest rate, day count, settlement date, and npv date

previous_cashflow_date (*cf::Leg*, *settlement_date::Date*)
Returns the previous cash flow date of a cash flow leg given a settlement date

accrual_days (*cf::CashFlows*, *dc::DayCount*, *settlement_date::Date*)
Returns the number of accrued days of a cashflow given a day counter and a settlement date

next_cashflow (*cf::Leg*, *settlement_date::Date*)
Returns the next cash flow from a cash flow leg given a settlement date

accrued_amount (*cf::Leg*, *settlement_date::Date*, *include_settlement_cf::Bool = false*)
Calculates the accrued amount from a cash flow leg given a settlement date

has_occurred (*cf::CashFlow*, *ref_date::Date*, *include_settlement_cf::Bool = true*)
Determines whether or not a particular cash flow has occurred or not

maturity_date (*leg::Leg*)
Returns the maturity date of a cash flow leg

yield (*leg::Leg*, *npv::Float64*, *dc::DayCount*, *compounding::CompoundingType*, *freq::Frequency*, *include_settlement_cf::Bool*, *settlement_date::Date*, *npv_date::Date*, *accuracy::Float64*, *max_iter::Int*, *guess::Float64*)
Calculates the yield of a cash flow leg

2.2.2 Zero Coupon Leg

```
type ZeroCouponLeg <: Leg
  redemption::SimpleCashFlow
end
```

npv (*leg::ZeroCouponLeg*, *yts::YieldTermStructure*, *settlement_date::Date*, *npv_date::Date*)
Calculates the npv of a zero coupon cash flow leg given a term structure, settlement date, and npv date

npv (*leg::ZeroCouponLeg*, *y::InterestRate*, *include_settlement_cf::Bool*, *settlement_date::Date*, *npv_date::Date*)
Calculates the npv of a zero coupon cash flow leg given an interest rate, settlement date, and npv date

duration (*::ModifiedDuration*, *leg::ZeroCouponLeg*, *y::InterestRate*, *dc::DC*, *include_settlement_cf::Bool*, *settlement_date::Date*, *npv_date::Date = Date()*)
Calculates the modified duration of a zero coupon cash flow leg

2.2.3 Fixed Rate Leg

```
type FixedRateLeg <: Leg
  coupons::Vector{Union{FixedRateCoupon, SimpleCashFlow}}
end
```

FixedRateLeg (*schedule::Schedule*, *faceAmount::Float64*, *rate::Float64*, *calendar::BusinessCalendar*, *paymentConvention::BusinessDayConvention*, *dc::DayCount;* *add_redemption::Bool = true*)
Constructor for FixedRateLeg, passing in one rate

FixedRateLeg(*schedule::Schedule*, *faceAmount::Float64*, *rates::Vector{Float64}*, *calendar::BusinessCalendar*, *paymentConvention::BusinessDayConvention*, *dc::DayCount; add_redemption::Bool = false*)

Constructor for FixedRateleg, passing in a vector of rates

2.2.4 Ibor Leg

```
type IborLeg <: Leg
  coupons::Vector{Union{IborCoupon, SimpleCashFlow}}
end
```

IborLeg(*schedule::Schedule*, *nominal::Float64*, *iborIndex::InterestRateIndex*, *paymentDC::DayCount*, *paymentAdj::BusinessDayConvention*, *fixingDays::Vector{Int} = fill(iborIndex.fixingDays, length(schedule.dates) - 1)*, *gearings::Vector{Float64} = ones(length(schedule.dates) - 1)*, *spreads::Vector{Float64} = zeros(length(schedule.dates) - 1)*, *caps::Vector{Float64} = Vector{Float64}()*, *floors::Vector{Float64} = Vector{Float64}()*, *isInArrears::Bool = false*, *isZero::Bool = false*, *pricer::IborCouponPricer = BlackIborCouponPricer(); add_redemption::Bool = true*)

Constructor for Ibor Leg (will construct the coupons given the parameters passed in)

CHAPTER 3

Currencies

Derived from Ito.jl, the Currency type contains a variety of currencies worldwide, which are generated at compile time.

```
immutable Currency <: AbstractCurrency
  name::AbstractString
  code::AbstractString
  numeric::Int
  symbol::AbstractString
  fractionSymbol::AbstractString
  fractionsPerUnit::Int
  rounding::Function
  formatString::AbstractString
end
```

Example of usage:

```
EURCurrency() # Euro
USDCurrency() # USD
```

CHAPTER 4

Index Types and Functions

These types and methods help to construct indexes (e.g. Libor) and use them in instrument pricing.

4.1 General Interest Rate Index Methods

InterestRateIndex is the Abstract Type from which the Ibor and Libor indexes are derived. These are some general methods for all interest rate indexes.

fixing_date(*idx::InterestRateIndex*, *d::Date*)
Returns the fixing date of the index

value_date(*idx::InterestRateIndex*, *d::Date*)
Returns the value date of the index

fixing(*idx::InterestRateIndex*, *ts::TermStructure*, *_fixing_date::Date*, *forecast_todays_fixing::Bool=true*)
Returns the fixing of the index at a given date

forecast_fixing(*idx::InterestRateIndex*, *ts::TermStructure*, *_fixing_date::Date*)
Calculates a forecasted fixing for a future date, given a fixing date

forecast_fixing(*idx::InterestRateIndex*, *ts::TermStructure*, *d1::Date*, *d2::Date*, *t::Float64*)
Calculates a forecasted fixing for a future date given two dates and the time period between the two dates

is_valid_fixing_date(*idx::InterestRateIndex*, *d::Date*)
Determines whether or not a date is a valid fixing date for the index (namely, is it a valid business day)

add_fixing!(idx::InterestRateIndex, d::Date, fixingVal::Float64)
Adds a fixing and date to the index's cache of fixings

4.2 Ibor Index

```
immutable IborIndex <: InterestRateIndex
  familyName::AbstractString
  tenor::TenorPeriod
  fixingDays::Int
  currency::AbstractCurrency
  fixingCalendar::BusinessCalendar
  convention::BusinessDayConvention
  endOfMonth::Bool
  dc::DayCount
  ts::TermStructure
  pastFixings::Dict{Date, Float64}
end
```

IborIndex (*familyName::AbstractString*, *tenor::TenorPeriod*, *fixingDays::Int*, *currency::AbstractCurrency*, *fixingCalendar::BusinessCalendar*, *convention::BusinessDayConvention*, *endOfMonth::Bool*, *dc::DayCount*, *ts::TermStructure = NullTermStructure()*)

Constructor for the Ibor Index, will default to a NullTermStructure (which must be set later - this actually will clone this index with a new TS, since the type is immutable)

maturity_date (*idx::IborIndex*, *d::Date*)

Returns the maturity date of the index

4.3 Libor Index

```
immutable LiborIndex <: InterestRateIndex
  familyName::AbstractString
  tenor::TenorPeriod
  fixingDays::Int
  currency::Currency
  fixingCalendar::BusinessCalendar
  jointCalendar::JointCalendar
  convention::BusinessDayConvention
  endOfMonth::Bool
  dc::DayCount
  ts::TermStructure
end
```

LiborIndex (*familyName::AbstractString*, *tenor::TenorPeriod*, *fixingDays::Int*, *currency::Currency*, *fixingCalendar::BusinessCalendar*, *jointCalendar::JointCalendar*, *convention::BusinessDayConvention*, *endOfMonth::Bool*, *dc::DayCount*, *ts::TermStructure = NullTermStructure()*)

Default constructor for a Libor Index

LiborIndex (*familyName::AbstractString*, *tenor::TenorPeriod*, *fixingDays::Int*, *currency::Currency*, *fixingCalendar::BusinessCalendar*, *dc::DayCount*, *yts::YieldTermStructure*)

Additional constructor for a Libor Index, with no joint calendar or business day convention passed (the joint calendar is calculated)

value_date (*idx::LiborIndex*, *d::Date*)

Returns the value date of a libor index

maturity_date (*idx::LiborIndex*, *d::Date*)

Returns the maturity date of a libor index

4.4 Indexes Derived from Libor and Ibor

euribor_index (*tenor::TenorPeriod*)
Builds a Euribor Ibor index with a given time period (e.g. 6 month)

euribor_index (*tenor::TenorPeriod*, *ts::TermStructure*)
Builds a Euribor Ibor index with a given time period (e.g. 6 month) with a custom term structure

usd_libor_index (*tenor::TenorPeriod*, *yts::YieldTermStructure*)
Builds a USD Libor index with a given time period (e.g. 6 month) and term structure

4.5 Swap Index

```
immutable SwapIndex <: InterestRateIndex
  familyName::AbstractString
  tenor::TenorPeriod
  fixingDays::Int
  currency::Currency
  fixingCalendar::BusinessCalendar
  fixedLegTenor::Dates.Period
  fixedLegConvention::BusinessDayConvention
  fixedLegDayCount::DayCount
  discount::TermStructure
  iborIndex::IborIndex
  exogenousDiscount::Bool
  lastSwap::VanillaSwap
  lastFixingDate::Date
end
```

SwapIndex (*familyName::AbstractString*, *tenor::TenorPeriod*, *fixingDays::Int*, *currency::Currency*, *fixingCalendar::BusinessCalendar*, *fixedLegTenor::Dates.Period*, *fixedLegConvention::BusinessDayConvention*, *fixedLegDayCount::DayCount*, *discount::TermStructure*, *iborIndex::IborIndex*, *exogenousDiscount::Bool = true*)
Constructor for a Swap Index

CHAPTER 5

Instruments

These are some of the core types of QuantLib.jl. They involve the various instruments that QuantLib.jl is used to price, such as bonds, swaps, and options.

```
abstract Instrument <: LazyObject
```

5.1 Bonds

```
abstract Bond <: Instrument
```

5.1.1 Common Bond type and functions

```
type BondMixin
    settlementDays::Int
    issueDate::Date
    maturityDate::Date
end
```

The bond mixin provides a common interface for all bonds. These attributes are shared by all the bond types in QuantLib.jl

get_settlement_days (*bond::Bond*)
Returns the bond settlement days

get_issue_date (*bond::Bond*)
Returns the issue date of the bond

get_maturity_date (*bond::Bond*)
Returns the bond's maturity date

get_settlement_date (*b::Bond*)
Returns the bond's settlement date

notional (*bond::Bond*, *d::Date*)
Returns the bond's notional

accrued_amount (*bond::Bond*, *settlement::Date*)
Calculates the accrued amount of a bond's cashflows given a settlement date

maturity_date (*bond::Bond*)
Returns the maturity date of the bond

yield (*bond::Bond*, *clean_price::Float64*, *dc::DayCount*, *compounding::CompoundingType*, *freq::Frequency*, *settlement::Date*, *accuracy::Float64 = 1.0e-10*, *max_iter::Int = 100*, *guess::Float64 = 0.05*)
Calculates the bond yield, passing in an accuracy limit and iteration limit for the calculation

yield (*bond::Bond*, *dc::DayCount*, *compounding::CompoundingType*, *freq::Frequency*, *settlement::Date*, *accuracy::Float64 = 1.0e-10*, *max_iter::Int = 100*)
Calculates the bond yield as above, but with no clean price passed in (this gets calculated)

yield (*bond::Bond*, *dc::DayCount*, *compounding::CompoundingType*, *freq::Frequency*, *accuracy::Float64 = 1.0e-10*, *max_iter::Int = 100*)
Calculates the bond yield as above, but with no clean price or settlement date passed in

duration (*bond::Bond*, *yld::InterestRate*, *duration_::Duration*, *dc::DayCount*, *settlement_date::Date*)
Calculates the bond duration

duration (*bond::Bond*, *yld::Float64*, *dc::DayCount*, *compounding::CompoundingType*, *freq::Frequency*, *duration_::Duration*, *settlement_date::Date*)
Calculates the bond duration, with a rate passed in instead of an InterestRate object

npv (*bond::Bond*)
Calculates the bond NPV (this will trigger the bond calculation)

clean_price (*bond::Bond*, *settlement_value::Float64*, *settlement_date::Date*)
Calculates the bond clean price given a settlement value and settlement date

clean_price (*bond::Bond*)
Calculates the bond clean price (this will trigger calculation to get the settlement value)

dirty_price (*bond::Bond*, *settlement_value::Float64*, *settlement_date::Date*)
Calculates the bond dirty price given a settlement value and settlement date

dirty_price (*bond::Bond*)
Calculates the bond dirty price

settlement_date (*bond::Bond*, *d::Date = Date()*)
Returns the bond settlement date

5.1.2 Fixed Rate Bond

```
type FixedRateBond <: Bond
  lazyMixin::LazyMixin
  bondMixin::BondMixin
  faceAmount::Float64
  schedule::Schedule
  cashflows::FixedRateLeg
  dc::DayCount
  redemption::Float64
  startDate::Date
  pricingEngine::PricingEngine
  settlementValue::Float64
end
```

FixedRateBond(*settlementDays::Int*, *faceAmount::Float64*, *schedule::Schedule*, *coup_rate::Float64*, *dc::DayCount*, *paymentConvention::BusinessDayConvention*, *redemption::Float64*, *issueDate::Date*, *calendar::BusinessCalendar*, *pricing_engine::PricingEngine*)

Constructor for a Fixed Rate Bond

5.1.3 Floating Rate Bond

```
type FloatingRateBond <: Bond
  lazyMixin::LazyMixin
  bondMixin::BondMixin
  faceAmount::Float64
  schedule::Schedule
  cashflows::IborLeg
  iborIndex::InterestRateIndex
  dc::DayCount
  fixingDays::Int
  gearings::Vector{Float64}
  spreads::Vector{Float64}
  caps::Vector{Float64}
  floors::Vector{Float64}
  inArrears::Bool
  redemption::Float64
  pricingEngine::PricingEngine
  settlementValue::Float64
end
```

FloatingRateBond(*settlementDays::Int*, *faceAmount::Float64*, *schedule::Schedule*, *iborIndex::InterestRateIndex*, *dc::DayCount*, *convention::BusinessDayConvention*, *fixingDays::Int*, *issueDate::Date*, *pricingEngine::PricingEngine*, *inArrears::Bool = false*, *redemption::Float64 = 100.0*, *gearings::Vector{Float64} = ones(length(schedule.dates) - 1)*, *spreads::Vector{Float64} = zeros(length(schedule.dates) - 1)*, *caps::Vector{Float64} = Vector{Float64}()*, *floors::Vector{Float64} = Vector{Float64}()*; *cap_vol::OptionletVolatilityStructure=NullOptionletVolatilityStructure()*)

Constructor for a floating rate bond, optional argument to pass in an optionlet volatility term structure for the floating rate coupon pricer

5.1.4 Zero Coupon Bond

```
type ZeroCouponBond <: Bond
  lazyMixin::LazyMixin
  bondMixin::BondMixin
  faceAmount::Float64
  redemption::Float64
  cashflows::ZeroCouponLeg
  calendar::BusinessCalendar
  settlementValue::Float64
  pricingEngine::PricingEngine
end
```

ZeroCouponBond(*settlementDays::Int*, *calendar::BusinessCalendar*, *faceAmount::Float64*, *maturityDate::Date*, *paymentConvention::BusinessDayConvention=Following()*, *redemption::Float64=100.0*, *issueDate::Date=Date()*, *pe::PricingEngine = DiscountingBondEngine()*)

Constructor for a zero coupon bond

5.1.5 Callable Fixed Rate Bond

```
abstract AbstractCallableBond <: Bond

type CallableFixedRateBond <: AbstractCallableBond
  lazyMixin::LazyMixin
  bondMixin::BondMixin
  faceAmount::Float64
  schedule::Schedule
  cashflows::FixedRateLeg
  dc::DayCount
  redemption::Float64
  startDate::Date
  pricingEngine::PricingEngine
  settlementValue::Float64
  putCallSchedule::CallabilitySchedule
  blackEngine::PricingEngine
  blackVolQuote::Quote
end
```

CallableFixedRateBond(*settlementDays::Int*, *faceAmount::Float64*, *schedule::Schedule*, *coupons::Union{Vector{Float64}, Float64}*, *accrualDayCounter::DayCount*, *paymentConvention::BusinessDayConvention*, *redemption::Float64*, *issueDate::Date*, *putCallSchedule::CallabilitySchedule*, *pe::PricingEngine*)

Constructor for a callable fixed rate bond

5.1.6 Callability Schedule

```
type DirtyCall <: CallType end
type CleanCall <: CallType end

type Price
  amount::Float64
  callType::CallType
end

type Callability
  price::Price
  optionType::OptionType
  date::Date
end

typealias CallabilitySchedule Vector{Callability}
```

5.2 Forwards

```
abstract AbstractForward <: Instrument

type ForwardRateAgreement <: AbstractForward
  lazyMixin::LazyMixin
  underlyingIncome::Float64
  underlyingSpotValue::Float64
  dc::DayCount
```

(continues on next page)

(continued from previous page)

```
  calendar::BusinessCalendar
  convention::BusinessDayConvention
  settlementDays::Int
  payoff::ForwardTypePayoff
  valueDate::Date
  maturityDate::Date
  discountCurve::YieldTermStructure
  fraType::PositionType
  forwardRate::InterestRate
  strikeForwardRate::InterestRate
  notionalAmount::Float64
  iborIndex::IborIndex
end
```

ForwardRateAgreement (*valueDate::Date*, *maturityDate::Date*, *position::PositionType*, *strikeForward::Float64*, *notionalAmount::Float64*, *iborIndex::IborIndex*, *discountCurve::YieldTermStructure*)
Constructor for a forward rate agreement

spot_value (*fra::ForwardRateAgreement*)
Calculates the spot value of a forward rate agreement (triggers calculation)

forward_value (*fra::ForwardRateAgreement*)
Calculates the forward value of a forward rate agreement (triggers calculation)

forward_rate (*fra::ForwardRateAgreement*)
Calculates the forward rate of a forward rate agreement (triggers calculation)

implied_yield (*fra::ForwardRateAgreement*, *underlyingSpotValue::Float64*, *forwardValue::Float64*, *settlementDate::Date*, *comp::CompoundingType*, *dc::DayCount*)
Calculates the implied yield of a forward rate agreement (triggers calculation)

5.3 Options

```
abstract Option{E} <: Instrument
abstract OneAssetOption{E} <: Option{E}

typealias EuropeanOption Option{EuropeanExercise}
typealias AmericanOption Option{AmericanExercise}
typealias BermudanOption Option{BermudanExercise}

type VanillaOption <: OneAssetOption{Exercise}
  lazyMixin::LazyMixin
  payoff::StrikedTypePayoff
  exercise::Exercise
  pricingEngine::PricingEngine
  results::OptionResults
end

# These are the results that are calculated when the option is priced
type OptionResults # Greeks
  delta::Float64
  gamma::Float64
  theta::Float64
  vega::Float64
```

(continues on next page)

(continued from previous page)

```
  rho::Float64
  dividendRho::Float64
  deltaForward::Float64
  elasticity::Float64
  thetaPerDay::Float64
  strikeSensitivity::Float64
  itmCashProbability::Float64
  value::Float64
end
```

VanillaOption (*payoff::StrikedTypePayoff*, *exercise::Exercise*, *pe::PricingEngine*)
Constructor for a vanilla option

5.4 Payoffs

```
abstract AbstractPayoff
abstract StrikedTypePayoff <: AbstractPayoff

type Put <: OptionType end
type Call <: OptionType end

type PlainVanillaPayoff <: StrikedTypePayoff
  optionType::OptionType
  strike::Float64
end
```

5.5 Swaps

```
abstract Swap <: Instrument
```

5.5.1 Vanilla Swaps

```
type Payer <: SwapType end
type Receiver <: SwapType end

# these are the results that are calculated when the swap is priced
type SwapResults <: Results
  legNPV::Vector{Float64}
  legBPS::Vector{Float64}
  npvDateDiscount::Float64
  startDiscounts::Vector{Float64}
  endDiscounts::Vector{Float64}
  fairRate::Float64
  value::Float64
end

type VanillaSwap <: Swap
  lazyMixin::LazyMixin
  swapT::SwapType
```

(continues on next page)

(continued from previous page)

```
  nominal::Float64
  fixedSchedule::Schedule
  fixedRate::Float64
  fixedDayCount::DayCount
  iborIndex::IborIndex
  spread::Float64
  floatSchedule::Schedule
  floatDayCount::DayCount
  paymentConvention::BusinessDayConvention
  legs::Vector{Leg}
  payer::Vector{Float64}
  pricingEngine::PricingEngine
  results::SwapResults
  args::VanillaSwapArgs
end
```

VanillaSwap (*swapT::SwapType*, *nominal::Float64*, *fixedSchedule::Schedule*, *fixedRate::Float64*, *fixedDayCount::DayCount*, *iborIndex::IborIndex*, *spread::Float64*, *floatSchedule::Schedule*, *floatDayCount::DayCount*, *pricingEngine::PricingEngine = NullPricingEngine()*, *paymentConvention::B = floatSchedule.convention*)

Constructor for a vanilla swap

fair_rate (*swap::VanillaSwap*)

Calculates the fair rate of a vanilla swap (triggers calculation)

5.5.2 Nonstandard Swap

This swap is used in all Gaussian Short Rate model calculations

```
type NonstandardSwap <: Swap
    lazyMixin::LazyMixin
    swapT::SwapType
    fixedNominal::Vector{Float64}
    floatingNominal::Vector{Float64}
    fixedSchedule::Schedule
    fixedRate::Vector{Float64}
    fixedDayCount::DayCount
    iborIndex::IborIndex
    spread::Float64
    gearing::Float64
    floatSchedule::Schedule
    floatDayCount::DayCount
    paymentConvention::BusinessDayConvention
    intermediateCapitalExchange::Bool
    finalCapitalExchange::Bool
    legs::Vector{Leg}
    payer::Vector{Float64}
    pricingEngine::PricingEngine
    results::SwapResults
    args::NonstandardSwapArgs
end
```

NonstandardSwap (*vs::VanillaSwap*)

Constructor for a Nonstandard Swap

5.5.3 Credit Default Swap

CreditDefaultSwap(*side::CDSProtectionSide*, *notional::Float64*, *spread::Float64*, *schedule::Schedule*, *convention::BusinessDayConvention*, *dc::DayCount*, *settlesAccrual::Bool*, *paysAtDefaultTime::Bool*, *protectionStart::Date*, *pricingEngine::PricingEngine*)
Constructor for a credit default swap

5.6 Swaptions

```
type SettlementPhysical <: SettlementType end
type SettlementCash <: SettlementType end

type SwaptionResults{S <: AbstractString}
  value::Float64
  additionalResults::Dict{S, Float64}
end

type Swaption <: Option
  lazyMixin::LazyMixin
  swap::VanillaSwap
  exercise::Exercise
  delivery::SettlementType
  results::SwaptionResults
  pricingEngine::PricingEngine
end
```

Swaption(*swap::VanillaSwap*, *exercise::Exercise*)
Constructor for a swaption with no pricing engine or settlement type set

Swaption(*swap::VanillaSwap*, *exercise::Exercise*, *pe::PricingEngine*)
Constructor for a swaption with no settlement type set

5.6.1 Nonstandard Swaptions

These swaptions are used in Gaussian Short Rate model calculations

```
type NonstandardSwaption <: Option
  lazyMixin::LazyMixin
  swap::NonstandardSwap
  exercise::Exercise
  delivery::SettlementType
  results::SwaptionResults
  pricingEngine::PricingEngine
end
```

NonstandardSwaption(*swap::NonstandardSwap*, *exercise::Exercise*)
Constructor for a nonstandard swaption with no pricing engine or settlement type set

NonstandardSwaption(*swap::NonstandardSwap*, *exercise::Exercise*, *pe::PricingEngine*)
Constructor for a nonstandard swaption with no settlement type set

CHAPTER 6

Interest Rates

Concrete Interest Rate type

```
type InterestRate
  rate::Float64
  dc::DayCount
  comp::CompoundingType
  freq::Frequency
end
```

discount_factor(*ir::InterestRate*, *time_frac::Float64*)
Discount factor implied by the rate compounded at time time_frac

discount_factor(*ir::InterestRate*, *date1::Date*, *date2::Date*, *ref_start::Date = Date()*, *ref_end::Date = Date()*)
Discount factor implied by the rate compounded between two dates

compound_factor(*ir::InterestRate*, *time_frac::Float64*)
Compound factor implied by the rate compounded at time time_frac

compound_factor(*ir::InterestRate*, *date1::Date*, *date2::Date*, *ref_start::Date = Date()*, *ref_end::Date = Date()*)
Compound factor implied by the rate compounded between two dates

equivalent_rate(*ir::InterestRate*, *comp::CompoundingType*, *freq::Frequency*, *time_frac::Float64*)
Equivalent interest rate for a compounding period time_frac. The resulting InterestRate shares the same implicit day-counting rule of the original InterestRate instance

equivalent_rate(*ir::InterestRate*, *result_dc::DayCount*, *comp::CompoundingType*, *freq::Frequency*, *date1::Date*, *date2::Date*, *ref_start::Date = Date()*, *ref_end::Date = Date()*)
Equivalent rate for a compounding period between two dates. The resulting rate is calculated taking the required day-counting rule into account

implied_rate(*compound::Float64*, *dc::DayCount*, *comp::CompoundingType*, *time_frac::Float64*, *freq::Frequency*)
Implied interest rate for a given compound factor at a given time. The resulting InterestRate has the day-counter provided as input

implied_rate(*compound::Float64*, *dc::DayCount*, *comp::CompoundingType*, *date1::Date*, *date2::Date*, *freq::Frequency*, *ref_start::Date = Date()*, *ref_end::Date = Date()*)

Implied rate for a given compound factor between two dates. The resulting rate is calculated taking the required day-counting rule into account

6.1 Compounding Types

```
abstract CompoundingType

type ContinuousCompounding <: CompoundingType end # exp(r * t)
type SimpleCompounding <: CompoundingType end     # (1+r*t)
type CompoundedCompounding <: CompoundingType end # (1 + r)^t
type SimpleThenCompounded <: CompoundingType end  # Simple up to the first period
↪then Compounded
```

6.2 Duration

```
abstract Duration

type ModifiedDuration <: Duration end
```

CHAPTER 7

Math Module

This is a sub-module in QuantLib.jl that has math-related types and methods. It includes various interpolation methods, solvers, and optimization methods.

7.1 General Math methods

is_close{T <: Number}(x::T, y::T, n::Int = 42)
Determines whether two numbers are almost equal

close_enough{T <: Number}(x::T, y::T, n::Int = 42)
Determines whether two numbers are almost equal but with slightly looser criteria than is_close

bounded_log_grid(*xMin::Float64*, *xMax::Float64*, *steps::Int*)
Bounded log grid constructor

7.1.1 Bernstein Polynomial

Bernstein Polynomial type

```
type BernsteinPolynomial end
```

get_polynomial(*::BernsteinPolynomial*, *i::Int*, *n::Int*, *x::Float64*)
Build and return a Bernstein polynomial

7.2 Interpolation

QuantLib.jl provides various interpolation methods for building curves

```
abstract Interpolation
abstract Interpolation2D <: Interpolation
```

update!(interp::Interpolation, idx::Int, val::Float64)
Updates the y_vals of an interpolation with a given value at a given index

7.2.1 Linear Interpolation

```
type LinearInterpolation <: Interpolation
  x_vals::Vector{Float64}
  y_vals::Vector{Float64}
  s::Vector{Float64}
end
```

initialize!(interp::LinearInterpolation, x_vals::Vector{Float64}, y_vals::Vector{Float64})
Initializes the linear interpolation with a given set of x values and y values

update!(interp::LinearInterpolation, idx::Int)
Updates the linear interpolation from a given index

update!(interp::LinearInterpolation)
Updates the linear interpolation from the first index

value(*interp::LinearInterpolation*, *val::Float64*)
Returns the interpolated value

derivative(*interp::LinearInterpolation*, *val::Float64*)
Returns the derivative of the interpolated value

7.2.2 Log Interpolation

Log interpolation between points - this must be associated with another interpolation method (e.g. Linear).

```
type LogInterpolation <: Interpolation
  x_vals::Vector{Float64}
  y_vals::Vector{Float64}
  interpolator::Interpolation
end

typealias LogLinearInterpolation LogInterpolation{LinearInterpolation}
```

LogLinear(*x_vals::Vector{Float64}*, *y_vals::Vector{Float64}*)
Constructor for a log linear interpolation

LogLinear()
Construct for a log linear interpolation with no initial values provided

initialize!(interp::LogInterpolation, x_vals::Vector{Float64}, y_vals::Vector{Float64})
Initialize a log interpolation and its interpolator with a given set of x and y values

update!(interp::LogInterpolation, idx::Int)
Updates a log interpolation and its interpolator from a given index

update!(interp::LogInterpolation)
Updates a log interpolation and its interpolator from the first index

value(*interp::LogInterpolation*, *val::Float64*)
Returns the interpolated value

derivative(*interp::LogInterpolation*, *val::Float64*)
Returns the derivative of the interpolated value

7.2.3 Spline Interpolation

A base cubic interpolation type is provided, with just cubic spline interpolation provided at this time.

CubicInterpolation(*dApprox::DerivativeApprox*, *leftBoundary::BoundaryCondition*, *rightBoundary::BoundaryCondition*, *x_vals::Vector{Float64}*, *y_vals::Vector{Float64}*, *leftValue::Float64 = 0.0*, *rightValue::Float64 = 0.0*, *monotonic::Bool = true*)
Constructor for any cubic interpolation type

value(*interp::CubicInterpolation*, *x::Float64*)
Returns the interpolated value

QuantLib.jl also has a Bicubic spline type, using the Dierckx library.

```
type BicubicSpline
  spline::Dierckx.Spline2D
end
```

7.2.4 Backward-Flat Interpolation

A backward-flat interpolation between points

```
type BackwardFlatInterpolation <: Interpolation
  x_vals::Vector{Float64}
  y_vals::Vector{Float64}
  primitive::Vector{Float64}
end
```

BackwardFlatInterpolation()
Constructor for a backward-flat interpolation - no passed in values

initialize!(interp::BackwardFlatInterpolation, x_vals::Vector{Float64}, y_vals::Vector{Flo
Initializes the backward-flat interpolation with a set of x and y values

update!(interp::BackwardFlatInterpolation, idx::Int)
Updates the backward-flat interpolation from a given index

update!(interp::BackwardFlatInterpolation)
Updates the backward-flat interpolation from the first index

value(*interp::BackwardFlatInterpolation*, *x::Float64*)
Returns the interpolated value

get_primitive(*interp::BackwardFlatInterpolation*, *x::Float64*, *::Bool*)
Returns the primative of the interpolated value

7.3 Optimization

QuantLib.jl has various optimization methods for root finding and other calculations

7.3.1 General Optimization types and methods

```
abstract OptimizationMethod
abstract CostFunction
abstract Constraint
```

OptimizationMethod is the abstract base type. The CostFunction abstract type is the base type for any function passed to an optimization method. And the Constraint abstract type provides constraints for the optimization method.

Projection

```
type Projection
  actualParameters::Vector{Float64}
  fixedParameters::Vector{Float64}
  fixParams::BitArray
  numberOfFreeParams::Int
end
```

The Projection type provides a data structure for actual and fixed parameters for any cost function.

project (*proj::Projection*, *params::Vector{Float64}*)
Returns the subset of free parameters corresponding to set of parameters

include_params (*proj::Projection*, *params::Vector{Float64}*)
Returns whole set of parameters corresponding to the set of projected parameters

Constraints:

```
type NoConstraint <: Constraint end
type PositiveConstraint <: Constraint end
type BoundaryConstraint <: Constraint
  low::Float64
  high::Float64
end

type ProjectedConstraint{C <: Constraint} <: Constraint
  constraint::C
  projection::Projection
end
```

Various constraint types used in optimization methods

End Criteria:

```
type EndCriteria
  maxIterations::Int
  maxStationaryStateIterations::Int
  rootEpsilon::Float64
  functionEpsilon::Float64
  gradientNormEpsilon::Float64
end
```

This type provides the end criteria for an optimization method.

Problem:

```
type Problem{T}
  costFunction::CostFunction
  constraint::Constraint
  initialValue::Vector{T}
  currentValue::Vector{T}
  functionValue::Float64
  squaredNorm::Float64
  functionEvaluation::Int
  gradientEvaluation::Int
end
```

Data structure for a constrained optimization problem.

value!{T}(p::Problem, x::Vector{T})
Calls cost function computation and increment evaluation counter

values!(p::Problem, x::Vector{Float64})
Calls cost values computation and increment evaluation counter

7.3.2 Levenberg Marquardt

This is QuantLib.jl's Levenberg Marquardt optimization method

LevenbergMarquardt()
Base constructor for the Levenberg Marquardt optimization method

minimize!(lm::LevenbergMarquardt, p::Problem, endCriteria::EndCriteria)
Minimization method for the Levenberg Marquardt optimization method

lmdif!(m::Int, n::Int, x::Vector{Float64}, fvec::Vector{Float64}, ftol::Float64, xtol::Floa
Lmdif function from the MINPACK minimization routine, which has been rewritten in Julia for QuantLib.jl

7.3.3 Simplex

This is QuantLib.jl's Simplex optimization method

minimize!(simplex::Simplex, p::Problem, end_criteria::EndCriteria)
Minimization method for the simplex optimization method

7.4 Solvers

QuantLib.jl also has several solvers available to find x such that f(x) == 0.

7.4.1 General solver methods

These solve methods will call an underlying solve method based on what type of solver you are using.

solve(*solver::Solver1D*, *f::Function*, *accuracy::Float64*, *guess::Float64*, *step::Float64*)
General solve method given a guess and step (bounds enforcement is calculated)

solve(*solver::Solver1D*, *f::Function*, *accuracy::Float64*, *guess::Float64*, *xMin::Float64*, *xMax::Float64*)
General solve method given a guess, min, and max.

Mixin shared by all solvers:

```
type SolverInfo
  maxEvals::Int
  lowerBoundEnforced::Bool
  upperBoundEnforced::Bool
  lowerBound::Float64
  upperBound::Float64
end
```

7.4.2 Brent Solver

```
type BrentSolver <: Solver1D
  solverInfo::SolverInfo
end
```

BrentSolver(*maxEvals::Int = 100*, *lowerBoundEnforced::Bool = false*, *upperBoundEnforced::Bool = false*, *lowerBound::Float64 = 0.0*, *upperBound::Float64 = 0.0*)
Constructor for a brent solver, with defaults

7.4.3 Finite Differences Solver

```
type FiniteDifferenceNewtonSafe <: Solver1D
  solverInfo::SolverInfo
end
```

FiniteDifferenceNewtonSafe(*maxEvals::Int = 100*, *lowerBoundEnforced::Bool = false*, *upperBoundEnforced::Bool = false*, *lowerBound::Float64 = 0.0*, *upperBound::Float64 = 0.0*)
Constructor for a finite differences newton safe solver, with defaults

7.4.4 Newton Solver

```
type NewtonSolver <: Solver1D
  solverInfo::SolverInfo
end
```

NewtonSolver(*maxEvals::Int = 100*, *lowerBoundEnforced::Bool = false*, *upperBoundEnforced::Bool = false*, *lowerBound::Float64 = 0.0*, *upperBound::Float64 = 0.0*)
Constructor for a newton solver, with defaults

7.4.5 General Linear Least Squares

```
type GeneralLinearLeastSquares
  a::Vector{Float64}
  err::Vector{Float64}
  residuals::Vector{Float64}
  standardErrors::Vector{Float64}
end
```

GeneralLinearLeastSquares{T}(x::Vector{Float64}, y::Vector{Float64}, v::Vector{T})
Constructor for the general linear least squares solver

get_coefficients(glls::GeneralLinearLeastSquares) = glls.a
Returns the coefficients from the general linear least squares calculation

7.5 Distributions

QuantLib.jl largely uses the StatsFuns.jl and Distributions.jl packages for distribution functionality, but we have added a couple additional methods that are necessary for pricing.

distribution_derivative (*w::Normal*, *x::Float64*)
Returns the distribution derivative from a normal distribution

peizer_pratt_method_2_inversion (*z::Float64*, *n::Int*)
Given an odd integer n and real number z, returns p such that: 1 - CumulativeBinomialDistribution((n-1) / 2, n, p) = CumulativeNormalDistribution(z)

7.6 Integration

QuantLib.jl has various integration methods available. All are derived from an abstract type Integrator

Any integrator has a base "call" method (in Julia, you can make types callable), defined as follows:

```
function call(integrator::Integrator, f::IntegrationFunction, a::Float64, b::Float64)
  integrator.evals = 0
  if a == b
    return 0.0
  end

  if (b > a)
    return integrate(integrator, f, a, b)
  else
    return -integrate(integrator, f, b, a)
  end
end
```

7.6.1 Gauss Laguerre Integration

Performs a one-dimensional Gauss-Laguerre integration.

```
abstract GaussianQuadrature

type GaussLaguerreIntegration <: GaussianQuadrature
  x::Vector{Float64}
  w::Vector{Float64}
end
```

The Guass Laguerre integration has its own call method:

call (*gli::GaussLaguerreIntegration*, *f::IntegrationFunction*)
This method is called like this, given an instance of the GaussLaguerreIntegration myGLI: myGLI(f) where f is your IntegrationFunction

GaussLaguerreIntegration (*n::Int*, *s::Float64 = 0.0*)
Constructor for Gauss Laguerre Integration

get_order (*integration::GaussianQuadrature*)
Returns the order of the gaussian quadrature

7.6.2 Segment Interval

```
type SegmentIntegral <: Integrator
  absoluteAccuracy::Float64
  absoluteError::Float64
```

(continues on next page)

(continued from previous page)

```
  maxEvals::Int
  evals::Int
  intervals::Int
end
```

SegmentIntegral (*intervals::Int*)
Constructor for the segment tntegral

Math.**integrate** (*integrator::SegmentIntegral*, *f::Union{Function, IntegrationFunction}*, *a::Float64*, *b::Float64*)
Integrator method for the segment integral

7.7 Matrices

Julia has a plethora of matrix-related methods, so we have just added a few specific additional ones needed by various QuantLib.jl calculations

Some specific types used for rank calculation

```
abstract SalvagingAlgo
type NoneSalvagingAlgo <: SalvagingAlgo end
```

rank_reduced_sqrt (*matrix::Matrix*, *maxRank::Int*, *componentRetainedPercentage::Float64*, *sa::NoneSalvagingAlgo*)
Returns the rank-reduced pseudo square root of a real symmetric matrix. The result matrix has rank<=maxRank. If maxRank>=size, then the specified percentage of eigenvalues out of the eigenvalues' sum is retained. If the input matrix is not positive semi definite, it can return an approximation of the pseudo square root using a (user selected) salvaging algorithm. The given matrix must be symmetric.

7.7.1 Orthogonal Projection

```
type OrthogonalProjection
  originalVectors::Matrix{Float64}
  multiplierCutoff::Float64
  numberVectors::Int
  numberValidVectors::Int
  dimension::Int
  validVectors::BitArray{1}
  projectedVectors::Vector{Vector{Float64}}
  orthoNormalizedVectors::Matrix{Float64}
end
```

OrthogonalProjection (*originalVectors::Matrix{Float64}*, *multiplierCutoff::Float64*, *tolerance::Float64*)
Given a collection of vectors, w_i, find a collection of vectors x_i such that x_i is orthogonal to w_j for i != j, and <x_i, w_i> = <w_i, w_i>. This is done by performing GramSchmidt on the other vectors and then projecting onto the orthogonal space.

get_vector (*op::OrthogonalProjection*, *i::Int*)
Returns the projected vector of a given index

7.8 Random Number Generation

We use Julia's built in MersenneTwister RNG for random number generation (RNG), and we have defined types to generate sequences (RSG) based on a few different methods. We use the StatsFuns and Sobol Julia packages here.

All sequence generators are derived from:

```
abstract AbstractRandomSequenceGenerator
```

Some general methods:

init_sequence_generator!(rsg::AbstractRandomSequenceGenerator, dimension::Int)
Initializes a RSG with a given dimension (the MersenneTwister, if used, is init-ed with a seed upon type instantiation)

last_sequence(*rsg::AbstractRandomSequenceGenerator*)
Returns the last sequence generated (the last sequence generated is always cached)

7.8.1 Pseudo Random RSG

This is the most basic random sequence generator. It simply generates a random sequence based on a set dimension from the Mersenne Twister RNG.

```
type PseudoRandomRSG <: AbstractRandomSequenceGenerator
  rng::MersenneTwister
  dimension::Int
  values::Vector{Float64}
  weight::Float64
end
```

PseudoRandomRSG(*seed::Int*, *dimension::Int = 1*, *weight::Float64 = 1.0*)
Constructor for the Pseudo Random RSG, defaults to a sequence length of 1 and weight of 1

next_sequence!(rsg::PseudoRandomRSG)
Builds and returns the next sequence (returns a tuple of the sequence and the weight)

7.8.2 Inverse Cumulative RSG

The random numbers in this RSG are generated by the Mersenne Twister and manipulated by the inverse CDF of a normal distribution.

```
type InverseCumulativeRSG <: AbstractRandomSequenceGenerator
  rng::MersenneTwister
  dimension::Int
  values::Vector{Float64}
  weight::Float64
end
```

InverseCumulativeRSG(*seed::Int*, *dimension::Int = 1*, *weight::Float64 = 1.0*)
Constructor for the Inverse Cumulative RSG, defaulting to a sequence length of 1 and weight of 1

next_sequence!(rsg::InverseCumulativeRSG)
Builds and returns the next sequence (returns a tuple of the sequence and the weight)

7.8.3 Sobol RSG

This RSG uses the Julia package Sobol to generate sequences from the Sobol RNG.

```
type SobolRSG <: AbstractRandomSequenceGenerator
  rng::Sobol.SobolSeq
  dimension::Int
  values::Vector{Float64}
  weight::Float64
end
```

SobolRSG (*dimension::Int = 1*, *weight::Float64 = 1.0*)
Constructor for the Sobol RSG, defaulting to a sequence length of 1 and weight of 1 (no seed required)

next_sequence!(rsg::SobolRSG)
Builds and returns the next sequence (returns a tuple of the sequence and the weight)

7.8.4 Sobol Inverse Cumulative RSG

This RSG uses the Julia package Sobol to generate Sobol sequences, which are then manipulated via the inverse CDF of a normal distribution.

```
type SobolInverseCumulativeRSG <: AbstractRandomSequenceGenerator
  rng::Sobol.SobolSeq
  dimension::Int
  values::Vector{Float64}
  weight::Float64
end
```

SobolInverseCumulativeRSG (*dimension::Int = 1*, *weight::Float64 = 1.0*)
Constructor for the Sobol inverse cumulative RSG

next_sequence!(rsg::SobolInverseCumulativeRSG)
Builds and returns the next sequence (returns a tuple of the sequence and the weight)

7.9 Sampled Curve

Contains a sampled curve. Initially, the structure will contain one indexed curve.

```
type SampledCurve
  grid::Vector{Float64}
  values::Vector{Float64}
end
```

SampledCurve (*gridSize::Int*)
Constructor for a sampled curve with a specified grid size

SampledCurve ()
Constructor for a sampled curve with no specified grid size

get_size (*curve::SampledCurve*)
Returns the size of the sampled curve grid

set_log_grid!(curve::SampledCurve, min::Float64, max::Float64)
Builds a log grid and sets it as the sampled curve's grid

set_grid!(curve::SampledCurve, grid::Vector{Float64})
Sets the sampled curve grid

Math.sample!{T}(curve::SampledCurve, f::T)
Samples from the curve given a passed in function (or something callable)

value_at_center(*curve::SampledCurve*)
Calculates the value at the center of the sampled curve

first_derivative_at_center(*curve::SampledCurve*)
Calculates the first derivative at the center of the sampled curve

second_derivative_at_center(*curve::SampledCurve*)
Calculates the second derivative at the center of the sampled curve

7.10 Statistics

These types and methods provide statistical functionality, such as storing data (and weighted data) and performing basic stats calculations (mean, std dev, skewness, etc). Basic stats functions are provided by StatsBase

```
abstract AbstractStatistics

abstract StatsType
type GaussianStatsType <: StatsType end
```

7.10.1 Basic Stats Methods

error_estimate(*stat::AbstractStatistics*)
Calculates the error estimate of the samples

weight_sum(*stat::AbstractStatistics*)
Calculates the sum of the weights of all the samples

sample_num(*stat::AbstractStatistics*)
Returns the number of samples

stats_mean(*stat::AbstractStatistics*)
Calculates the mean of all the samples

stats_std_deviation(*stat::AbstractStatistics*)
Calculates the standard deviation of all the samples

stats_skewness(*stat::AbstractStatistics*)
Calculates the skewness of all the samples

stats_kurtosis(*stat::AbstractStatistics*)
Calculates the kurtosis of all the samples

7.10.2 Non-weighted Statistics

The simplest of our stats types, NonWeightedStatistics simply stores non-weighted samples

```
type NonWeightedStatistics <: AbstractStatistics
  samples::Vector{Float64}
  isSorted::Bool
end
```

NonWeightedStatistics()
Constructor that initializes an empty NonWeightedStatistics object

add_sample!(stat::NonWeightedStatistics, price::Float64)
Adds a sample

7.10.3 Generic Risk Statistics

This is the basic weighted stats collector

```
type GenericRiskStatistics <: AbstractStatistics
  statsType::StatsType
  samples::Vector{Float64}
  sampleWeights::StatsBase.WeightVec
  samplesMatrix::Matrix{Float64}
  isSorted::Bool
end

typealias RiskStatistics GenericRiskStatistics{GaussianStatsType}
```

gen_RiskStatistics(*dims::Int = 0*)
Constructs and returns a Risk Statistics object (Generic Risk Statistics with a Gaussian stats type)

adding_data!(stat::GenericRiskStatistics, sz::Int)
This prepares the stats collector to accept new samples. Because we use a matrix to store the samples and their weights, the size of the matrix has to be preallocated with the number of expected samples. If you are going to then add additional samples, this must be called again with the number of expected additional samples.

add_sample!(stat::GenericRiskStatistics, price::Float64, weight::Float64, idx::Int)
Adds a new sample with a given weight. The index is required because we are storing data in a pre-allocated matrix.

add_sample!(stat::GenericRiskStatistics, price::Float64, idx::Int)
Adds a new sample with a default weight of 1. The index is required because we are storing data in a pre-allocated matrix.

7.10.4 Generic Sequence Statistics

This data structure stores sequences of AbstractStatistics objects.

```
type GenericSequenceStats <: AbstractStatistics
  dimension::Int
  stats::Vector{AbstractStatistics}
  results::Vector{Float64}
  quadraticSum::Matrix{Float64}
end
```

GenericSequenceStats(*dimension::Int*, *dim2::Int = dimension*)
Constructor for the generic sequence stats type. "dimension" is for the number of sequences expected, while "dim2" is for the size of each expected sequence.

GenericSequenceStats()
Constructor for the generic sequence stats type, initializes everything to 0

reset!(gss::GenericSequenceStats, dimension::Int, dim2::Int = dimension)
Resets the generic sequence stats object with the provided dimensions. "dimension" is for the number of sequences expected, while "dim2" is for the size of each expected sequence.

adding_data!(stat::GenericSequenceStats, sz::Int, sz2::Int)
This is required for adding new data to our sampler. "sz" is for the number of new sequences added while "sz2" is for the size of those sequences.

add_sample!(stat::GenericSequenceStats, vals::Vector, idx::Int, weight::Float64 = 1.0)
Adds a new sequence of values with a given weight.

weight_sum (*stat::GenericSequenceStats*)
Calculates the sum of the weights in the first sequence

stats_mean (*stat::GenericSequenceStats*)
Calculates the mean for each sequence and returns a vector of the means.

stats_covariance (*stat::GenericSequenceStats*)
Calculates the covariance across all the sample sequences and returns a covariance matrix.

7.11 Transformed Grid

This data structure encapsulates an array of grid points. It is used primariy in PDE calculations.

```
type TransformedGrid
  grid::Vector{Float64}
  transformedGrid::Vector{Float64}
  dxm::Vector{Float64}
  dxp::Vector{Float64}
  dx::Vector{Float64}
end
```

TransformedGrid (*grid::Vector{Float64}*, *f::Function*)
Constructor for the transformed grid, given a function to transform the points

LogGrid (*grid::Vector{Float64}*)
Builds a transformed grid based on the log of the grid values

7.12 Tridiagonal Operator

We are using the custom Tridiagonal Operator from the original QuantLib, rewritten in Julia.

```
type TridiagonalOperator
  diagonal::Vector{Float64}
  lowerDiagonal::Vector{Float64}
  upperDiagonal::Vector{Float64}
  temp::Vector{Float64}
  n::Int
end
```

TridiagonalOperator (*n::Int*)
Constructor for the tridiagonal operator given a dimension

TridiagonalOperator ()
Constructor for the tridiagonal operator that defaults to being empty

TridiagIdentity (*n::Int*)
Builds and returns an identity tridiagonal operator

set_first_row!(L::TridiagonalOperator, valB::Float64, valC::Float64)
Sets the first row of the tridiagonal structure

set_mid_row!(L::TridiagonalOperator, i::Int, valA::Float64, valB::Float64, valC::Float64)
Sets a middle row of the tridiagonal structure, given an index

set_last_row!(L::TridiagonalOperator, valA::Float64, valB::Float64)
Sets the last row of the tridiagonal structure

solve_for!(L::TridiagonalOperator, rhs::Vector{Float64}, result::Vector{Float64})
Solve the linear system for a given right-hand side, with the solution in the result vector.

solve_for(*L::TridiagonalOperator*, *rhs::Vector{Float64}*)
Solve the linear system, using LU factorization (builds a Julia tridiagonal structure), returns the result vector.

CHAPTER 8

Pricing Methods

QuantLib.jl has various methods for asset pricing and calculation.

8.1 Finite Differences

Finite Differences framework for option pricing

8.1.1 General Finite Differences types

FDM Solver Description:

8.1.2 FDM Calculation Related Types

Inner value calculators

```
abstract FdmInnerValueCalculator
```

FDM Affine Model Swap Inner Value Calculator:

```
type FdmAffineModelSwapInnerValue <: FdmInnerValueCalculator
  disModel::Model
  fwdModel::Model
  swap::VanillaSwap
  exerciseDates::Dict{Float64, Date}
  mesher::FdmMesher
  direction::Int
  disTs::FdmAffineModelTermStructure
  fwdTs::FdmAffineModelTermStructure
end
```

FdmAffineModelSwapInnerValue(*disModel::Model*, *fwdModel::Model*, *swap::VanillaSwap*, *exerciseDates::Dict{Float64, Date}*, *mesher::FdmMesher*, *direction::Int*)

Constructor for the FDM Affine Swap Inner Value calculator, given a discount model, a forward model, a swap, and mesher

8.1.3 Finite Difference Step Conditions

```
abstract StepCondition
```

FDM Composite Step Condition

```
type FdmStepConditionComposite{C <: StepCondition} <: StepCondition
  stoppingTimes::Vector{Float64}
  conditions::Vector{C}
end
```

vanilla_FdmStepConditionComposite(*cashFlow::DividendSchedule*, *exercise::Exercise*, *mesher::FdmMesher*, *calculator::FdmInnerValueCalculator*, *refDate::Date*, *dc::DayCount*)

Builds and returns an FDM Step Condition composite for vanilla options

8.1.4 Finite Difference meshers

Brief mesher for an FDM grid

```
abstract FdmMesher
abstract Fdm1DMesher
```

Composite FDM Mesher type

```
type FdmMesherComposite <: FdmMesher
  layout::FdmLinearOpLayout
  meshers::Vector{Fdm1DMesher} # this could change
end
```

FdmMesherComposite(*mesh::Fdm1DMesher*)

Constructor for FDM Mesher composite type, with one mesher

FdmMesherComposite{F1D <: Fdm1DMesher}(xmesher::F1D, ymesher::F1D)

Constructor for FDM Mesher composite type with two meshers of the same type

1D FDM Mesher using a stochastic process:

```
type FdmSimpleProcess1dMesher <: Fdm1DMesher
  size::Int
  process::StochasticProcess1D
  maturity::Float64
  tAvgSteps::Int
  epsilon::Float64
  mandatoryPoint::Float64
  locations::Vector{Float64}
  dplus::Vector{Float64}
  dminus::Vector{Float64}
end
```

FdmSimpleProcess1dMesher (*sz::Int*, *process::StochasticProcess1D*, *maturity::Float64*, *tAvgSteps::Int*, *_eps::Float64*, *mandatoryPoint::Float64 = -1.0*)
Constructor for the FDM Simple Process (stochastic process) 1D mesher

8.1.5 Finite Difference operators

Linear operators to model a multi-dimensional PDE system

```
abstract FdmLinearOpComposite
```

FDM G2 operator - linear operator for the FDM G2 solver

```
type FdmG2Op <: FdmLinearOpComposite
  direction1::Int
  direction2::Int
  x::Vector{Float64}
  y::Vector{Float64}
  dxMap::TripleBandLinearOp
  dyMap::TripleBandLinearOp
  corrMap::SecondOrderMixedDerivativeOp
  mapX::TripleBandLinearOp
  mapY::TripleBandLinearOp
  model::G2
end
```

FdmG2Op (*mesher::FdmMesher*, *model::G2*, *direction1::Int*, *direction2::Int*)
Constructor for the FDM G2 operator

8.1.6 Finite Difference 2D Solver

2D Finite Differences solver

```
type Fdm2DimSolver <: LazyObject
  lazyMixin::LazyMixin
  solverDesc::FdmSolverDesc
  schemeDesc::FdmSchemeDesc
  op::FdmLinearOpComposite
  thetaCondition::FdmSnapshotCondition
  conditions::FdmStepConditionComposite
  initialValues::Vector{Float64}
  resultValues::Matrix{Float64}
  x::Vector{Float64}
  y::Vector{Float64}
  interpolation::BicubicSpline
end
```

Fdm2DimSolver (*solverDesc::FdmSolverDesc*, *schemeDesc::FdmSchemeDesc*, *op::FdmLinearOpComposite*)
Constructor for the FDM 2D solver

8.1.7 Finite Difference G2 Solver

Finite differences solver with a G2 short rate model

```
type FdmG2Solver <: LazyObject
  lazyMixin::LazyMixin
  model::G2
  solverDesc::FdmSolverDesc
  schemeDesc::FdmSchemeDesc
  solver::Fdm2DimSolver
end
```

FdmG2Solver (*model::G2*, *solverDesc::FdmSolverDesc*, *schemeDesc::FdmSchemeDesc*)
Constructor for the FDM G2 Solver

8.1.8 Finite Difference Hull White Solver

```
type FdmHullWhiteSolver <: LazyObject
  lazyMixin::LazyMixin
  model::HullWhite
  solverDesc::FdmSolverDesc
  schemeDesc::FdmSchemeDesc
  solver::Fdm1DimSolver
end
```

FdmHullWhiteSolver (*model::HullWhite*, *solverDesc::FdmSolverDesc*, *schemeDesc::FdmSchemeDesc*)
Constructor for the FDM Hull White solver

8.2 Monte Carlo Simulation

QuantLib.jl's Monte Carlo simulation tools include path generation and path pricer types

8.2.1 Monte Carlo Model

This is the general monte carlo model that is used for the simulation

```
abstract AbstractMonteCarloModel

type MonteCarloModel <: AbstractMonteCarloModel
  pathGenerator::PathGenerator
  pathPricer::AbstractPathPricer
  sampleAccumulator::RiskStatistics
  isAntitheticVariate::Bool
end
```

add_samples!(mcmodel::MonteCarloModel, samples::Int, idx::Int=1)
Adds samples generated by the simulation

8.2.2 Path Generator

```
type PathGenerator
  brownianBridge::Bool
  generator::AbstractRandomSequenceGenerator
  dimension::Int
```

(continues on next page)

(continued from previous page)

```
  timeGrid::TimeGrid
  process::StochasticProcess1D
  nextSample::Sample
  temp::Vector{Float64}
  bb::BrownianBridge
end
```

PathGenerator (*process::StochasticProcess*, *tg::TimeGrid*, *generator::AbstractRandomSequenceGenerator*, *brownianBridge::Bool*)
Constructor for the path generator, given a process, time grid, RSG, and brownian bridge flag

PathGenerator (*process::StochasticProcess*, *len::Float64*, *timeSteps::Int*, *generator::AbstractRandomSequenceGenerator*, *brownianBridge::Bool*)
Constructor for the path generator, given a process, two variables to build a time grid, a RSG, and brownian bridge flag

8.2.3 Path and Node

Path: Basic path data structure

```
type Path
  tg::TimeGrid
  values::Vector{Float64}
end
```

Path (*tg::TimeGrid*)
Constructor for a Path given a time grid

Node: Node data structure

```
type NodeData
  exerciseValue::Float64
  cumulatedCashFlows::Float64
  values::Vector{Float64}
  controlValue::Float64
  isValid::Bool
end
```

NodeData ()
Constructor for an empty NodeData object

8.2.4 Misc Methods and Types

generic_longstaff_schwartz_regression!(simulationData::Vector{Vector{NodeData}}, basisCoef
Returns the biased estimate obtained while regressing n exercises, n+1 elements in simulationData simulationData[1][j] -> cashflows up to first exercise, j-th path simulationData[i+1][j] -> i-th exercise, j-th path length(basisCoefficients) = n

8.3 Lattices and Trees

8.3.1 Trinomial Tree

This type defines a recombining trinomial tree approximating a 1D stochastic process.

```
type TrinomialTree <: AbstractTree
  process::StochasticProcess
  timeGrid::TimeGrid
  dx::Vector{Float64}
  branchings::Vector{Branching}
  isPositive::Bool
end
```

TrinomialTree (*process::StochasticProcess*, *timeGrid::TimeGrid*, *isPositive::Bool = false*)
Constructor for a trinomial tree given a stochastic process

8.3.2 Tree Lattice 1D

One-dimensional tree-based lattice

```
type TreeLattice1D <: TreeLattice
  tg::TimeGrid
  impl::TreeLattice
  statePrices::Vector{Vector{Float64}}
  n::Int
  statePricesLimit::Int
end
```

TreeLattice1D (*tg::TimeGrid*, *n::Int*, *impl::TreeLattice*)
Constructor of a 1D Tree lattice

CHAPTER 9

Pricing Models

QuantLib.jl has various pricing models for asset pricing.

9.1 General Model Types and Methods

ConstantParameter

```
type ConstantParameter <: Parameter
  data::Vector{Float64}
  constraint::Constraint
end
```

9.1.1 Calibration Helpers

Basket Types

```
type NaiveBasketType <: CalibrationBasketType end
type MaturityStrikeByDeltaGammaBasketType <: CalibrationBasketType end
```

** Swaption Helper**

SwaptionHelper(*maturity::Dates.Period*, *swapLength::Dates.Period*, *volatility::Quote*, *iborIndex::IborIndex*, *fixedLegTenor::TenorPeriod*, *fixedLegDayCount::DayCount*, *floatingLegDayCount::DayCount*, *yts::YieldTermStructure*, *pe::PricingEngine*, *strike::Float64 = -1.0*, *nominal::Float64 = 1.0*, *shift::Float64 = 0.0*, *exerciseRate::Float64 = 0.0*)

Constructor for SwaptionHelper

SwaptionHelper(*expiryDate::Date*, *endDate::Date*, *volatility::Quote*, *iborIndex::IborIndex*, *fixedLegTenor::TenorPeriod*, *fixedLegDayCount::DayCount*, *floatingLegDayCount::DayCount*, *yts::YieldTermStructure*, *pe::PricingEngine = NullSwaptionEngine()*, *strike::Float64 = -1.0*, *nominal::Float64 = 1.0*, *shift::Float64 = 0.0*, *exerciseRate::Float64 = 0.0*)

Constructor for SwaptionHelper

implied_volatility!(ch::CalibrationHelper, targetValue::Float64, accuracy::Float64, maxEva
Calculates the black volatility implied by the model

add_times_to!(swaptionHelper::SwaptionHelper, times::Vector{Float64})
Add times to the calibration helper

model_value!(sh::SwaptionHelper)
Returns the price of the instrument according to the model

underlying_swap!(swaptionHelper::SwaptionHelper)
Returns the underlying swap of the swaption

9.2 Equity Models

9.2.1 Bates Model

Bates stochastic volatility model

```
type BatesModel{CalibratedModelType} <: Model{CalibratedModelType}
  modT::CalibratedModelType
  hestonModel::HestonModel
  nu::ConstantParameter
  delta::ConstantParameter
  lambda::ConstantParameter
  process::BatesProcess
  common::ModelCommon
end
```

BatesModel (*process::BatesProcess*)
Constructor for the Bates model, given a Bates process

9.2.2 Heston Model

Heston model for the stochastic volatility of an asset

```
type HestonModel{CalibratedModelType} <: Model{CalibratedModelType}
  modT::CalibratedModelType
  theta::ConstantParameter
  kappa::ConstantParameter
  sigma::ConstantParameter
  rho::ConstantParameter
  v0::ConstantParameter
  process::HestonProcess
  common::ModelCommon
end
```

HestonModel (*process::HestonProcess*)
Constructor for a Heston model given a Heston process

9.3 Market Models

A good overview of the implementation of QuantLib.jl's market models can be seen in the MarketModel Example

9.3.1 Accounting Engines

Accounting Engine

An engine that collects cash flows along a market-model simulation

AccountingEngine(*evolver::AbstractMarketModelEvolver*, *product::MarketModelMultiProduct*, *initialNumeraireValue::Float64*)
Constructor for the AccountingEngine, given a market model evolver, market model product, and initial numeraire value

multiple_path_values!(ae::AccountingEngine, stats::GenericSequenceStats, numberOfPaths::Int
Returns the paths generated by the underlying model simulation

Pathwise Vegas Outer Accounting Engine

Engine collecting cash flows along a market-model simulation for doing pathwise computation of Deltas and vegas using Giles–Glasserman smoking adjoints method. Note, only works with displaced Libor Market Model. The method is intimately connected with log-normal Euler evolution. We must work with discretely compounding MM account. To compute a vega means changing the pseudo-square root at each time step So for each vega, we have a vector of matrices. So we need a vector of vectors of matrices to compute all the vegas. We do the outermost vector by time step and inner one by which vega. This implementation is different in that all the linear combinations by the bumps are done as late as possible, whereas PathwiseVegasAccountingEngine (not yet implemented) does them as early as possible.

PathwiseVegasOuterAccountingEngine(*evolver::LogNormalFwdRateEuler*, *product::MarketModelPathwiseMultiProduct*, *pseudoRootStructure::AbstractMarketModel*, *vegaBumps::Vector{Vector{Matrix{Float64}}}*, *initialNumeraireValue::Float64*)
Constructor for the PathwiseVegasOuterAccountingEngine, given a LogNormalFwdRateEuler evolver, pathwise market model product, market model, a vector of a vector of vega matrices, and an initial numeraire value.

multiple_path_values!(pwEng::PathwiseVegasOuterAccountingEngine, means::Vector{Float64}, er
Returns the path values generated by the underlying model simulation

9.3.2 Brownian Generators

SobolBrownianGeneratorFactory

Sobol Brownian generator for market model simulations. This is an incremental brownian generator using a Sobol random sequence generator, inverse-cumulative gaussian method, and brownian bridging. Only diagonal ordering is implemented at this time - A diagonal schema will be used to assign the variates with the best quality to the most important factors and the largest steps.

```
type SobolDiagonalOrdering <: SobolOrdering end

type SobolBrownianGeneratorFactory <: BrownianGeneratorFactory
  ordering::SobolOrdering
  seed::Int
end
```

create(*sob::SobolBrownianGeneratorFactory*, *factors::Int*, *steps::Int*)
Spawn a Sobol Brownian generator

9.3.3 Evolution Description

Market-model evolution description

This type stores * evolutionTimes = the times at which the rates need to be known, * rateTimes = the times defining the rates that are to be evolved, * relevanceRates = which rates need to be known at each time.

This type is really just a tuple of evolution and rate times * there will be n+1 rate times expressing payment and reset times of forward rates. * there will be any number of evolution times. * we also store which part of the rates are relevant for pricing via relevance rates. The important part for the i-th step will then range from relevanceRates[i].first to relevanceRates[i].second. Default values for relevance rates will be 1 and n+1.

Example for n = 5 (size = 6)

t0	t1	t2	t3	t4	t5	rateTimes
f0	f1	f2	f3	f4		forwardRates
d0	d1	d2	d3	d4	d5	discountBonds
d0/d0	d1/d0	d2/d0	d3/d0	d4/d0	d5/d0	discountRatios
sr0	sr1	sr2	sr3	sr4	sr5	coterminalSwaps

```
type EvolutionDescription
  numberOfRates::Int
  rateTimes::Vector{Float64}
  evolutionTimes::Vector{Float64}
  relevanceRates::Vector{Pair{Int, Int}}
  rateTaus::Vector{Float64}
  firstAliveRate::Vector{Int}
end
```

EvolutionDescription()
Constructor for an EvolutionDescription that generates initializes everything at 0.

EvolutionDescription(*rateTimes::Vector{Float64}*, *evolutionTimes::Vector{Float64}*, *relevanceRates::Vector{Pair{Int, Int}} = Vector{Pair{Int, Int}}()*)
Constructor for an EvolutionDescription based on rate times, evolution times, and optionally relevance rates passed in.

money_market_measure(*evol::EvolutionDescription*)
Discretely compounded money market account measure: for each step the first unexpired bond is used as numeraire.

9.3.4 Callability

collect_node_data!(evolver::AbstractMarketModelEvolver, product::MarketModelMultiProduct, c
Collects all node data generated by market model evolver

NothingExerciseValue

Null rebate type

```
type NothingExerciseValue <: MarketModelExerciseValue
  numberOfExercises::Int
  rateTimes::Vector{Float64}
  isExerciseTime::BitArray{1}
  evolution::EvolutionDescription
  currentIndex::Int
  cf::MarketModelCashFlow
end
```

NothingExerciseValue(*rateTimes::Vector{Float64}*, *isExerciseTime::BitArray{1} = BitArray{1}()*)
Constructor for a NothingExerciseValue

SwapForwardBasisSystem

SwapForwardBasisSystem(*rateTimes::Vector{Float64}*, *exerciseTimes::Vector{Float64}*)
Constructor for a SwapForwardBasisSystem, given rate times and exercise times

LongstaffSchwartzExerciseStrategy

LongstaffSchwartzExerciseStrategy(*basisSystem::MarketModelBasisSystem*, *basisCoefficients::Vector{Vector{Float64}}*, *evolution::EvolutionDescription*, *numeraires::Vector{Int}*, *exercise::MarketModelExerciseValue*, *control::MarketModelExerciseValue*)
Constructor for the LongStaff-Schwartz exercise strategy, given a MarketModelBasisSystem, basis coefficients, an EvolutionDescription, a vector of numeraires, a MarketModelExerciseValue, and a control MarketModelExerciseValue

SwapRateTrigger

SwapRateTrigger(*rateTimes::Vector{Float64}*, *swapTriggers::Vector{Float64}*, *exerciseTimes::Vector{Float64}*)
Constructor for the SwapRateTrigger exercise strategy

9.3.5 Correlations

ExponentialForwardCorrelation

Exponential correlation

- L = long term correlation
- beta = exponential decay of correlation between far away forward rates
- gamma = exponent for time to go
- times = vector of time dependences

ExponentialForwardCorrelation(*rateTimes::Vector{Float64}*, *longTermCorr::Float64 = 0.5*, *beta::Float64 = 0.2*, *gamma::Float64 = 1.0*, *times::Vector{Float64} = Vector{Float64}()*)
Constructor for the ExponentialForwardCorrelation type, with default values provided.

9.3.6 Evolvers

Evolvers do the actual gritty work of evolving the forward rates from one time to the next

LogNormalFwdRateEuler

```
type LogNormalFwdRateEuler <: AbstractMarketModelEvolver
  marketModel::AbstractMarketModel
  numeraires::Vector{Int}
  initialStep::Int
  generator::BrownianGenerator
  fixedDrifts::Vector{Vector{Float64}}
  numberOfRates::Int
  numberOfFactors::Int
  curveState::LMMCurveState
  currentStep::Int
  forwards::Vector{Float64}
  displacement::Vector{Float64}
```

(continues on next page)

(continued from previous page)

```
  logForwards::Vector{Float64}
  initialLogForwards::Vector{Float64}
  drifts1::Vector{Float64}
  initialDrifts::Vector{Float64}
  brownians::Vector{Float64}
  correlatedBrownians::Vector{Float64}
  alive::Vector{Int}
  calculators::Vector{LMMDriftCalculator}
end
```

LogNormalFwdRateEuler (*marketModel::AbstractMarketModel*, *factory::BrownianGeneratorFactory*, *numeraires::Vector{Int}*, *initialStep::Int = 1*)

Constructor for the Log-Normal Forward Rate Euler evolver, given a market model, brownian generation factory, vector of numeraires, and optionally an initial starting step.

LogNormalFwdRatePc

Predictor-corrector log-normal forward rate evolver

```
type LogNormalFwdRatePc <: AbstractMarketModelEvolver
  marketModel::AbstractMarketModel
  numeraires::Vector{Int}
  initialStep::Int
  generator::BrownianGenerator
  fixedDrifts::Vector{Vector{Float64}}
  numberOfRates::Int
  numberOfFactors::Int
  curveState::LMMCurveState
  currentStep::Int
  forwards::Vector{Float64}
  displacement::Vector{Float64}
  logForwards::Vector{Float64}
  initialLogForwards::Vector{Float64}
  drifts1::Vector{Float64}
  drifts2::Vector{Float64}
  initialDrifts::Vector{Float64}
  brownians::Vector{Float64}
  correlatedBrownians::Vector{Float64}
  alive::Vector{Int}
  calculators::Vector{LMMDriftCalculator}
end
```

LogNormalFwdRatePc (*marketModel::AbstractMarketModel*, *factory::BrownianGeneratorFactory*, *numeraires::Vector{Int}*, *initialStep::Int = 1*)

Constructor for the Log-Normal Forward Rate PC evolver, given a market model, brownian generation factory, vector of numeraires, and optionally an initial starting step.

9.3.7 Models

FlatVol

Flat volatility market model

FlatVol (*vols::Vector{Float64}*, *corr::PiecewiseConstantCorrelation*, *evolution::EvolutionDescription*, *numberOfFactors::Int*, *initialRates::Vector{Float64}*, *displacements::Vector{Float64}*)

Constructor for the FlatVol model

9.3.8 Market Model Multi-Products

Market model product types encapsulate the notion of a product: it contains the information that would be in the termsheet of the product.

It's useful to have it be able to do several products simultaneously. The products would have to have the same underlying rate times of course. The class is therefore really encapsulating the notion of a multi-product.

For each time evolved to, it generates the cash flows associated with that time for the state of the yield curve. If one was doing a callable product then this would encompass the product and its exercise strategy.

MarketModelComposite

Composition of two or more market-model products

```
type MarketModelComposite <: MarketModelMultiProduct
  components::Vector{SubProduct}
  rateTimes::Vector{Float64}
  evolutionTimes::Vector{Float64}
  finalized::Bool
  currentIndex::Int
  cashFlowTimes::Vector{Float64}
  allEvolutionTimes::Vector{Vector{Float64}}
  isInSubset::Vector{BitArray{1}}
  evolution::EvolutionDescription
end
```

MarketModelComposite()
Constructor for the MarketModelComposite

add_product!(mm::MarketModelComposite, product::MarketModelMultiProduct, multiplier::Float
Add a product to the composite

finalize!(mm::MarketModelComposite)
Finalize each of the component products

MultiStep MarketModel Multi-Products

Market model multi-products that can be evaluated in more than one step

MultiStepInverseFloater

```
type MultiStepInverseFloater <: MultiProductMultiStep
  common::MultiProductMultiStepCommon
  fixedAccruals::Vector{Float64}
  floatingAccruals::Vector{Float64}
  fixedStrikes::Vector{Float64}
  fixedMultipliers::Vector{Float64}
  floatingSpreads::Vector{Float64}
  paymentTimes::Vector{Float64}
  payer::Bool
  multiplier::Float64
  lastIndex::Int
  currentIndex::Int
end
```

MultiStepInverseFloater(*rateTimes::Vector{Float64}*, *fixedAccruals::Vector{Float64}*, *floatingAccruals::Vector{Float64}*, *fixedStrikes::Vector{Float64}*, *fixedMultipliers::Vector{Float64}*, *floatingSpreads::Vector{Float64}*, *paymentTimes::Vector{Float64}*, *payer::Bool = true*)
Constructor for the multi-step inverse floater multi-product

ExerciseAdaptor

```
type ExerciseAdapter <: MultiProductMultiStep
  common::MultiProductMultiStepCommon
  exercise::MarketModelExerciseValue
  numberOfProducts::Int
  isExerciseTime::BitArray{1}
  currentIndex::Int
end
```

ExerciseAdapter(*exercise::MarketModelExerciseValue*, *numberOfProducts::Int = 1*)
Constructor for an ExerciseAdaptor

CallSpecifiedMultiProduct

```
type CallSpecifiedMultiProduct <: MarketModelMultiProduct
  underlying::MarketModelMultiProduct
  strategy::ExerciseStrategy
  rebate::MarketModelMultiProduct
  evolution::EvolutionDescription
  isPresent::Vector{BitArray{1}}
  cashFlowTimes::Vector{Float64}
  rebateOffset::Int
  wasCalled::Bool
  dummyCashFlowsThisStep::Vector{Int}
  dummyCashFlowsGenerated::Vector{Vector{MarketModelCashFlow}}
  currentIndex::Int
  callable::Bool
end
```

CallSpecifiedMultiProduct(*underlying::MarketModelMultiProduct*, *strategy::ExerciseStrategy*, *rebate::MarketModelMultiProduct*)
Constructor for a CallSpecifiedMultiProduct

9.3.9 Market Model Pathwise Multi-Products

Market model product types encapsulate the notion of a product: it contains the information that would be in the termsheet of the product.

It's useful to have it be able to do several products simultaneously. The products would have to have the same underlying rate times of course. The class is therefore really encapsulating the notion of a multi-product.

For each time evolved to, it generates the cash flows associated to that time for the state of the yield curve. If one was doing a callable product then this would encompass the product and its exercise strategy.

This class differs from market-model multi-product in that it also returns the derivative of the pay-off with respect to each forward rate

MarketModelPathwiseInverseFloater

Pathwise product inverse floater for doing Greeks

```
type MarketModelPathwiseInverseFloater <: MarketModelPathwiseMultiProduct
  rateTimes::Vector{Float64}
  fixedAccruals::Vector{Float64}
  floatingAccruals::Vector{Float64}
  fixedStrikes::Vector{Float64}
  fixedMultipliers::Vector{Float64}
```

(continues on next page)

(continued from previous page)

```
  floatingSpreads::Vector{Float64}
  paymentTimes::Vector{Float64}
  payer::Bool
  multiplier::Float64
  lastIndex::Int
  evolution::EvolutionDescription
  currentIndex::Int
end
```

MarketModelPathwiseInverseFloater(*rateTimes::Vector{Float64}*, *fixedAccruals::Vector{Float64}*, *floatingAccruals::Vector{Float64}*, *fixedStrikes::Vector{Float64}*, *fixedMultipliers::Vector{Float64}*, *floatingSpreads::Vector{Float64}*, *paymentTimes::Vector{Float64}*, *payer::Bool*)

Constructor for the MarketModelPathwiseInverseFloater.

CallSpecifiedPathwiseMultiProduct

```
type CallSpecifiedPathwiseMultiProduct <: MarketModelPathwiseMultiProduct
  underlying::MarketModelPathwiseMultiProduct
  strategy::ExerciseStrategy
  rebate::MarketModelPathwiseMultiProduct
  evolution::EvolutionDescription
  isPresent::Vector{BitArray{1}}
  cashFlowTimes::Vector{Float64}
  rebateOffset::Int
  wasCalled::Bool
  dummyCashFlowsThisStep::Vector{Int}
  dummyCashFlowsGenerated::Vector{Vector{MarketModelPathWiseCashFlow}}
  currentIndex::Int
  callable::Bool
end
```

CallSpecifiedPathwiseMultiProduct(*underlying::MarketModelPathwiseMultiProduct*, *strategy::ExerciseStrategy*)

Constructor for the CallSpecifiedPathwiseMultiProduct

9.3.10 Pathwise Greeks

Types and methods for Greeks

VegaBumpCollection

There are too many pseudo-root elements to allow bumping them all independently so we cluster them together and then divide all elements into a collection of such clusters.

```
type VegaBumpCollection
  allBumps::Vector{VegaBumpCluster}
  associatedVolStructure::AbstractMarketModel
  checked::Bool
  nonOverlapped::Bool
  isFull::Bool
end
```

VegaBumpCollection(*volStructure::AbstractMarketModel*, *factorwiseBumping::Bool*)

Constructs the VegaBumpCollection

VolatilityBumpInstruments

These types are used in the Orthogonalized Bump Finder below.

```
type VolatilityBumpInstrumentJacobianSwaption
  startIndex::Int
  endIndex::Int
end

type VolatilityBumpInstrumentJacobianCap
  startIndex::Int
  endIndex::Int
  strike::Float64
end
```

OrthogonalizedBumpFinder

Pass in a market model, a list of instruments, and possible bumps. Get out pseudo-root bumps that shift each implied vol by one percent, and leave the other instruments fixed. If the contribution of an instrument is too correlated with other instruments used, discard it.

```
type OrthogonalizedBumpFinder
  derivativesProducer::VolatilityBumpInstrumentJacobian
  multiplierCutoff::Float64
  tolerance::Float64
end
```

OrthogonalizedBumpFinder (*bumps::VegaBumpCollection*, *swaptions::Vector{VolatilityBumpInstrumentJacobianSwaption}*, *caps::Vector{VolatilityBumpInstrumentJacobianCap}*, *multiplierCutoff::Float64*, *tolerance::Float64*) = *OrthogonalizedBumpFinder(VolatilityBumpInstrumentJacobian(bumps, swaptions, caps), multiplierCutoff, tolerance)*

Constructor for the OrthongalizedBumpFinder

get_vega_bumps!(obf::OrthogonalizedBumpFinder, theBumps::Vector{Vector{Matrix{Float64}}})

Creates the vector to pass into PathwiseVegasAccountingEngine

9.4 Short Rate Models

General short-rate model types and methods

PrivateConstraint

```
type PrivateConstraint <: Constraint
  arguments::Vector{Parameter}
end
```

test (*c::PrivateConstraint*, *x::Vector{Float64}*)

Tests if params satisfy the constraint

9.4.1 Short Rate Calibration Types and Methods

Calibration Function

Cost function for optimization methods

```
type CalibrationFunction <: CostFunction
  model::ShortRateModel
  helpers::Vector{CalibrationHelper}
  weights::Vector{Float64}
  projection::Projection
end
```

calibrate!(model::ShortRateModel, instruments::Vector{CalibrationHelper}, method::Optimizat
This method calibrates the model, which generates the appropriate calibration function to be used by the passed-in optimization method.

func_values (*calibF::CalibrationFunction*, *params::Vector{Float64}*)
Computes the cost function values in params

value (*calibF::CalibrationFunction*, *params::Vector{Float64}*)
Computes the cost function value in params

9.4.2 One Factor Short Rate Models

get_params (*m::OneFactorModel*)
Returns the model parameters

Gaussian Short Rate Model

One-factor Gaussian short-rate model. Formulation is in forward measure.

```
type GSR <: Gaussian1DModel{TermStructureConsistentModelType}
  lazyMixin::LazyMixin
  modT::TermStructureConsistentModelType
  stateProcess::StochasticProcess1D
  evaluationDate::Date
  enforcesTodaysHistoricFixings::Bool
  reversion::Parameter
  sigma::Parameter
  volatilities::Vector{Quote}
  reversions::Vector{Quote}
  volstepdates::Vector{Date}
  volsteptimes::Vector{Float64}
  volsteptimesArray::Vector{Float64}
  ts::YieldTermStructure
  swapCache::Dict{CachedSwapKey, VanillaSwap}
  common::ModelCommon
end
```

GSR (*ts::YieldTermStructure*, *volstepdates::Vector{Date}*, *volatilities::Vector{Float64}*, *reversion::Float64*, *T::Float64 = 60.0*)
Constructor for the Gaussian SR model

calibrate_volatilities_iterative!(model::GSR, helpers::Vector{CalibrationHelper}, method::C
Iterative calibration of model. With fixed reversion calibrate the volatilities one by one to the given helpers. It is assumed that that volatility step dates are suitable for this, i.e. they should be identical to the fixing dates of the helpers (except for the last one where we do not need a step). Also note that the endcritera reflect only the status of the last calibration when using this method.

get_volatilities (*model::GSR*)
Returns sigma values

get_params (*model::GSR*)
Returns model params

Hull White Model

Single-factor Hull White model

```
type HullWhiteFittingParameter <: Parameter
  a::Float64
  sigma::Float64
  ts::TermStructure
end

type HullWhite <: OneFactorModel{AffineModelType}
  modT::AffineModelType
  r0::Float64
  a::ConstantParameter
  sigma::ConstantParameter
  phi::HullWhiteFittingParameter
  ts::TermStructure
  privateConstraint::PrivateConstraint
  common::ModelCommon
end
```

HullWhite (*ts::TermStructure*, *a::Float64 = 0.1*, *sigma::Float64 = 0.01*)
Constructor for the HullWhite model

Black Karasinski Model

Standard Black Karasinski model

```
type BlackKarasinski <: OneFactorModel{TermStructureConsistentModelType}
  modT::TermStructureConsistentModelType
  a::ConstantParameter
  sigma::ConstantParameter
  ts::TermStructure
  privateConstraint::PrivateConstraint
  common::ModelCommon
end
```

BlackKarasinski (*ts::TermStructure*, *a::Float64 = 0.1*, *sigma = 0.1*)
Constructor for the Black Karasinski model

9.4.3 Two Factor Short Rate Models

G2 Model

Two-additive-factor gaussian model

```
type G2FittingParameter <: Parameter
  a::Float64
  sigma::Float64
  b::Float64
  eta::Float64
  rho::Float64
  ts::TermStructure
end
```

(continues on next page)

(continued from previous page)

```
type G2 <: TwoFactorModel{AffineModelType}
  modT::AffineModelType
  a::ConstantParameter
  sigma::ConstantParameter
  b::ConstantParameter
  eta::ConstantParameter
  rho::ConstantParameter
  phi::G2FittingParameter
  ts::TermStructure
  privateConstraint::PrivateConstraint
  common::ModelCommon
end
```

G2 (*ts::TermStructure*, *a::Float64 = 0.1*, *sigma::Float64 = 0.01*, *b::Float64 = 0.1*, *eta::Float64 = 0.01*, *rho::Float64 = -0.75*)

Constructor for the G2 model, with default values

CHAPTER 10

Pricing Engines

Pricing engines are the main pricing tools in QuantLib.jl. Each asset type has a variety of different pricing engines, depending on the pricing method. Every asset is associated with a pricing engine , which is used to calculate NPV and other asset data.

Pricing engines usually have one or more term structures tied to them for pricing.

10.1 Bond Pricing Engines

These pricing engines use a variety of methods to price bonds in QuantLib.jl

10.1.1 BlackCallableFixedRateBondEngine

Black-formula callable fixed rate bond engine

Callable fixed rate bond Black engine. The embedded (European) option follows the Black "European bond option" treatment in Hull, Fourth Edition, Chapter 20.

```
type BlackCallableFixedRateBondEngine <: PricingEngine
  volatility::CallableBondVolatilityStructure
end
```

BlackCallableFixedRateBondEngine (*fwdYieldVol::Quote*)
Constructor for the Black-Callable Fixed Rate Bond Engine. This will construct the volatility term structure.

10.1.2 DiscountingBondEngine

Basic discounting bond engine using a yield term structure for pricing

```
type DiscountingBondEngine <: PricingEngine
  yts::YieldTermStructure
end
```

DiscountingBondEngine()
Optional constructor that will generate a dummy null term structure that must be changed later.

10.1.3 TreeCallableFixedRateEngine

Numerical lattice engine for callable fixed rate bonds

```
type TreeCallableFixedRateEngine <: LatticeShortRateModelEngine{S}
  model::ShortRateModel
  timeSteps::Int
  common::LatticeShortRateModelEngineCommon
end
```

TreeCallableFixedRateEngine(*model::ShortRateModel*, *timeSteps::Int*)
Default constructor for the TreeCallableFixedRateEngine. This will generate the necessary lattice for pricing.

10.2 Credit Pricing Engines

These pricing engines are used for credit-related products

10.2.1 MidPointCdsEngine

```
type MidPointCdsEngine <: PricingEngine
  probability::AbstractDefaultProbabilityTermStructure
  recoveryRate::Float64
  discountCurve::YieldTermStructure
end
```

10.3 Swap Pricing Engines

Pricing engines used to price swaps

10.3.1 DiscountingSwapEngine

```
type DiscountingSwapEngine <: PricingEngine
  yts::YieldTermStructure
  includeSettlementDateFlows::Bool
end
```

DiscountingSwapEngine(*yts::YieldTermStructure*, *includeSettlementDateFlows::Bool = true*)
Constructor for the DiscountingSwapEngine

DiscountingSwapEngine()
Constructor for the DiscountingSwapEngine that will generate a dummy null yield term structure. This must be changed for any pricing.

10.4 Swaption Pricing Engines

Pricing engines to price swaptions

10.4.1 BlackSwaptionEngine

Shifted Lognormal Black-formula swaption engine. The engine assumes that the exercise date equals the start date of the passed swap.

```
type BlackSwaptionEngine <: PricingEngine
  yts::YieldTermStructure
  vol::Quote
  volStructure::SwaptionVolatilityStructure
  dc::DayCount
  displacement::Float64
end
```

BlackSwaptionEngine(*yts::YieldTermStructure*, *vol::Quote*, *dc::DayCount*, *displacement::Float64 = 0.0*)

Constructor for the BlackSwaptionEngine. Will generate the volatility term structure.

10.4.2 FdHullWhiteSwaptionEngine

Finite Differences swaption engine using a Hull White Model

```
type FdHullWhiteSwaptionEngine <: PricingEngine
  model::HullWhite
  tGrid::Int
  xGrid::Int
  dampingSteps::Int
  invEps::Float64
  schemeDesc::FdmSchemeDesc
  ts::TermStructure
end
```

FdHullWhiteSwaptionEngine(*model::HullWhite*, *tGrid::Int = 100*, *xGrid::Int = 100*, *dampingSteps::Int = 0*, *invEps::Float64 = 1e-5*, *schemeDesc::FdmSchemeDesc = FdmSchemeDesc(Douglas())*)

Constructor for the FdHullWhiteSwaptionEngine. Uses the term structure from the hull white model by default.

10.4.3 FdG2SwaptionEngine

Finite Differences swaption engine using a two-factor Gaussian model (G2)

```
type FdG2SwaptionEngine <: PricingEngine
  model::G2
  tGrid::Int
  xGrid::Int
  yGrid::Int
  dampingSteps::Int
  invEps::Float64
  schemeDesc::FdmSchemeDesc
  ts::TermStructure
end
```

FdG2SwaptionEngine(*model::G2*, *tGrid::Int = 100*, *xGrid::Int = 50*, *yGrid::Int = 50*, *dampingSteps::Int = 0*, *invEps::Float64 = 1e-5*, *schemeDesc::FdmSchemeDesc = FdmSchemeDesc(Hundsdorfer())*)

Constructor for the FdG2SwaptionEngine. Uses the term structure from the G2 model by default.

10.4.4 G2SwaptionEngine

Swaption priced by means of the Black formula, using a G2 model. The engine assumes that the exercise date equals the start date of the passed swap.

```
type G2SwaptionEngine <: PricingEngine
  model::G2
  range::Float64
  intervals::Int
end
```

10.4.5 Gaussian1DSwaptionEngine

One factor gaussian model swaption engine. All fixed coupons with start date greater or equal to the respective option expiry are considered to be part of the exercise into right.

Cash settled swaptions are not supported.

```
abstract GaussianProbabilities
type NoneProbabilities <: GaussianProbabilities end
type NaiveProbabilities <: GaussianProbabilities end
type DigitalProbabilities <: GaussianProbabilities end

type Gaussian1DSwaptionEngine <: PricingEngine
  model::Gaussian1DModel
  integrationPoints::Int
  stddevs::Float64
  extrapolatePayoff::Bool
  flatPayoffExtrapolation::Bool
  discountCurve::YieldTermStructure
  probabilities::GaussianProbabilities
end
```

Gaussian1DSwaptionEngine(*model::Gaussian1DModel*, *integrationPoints::Int = 64*, *stddevs::Float64 = 7.0*, *extrapolatePayoff::Bool = true*, *flatPayoffExtrapolation::Bool = false*, *discountCurve::YieldTermStructure = NullYieldTermStructure()*, *probabilities::GaussianProbabilities = NoneProbabilities()*)

Constructor for the Gaussian 1D Swaption Engine.

10.4.6 Gaussian1DNonstandardSwaptionEngine

One factor gaussian model non standard swaption engine.

All fixed coupons with start date greater or equal to the respective option expiry are considered to be part of the exercise into right.

All float coupons with start date greater or equal to the respective option expiry are considered to be part of the exercise into right.

For redemption flows an associated start date is considered in the criterion, which is the start date of the regular xcoupon period with same payment date as the redemption flow.

Cash settled swaptions are not supported

```
type Gaussian1DNonstandardSwaptionEngine <: PricingEngine
  model::Gaussian1DModel
  integrationPoints::Int
  stddevs::Float64
  extrapolatePayoff::Bool
  flatPayoffExtrapolation::Bool
  oas::Quote
  discountCurve::YieldTermStructure
  probabilities::GaussianProbabilities
end
```

Gaussian1DNonstandardSwaptionEngine(*model::Gaussian1DModel*, *integrationPoints::Int = 64*, *stddevs::Float64 = 7.0*, *extrapolatePayoff::Bool = true*, *flatPayoffExtrapolation::Bool = false*, *oas::Quote = Quote(-1.0)*, *discountCurve::YieldTermStructure = NullYieldTermStructure()*, *probabilities::GaussianProbabilities = NoneProbabilities()*)

Constructor for the Gaussian 1D Non-standard Swaption Engine.

10.4.7 JamshidianSwaptionEngine

Swaption engine using Jamshidian's decomposition. Concerning the start delay cf. http://ssrn.com/abstract=2246054

```
type JamshidianSwaptionEngine <: PricingEngine
  model::ShortRateModel
end
```

10.4.8 TreeSwaptionEngine

```
type TreeSwaptionEngine <: LatticeShortRateModelEngine{S}
  model::ShortRateModel
  timeSteps::Int
  common::LatticeShortRateModelEngineCommon
end
```

TreeSwaptionEngine(*model::ShortRateModel*, *tg::TimeGrid*)

Constructor for the TreeSwaptionEngine, using a time grid. This will generate the necessary lattice from the time grid.

TreeSwaptionEngine(*model::ShortRateModel*, *timeSteps::Int*)

Constructor for the TreeSwaptionEngine, using a number of time steps. With this construction, the necessary tree will not be generated until calculation.

10.5 Vanilla Option Pricing Engines

QuantLib.jl has a number of methods to price European, American, and Bermudan options

10.5.1 AnalyticEuropeanEngine

Pricing engine for European vanilla options using analytical formulae

```
type AnalyticEuropeanEngine <: PricingEngine
  process::AbstractBlackScholesProcess
end
```

10.5.2 AnalyticHestonEngine

Heston-model engine based on Fourier transform

```
type AnalyticHestonEngine <: AbstractHestonEngine
  model::HestonModel
  evaluations::Int
  cpxLog::ComplexLogFormula
  integration::HestonIntegration
end
```

AnalyticHestonEngine (*hestonModel::HestonModel*)
Constructor for the AnalyticHestonEngine. Will construct the integration method as well.

10.5.3 BaroneAdesiWhaleyApproximationEngine

Barone-Adesi and Whaley pricing engine for American options (1987)

```
type BaroneAdesiWhaleyApproximationEngine <: PricingEngine
  process::AbstractBlackScholesProcess
end
```

10.5.4 BatesEngine

Bates model engines based on Fourier transform

```
type BatesEngine <: AbstractHestonEngine
  model::BatesModel
  evaluations::Int
  cpxLog::ComplexLogFormula
  integration::HestonIntegration
end
```

BatesEngine (*batesModel::BatesModel*)
Constructor for the BatesEngine. This will also construct the integration method.

10.5.5 BinomialVanillaEngine

Pricing engine for vanilla options using binomial trees

```
type BinomialVanillaEngine{P <: AbstractBlackScholesProcess, T <: BinomialTreeType}
↪<: AbstractVanillaEngine
  process::P
```

(continues on next page)

(continued from previous page)

```
  timeSteps::Int
  treeClass::Type{T}
end
```

10.5.6 BjerksundStenslandApproximationEngine

Bjerksund and Stensland pricing engine for American options (1993)

```
type BjerksundStenslandApproximationEngine <: PricingEngine
  process::AbstractBlackScholesProcess
end
```

10.5.7 FDEuropeanEngine

Pricing engine for European options using finite-differences

```
type FDEuropeanEngine <: AbstractFDVanillaEngine
  process::AbstractBlackScholesProcess
  timeSteps::Int
  gridPoints::Int
  timeDependent::Bool
  exerciseDate::Date
  finiteDifferenceOperator::TridiagonalOperator
  intrinsicValues::SampledCurve
  BCs::Vector{BoundaryCondition}
  sMin::Float64
  center::Float64
  sMax::Float64
  prices::SampledCurve
  fdEvolverFunc::Function
end
```

FDEuropeanEngine (*process::AbstractBlackScholesProcess*, *fdEvolverFunc::Function*, *timeSteps::Int = 100*, *gridPoints::Int = 100*, *timeDependent::Bool = false*)
Constructor for the FDEuropeanEngine, requires a finite-differences evolver

10.5.8 FDBermudanEngine

Finite-differences Bermudan engine

```
type FDBermudanEngine <: FDMultiPeriodEngine
  process::AbstractBlackScholesProcess
  timeSteps::Int
  gridPoints::Int
  timeDependent::Bool
  exerciseDate::Date
  finiteDifferenceOperator::TridiagonalOperator
  intrinsicValues::SampledCurve
  BCs::Vector{BoundaryCondition}
  sMin::Float64
  center::Float64
  sMax::Float64
```

(continues on next page)

(continued from previous page)

```
  prices::SampledCurve
  stoppingTimes::Vector{Float64}
  timeStepPerPeriod::Int
  fdEvolverFunc::Function
end
```

FDBermudanEngine(*process::AbstractBlackScholesProcess*, *fdEvolverFunc::Function*, *timeSteps::Int = 100*, *gridPoints::Int = 100*, *timeDependent::Bool = false*)

Constructor for the FDBermudanEngine, requires a finite-differences evolver

10.5.9 FDAmericanEngine

Finite-differences pricing engine for American one asset options

```
type FDAmericanEngine <: FDStepConditionEngine
  process::AbstractBlackScholesProcess
  timeSteps::Int
  gridPoints::Int
  timeDependent::Bool
  exerciseDate::Date
  finiteDifferenceOperator::TridiagonalOperator
  intrinsicValues::SampledCurve
  BCs::Vector{BoundaryCondition}
  sMin::Float64
  center::Float64
  sMax::Float64
  prices::SampledCurve
  controlPrices::SampledCurve
  controlBCs::Vector{BoundaryCondition}
  controlOperator::TridiagonalOperator
  fdEvolverFunc::Function
end
```

FDAmericanEngine(*process::AbstractBlackScholesProcess*, *fdEvolverFunc::Function*, *timeSteps::Int = 100*, *gridPoints::Int = 100*, *timeDependent::Bool = false*)

Constructor for the FDAmericanEngine, requires a finite-differences evolver

10.5.10 IntegralEngine

Pricing engine for European vanilla options using integral approach

```
type IntegralEngine <: PricingEngine
  process::AbstractBlackScholesProcess
end
```

10.5.11 MCAmericanEngine

American Monte Carlo engine, using the Longstaff-Schwarz Monte Carlo engine for early exercise options

```
type MCAmericanEngine <: MCLongstaffSchwartzEngine
  process::AbstractBlackScholesProcess
  timeSteps::Int
```

(continues on next page)

(continued from previous page)

```
  timeStepsPerYear::Int
  requiredSamples::Int
  maxSamples::Int
  requiredTolerance::Float64
  brownianBridge::Bool
  seed::Int
  nCalibrationSamples::Int
  polynomOrder::Int
  polynomType::LsmBasisSystemPolynomType
  mcSimulation::MCSimulation
  pathPricer::LongstaffSchwartzPathPricer
end
```

MCAmericanEngine(*process::AbstractBlackScholesProcess; timeSteps::Int = -1, timeStepsPerYear::Int = -1, brownianBridge::Bool = false, antitheticVariate::Bool = false, requiredSamples::Int = -1, requiredTolerance::Float64 = -1.0, maxSamples::Int = typemax(Int), seed::Int = 0, rsg::AbstractRandomSequenceGenerator = InverseCumulativeRSG(seed), nCalibrationSamples::Int = 2048, polynomOrder::Int = 2, polynomType::LsmBasisSystemPolynomType = Monomial()*)

Constructor for the MCAmericanEngine

10.5.12 MCEuropeanEngine

European option pricing engine using Monte Carlo simulation

```
type MCEuropeanEngine <: MCVanillaEngine
  process::AbstractBlackScholesProcess
  timeSteps::Int
  timeStepsPerYear::Int
  requiredSamples::Int
  maxSamples::Int
  requiredTolerance::Float64
  brownianBridge::Bool
  seed::Int
  mcSimulation::MCSimulation
end
```

MCEuropeanEngine(*process::AbstractBlackScholesProcess; timeSteps::Int = -1, timeStepsPerYear::Int = -1, brownianBridge::Bool = false, antitheticVariate::Bool = false, requiredSamples::Int = -1, requiredTolerance::Float64 = -1.0, maxSamples::Int = typemax(Int), seed::Int = 1, rsg::AbstractRandomSequenceGenerator = InverseCumulativeRSG(seed)*)

Constructor for the MCEuropeanEngine

10.6 Black Calculator

Black 1976 calculator type

```
type BlackCalculator
  payoff::StrikedTypePayoff
  strike::Float64
  forward::Float64
```

(continues on next page)

(continued from previous page)

```
  stdDev::Float64
  discount::Float64
  variance::Float64
  d1::Float64
  d2::Float64
  alpha::Float64
  beta::Float64
  DalphaDd1::Float64
  DbetaDd2::Float64
  n_d1::Float64
  cum_d1::Float64
  n_d2::Float64
  cum_d2::Float64
  x::Float64
  DxDs::Float64
  DxDstrike::Float64
end
```

BlackCalculator (*p::StrikedTypePayoff*, *fwd::Float64*, *stdDev::Float64*, *disc::Float64*)
Constructor for the Black Calculator

initialize!(calc::BlackCalculator, p::StrikedTypePayoff)
Initializes the black calculator with a given payoff

value (*calc::BlackCalculator*)
Returns value of black calculator

delta (*calc::BlackCalculator*, *spot::Float64*)
Sensitivity to change in the underlying spot price.

vega (*calc::BlackCalculator*, *mat::Float64*)
Sensitivity to volatility.

10.7 Discretized Assets

Discretized Assets used by numerical methods

10.7.1 DiscretizedSwap

```
type DiscretizedSwap <: DiscretizedAsset
  nominal::Float64
  swapT::SwapType
  fixedResetTimes::Vector{Float64}
  fixedPayTimes::Vector{Float64}
  floatingResetTimes::Vector{Float64}
  floatingPayTimes::Vector{Float64}
  args::VanillaSwapArgs
  common::DiscretizedAssetCommon
end
```

DiscretizedSwap (*nominal::Float64*, *swapT::SwapType*, *referenceDate::Date*, *dc::DayCount*, *fixedPayDates::Vector{Date}*, *fixedResetDates::Vector{Date}*, *floatingPayDates::Vector{Date}*, *floatingResetDates::Vector{Date}*, *args::VanillaSwapArgs*)
Constructor for a Discretized Swap

10.7.2 DiscretizedSwaption

```
type DiscretizedSwaption <: DiscretizedOption
  underlying::DiscretizedSwap
  exercise::Exercise
  exerciseTimes::Vector{Float64}
  fixedPayDates::Vector{Date}
  fixedResetDates::Vector{Date}
  floatingPayDates::Vector{Date}
  floatingResetDates::Vector{Date}
  lastPayment::Float64
  common::DiscretizedAssetCommon
end
```

DiscretizedSwaption (*swaption::Swaption*, *referenceDate::Date*, *dc::DayCount*)
Constructor for a Discretized Swaption, based on an underlying swaption

CHAPTER 11

Stochastic Processes

Stochastic processes for use in various models and pricing engines

11.1 Heston Processes

11.1.1 HestonProcess

Square-root stochastic-volatility Heston process

```
type HestonProcess <: AbstractHestonProcess
  s0::Quote
  riskFreeRate::YieldTermStructure
  dividendYield::YieldTermStructure
  v0::Float64
  kappa::Float64
  theta::Float64
  sigma::Float64
  rho::Float64
  disc::AbstractDiscretization
  hestonDisc::AbstractHestonDiscretization
end
```

HestonProcess(*riskFreeRate::YieldTermStructure*, *dividendYield::YieldTermStructure*, *s0::Quote*, *v0::Float64*, *kappa::Float64*, *theta::Float64*, *sigma::Float64*, *rho::Float64*, *d::AbstractHestonDiscretization = QuadraticExponentialMartingale()*)

Default constructor for the Heston Process

11.1.2 BatesProcess

Square-root stochastic-volatility Bates process

```
type BatesProcess<: AbstractHestonProcess
  hestonProcess::HestonProcess
  lambda::Float64
  nu::Float64
  delta::Float64
  m::Float64
end
```

BatesProcess(*riskFreeRate::YieldTermStructure*, *dividendYield::YieldTermStructure*, *s0::Quote*, *v0::Float64*, *kappa::Float64*, *theta::Float64*, *sigma::Float64*, *rho::Float64*, *lambda::Float64*, *nu::Float64*, *delta::Float64*, *d::AbstractHestonDiscretization = FullTruncation()*)

Constructor for the Bates Process. Builds underlying Heston process as well

11.2 Black Scholes Processes

Various types of Black-Scholes stochastic processes

Black Scholes types are based off of a GeneralizedBlackScholesProcess, with a structure seen here:

```
type GeneralizedBlackScholesProcess <: AbstractBlackScholesProcess
  x0::Quote
  riskFreeRate::YieldTermStructure
  dividendYield::YieldTermStructure
  blackVolatility::BlackVolTermStructure
  localVolatility::LocalConstantVol
  disc::AbstractDiscretization
  isStrikeDependent::Bool
  blackScholesType::BlackScholesType
end
```

11.2.1 BlackScholesMertonProcess

Merton (1973) extension to the Black-Scholes stochastic process

BlackScholesMertonProcess(*x0::Quote*, *riskFreeRate::YieldTermStructure*, *dividendYield::YieldTermStructure*, *blackVolatility::BlackConstantVol*, *disc::AbstractDiscretization = EulerDiscretization()*)

Constructs a Black Scholes Merton Process, based off the GeneralizedBlackScholesProcess structure above.

11.3 Other Stochastic Processes

11.3.1 OrnsteinUhlenbeckProcess

```
type OrnsteinUhlenbeckProcess <: StochasticProcess1D
  speed::Float64
  vol::Float64
  x0::Float64
  level::Float64
end
```

OrnsteinUhlenbeckProcess (*speed::Float64*, *vol::Float64*, *x0::Float64 = 0.0*, *level::Float64 = 0.0*) = *new(speed, vol, x0, level)*

Constructor for the OrnsteinUhlenbeckProcess

11.3.2 GsrProcess

Gaussian short rate stochastic process

```
type GsrProcess <: StochasticProcess1D
  times::Vector{Float64}
  vols::Vector{Float64}
  reversions::Vector{Float64}
  T::Float64
  revZero::Vector{Bool}
  cache1::Dict{Pair{Float64, Float64}, Float64}
  cache2::Dict{Pair{Float64, Float64}, Float64}
  cache3::Dict{Pair{Float64, Float64}, Float64}
  cache4::Dict{Float64, Float64}
  cache5::Dict{Pair{Float64, Float64}, Float64}
end
```

GsrProcess (*times::Vector{Float64}*, *vols::Vector{Float64}*, *reversions::Vector{Float64}*, *T::Float64 = 60.0*)

Constructor for the GsrProcess

CHAPTER 12

Quotes

Basic Quote type for a price quote.

```
type Quote
  value::Float64
  is_valid::Bool
end
```

Quote (*value::Float64*, *is_valid::Bool = true) = new(value*, *is_valid*)
Default constructor for a Quote

CHAPTER 13

Term Structures and Curves

QuantLib.jl has various term structures and curves for asset pricing.

Various methods of bootstrapping rate curves are also available.

13.1 General Term Structure methods

time_from_reference(*ts::TermStructure*, *date::Date*)
Returns the time from the term structure's reference date given the day counter used by the term structure and a passed in date

13.2 Bootstrapping

QuantLib.jl has an iterative bootstrap type for bootstrapping a rate curve.

This bootstrapper uses a Brent Solver and Finite Differences Newton-Safe solver for bootstrap calculations

```
type IterativeBootstrap <: Bootstrap
  firstSolver::BrentSolver
  solver::FiniteDifferenceNewtonSafe
end
```

IterativeBootstrap()
Default constructor for the iterative bootstrap

initialize(*::IterativeBootstrap*, *ts::TermStructure*)
Initializes a term structure curve to prepare it for bootstrapping

Various traits govern how the bootstrapping is done, based on the curve that is being created. The currently implemented traits are:

```
type Discount <: BootstrapTrait end
type HazardRate <: BootstrapTrait end
```

13.2.1 Bootstrap Helpers

QuantLib.jl's bootstrapping uses various helpers to construct the appropriate curve.

FixedRateBondHelper

Fixed-coupon bond helper for curve bootstrap

```
type FixedRateBondHelper <: BondHelper
  price::Quote
  bond::FixedRateBond
end
```

SwapRateHelper

Rate helper for bootstrapping over swap rates

```
type SwapRateHelper <: RateHelper
  rate::Quote
  tenor::Dates.Period
  fwdStart::Dates.Period
  swap::VanillaSwap
end
```

DepositRateHelper

Rate helper for bootstrapping over deposit rates

```
type DepositRateHelper <: RateHelper
  rate::Quote
  tenor::TenorPeriod
  fixingDays::Int
  calendar::BusinessCalendar
  convention::BusinessDayConvention
  endOfMonth::Bool
  dc::DayCount
  iborIndex::IborIndex
  evaluationDate::Date
  referenceDate::Date
  earliestDate::Date
  maturityDate::Date
  fixingDate::Date
end
```

FraRateHelper

Rate helper for bootstrapping over FRA rates

```
type FraRateHelper <: RateHelper
  rate::Quote
  evaluationDate::Date
  periodToStart::Dates.Period
  iborIndex::IborIndex
  fixingDate::Date
```

(continues on next page)

(continued from previous page)

```
  earliestDate::Date
  latestDate::Date
end
```

SpreadCDSHelper

Spread-quoted CDS hazard rate bootstrap helper

```
type SpreadCDSHelper <: AbstractCDSHelper
  runningSpread::Quote
  tenor::Dates.Period
  settlementDays::Int
  calendar::BusinessCalendar
  frequency::Frequency
  paymentConvention::BusinessDayConvention
  dc::DayCount
  recoveryRate::Float64
  schedule::Schedule
  discountCurve::YieldTermStructure
  settlesAccrual::Bool
  paysAtDefaultTime::Bool
  protectionStart::Date
  probability::AbstractDefaultProbabilityTermStructure
  swap::CreditDefaultSwap
end
```

SpreadCDSHelper(*runningSpread::Quote*, *tenor::Dates.Period*, *settlementDays::Int*, *calendar::BusinessCalendar*, *frequency::Frequency*, *paymentConvention::BusinessDayConvention*, *rule::DateGenerationRule*, *dc::DayCount*, *recoveryRate::Float64*, *discountCurve::YieldTermStructure = NullYieldTermStructure()*, *settlesAccrual::Bool = true*, *paysAtDefaultTime::Bool = true*)

Constructor for the SpreadCDSHelper

13.3 Yield Term Structures

Interest-rate term structure

discount(*yts::YieldTermStructure*, *date::Date*)

Returns the discount factor from a given date

zero_rate(*yts::YieldTermStructure*, *date::Date*, *dc::DayCount*, *comp::CompoundingType*, *freq::Frequency = Annual()*)

Returns the implied zero-yield rate for a given date (returns an InterestRate object)

zero_rate(*yts::YieldTermStructure*, *time_frac::Float64*, *comp::CompoundingType*, *freq::Frequency = Annual()*)

Returns the implied zero-yield rate for a given time fraction from the reference date (returns an InterestRate object)

forward_rate(*yts::YieldTermStructure*, *date1::Date*, *date2::Date*, *dc::DayCount*, *comp::CompoundingType*, *freq::Frequency*)

Returns the forward interest rate between two dates

forward_rate(*yts::YieldTermStructure*, *date::Date*, *period::Integer*, *dc::DayCount*, *comp::CompoundingType*, *freq::Frequency*)

Returns the forward interest rate between a date and another date based on the passed-in time period

forward_rate(*yts::YieldTermStructure*, *time1::Float64*, *time2::Float64*, *comp::CompoundingType*, *freq::Frequency*)
Returns the forward interest rate between two time periods calculated from the reference date

13.3.1 FlatForwardTermStructure

Flat interest-rate curve

```
type FlatForwardTermStructure <: YieldTermStructure
  settlementDays::Int
  referenceDate::Date
  calendar::BusinessCalendar
  forward::Quote
  dc::DayCount
  comp::CompoundingType
  freq::Frequency
  rate::InterestRate
  jumpTimes::Vector{JumpTime}
  jumpDates::Vector{JumpDate}
end
```

FlatForwardTermStructure(*settlement_days::Int*, *referenceDate::Date*, *calendar::BusinessCalendar*, *forward::Quote*, *dc::DayCount*, *comp::CompoundingType = ContinuousCompounding()*, *freq::Frequency = QuantLib.Time.Annual()*)
Constructor for a FlatForwardTermStructure, with a quote used to generate the InterestRate object

FlatForwardTermStructure(*referenceDate::Date*, *calendar::BusinessCalendar*, *forward::Quote*, *dc::DayCount*, *comp::CompoundingType = ContinuousCompounding()*, *freq::Frequency = QuantLib.Time.Annual()*)
Constructor for a FlatForwardTermStructure with no settlement days passed (defaults to 0) and a quote to generate the interest rate object

FlatForwardTermStructure(*settlementDays::Int*, *calendar::BusinessCalendar*, *forward::Quote*, *dc::DayCount*, *comp::CompoundingType = ContinuousCompounding()*, *freq::Frequency = QuantLib.Time.Annual()*)
Constructor for a FlatForwardTermStructure with no reference date passed (will be calculated the first time it is requested) and a quote to generate the interest rate object

FlatForwardTermStructure(*referenceDate::Date*, *forward::Float64*, *dc::DayCount*)
Constructor for a FlatForwardTermStructure with only a reference date, forward rate, and day count passed in. Will default with a TargetCalendar, ContinuousCompounding, and an Annual frequence

FlatForwardTermStructure(*referenceDate::Date*, *forward::Float64*, *dc::DayCount*, *compounding::CompoundingType*, *freq::Frequency*)
Constructor for a FlatForwardTermStructure with no settlement days or calendar passed (defaults to 0 and TargetCalendar, respectively)

13.3.2 InterpolatedDiscountCurve

YieldTermStructure based on interpolation of discount factors

```
type InterpolatedDiscountCurve <: InterpolatedCurve
  settlementDays::Int
  referenceDate::Date
  dc::DayCount
  interp::Interpolation
```

(continues on next page)

(continued from previous page)

```
  cal::BusinessCalendar
  dates::Vector{Date}
  times::Vector{Float64}
  data::Vector{Float64}
end
```

InterpolatedDiscountCurve (*dates::Vector{Date}*, *discounts::Vector{Float64}*, *dc::DayCount*, *interpolator::Interpolation*)
Constructor for the InterpolatedDiscountCurve, with a given interpolation method

13.3.3 PiecewiseYieldCurve

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.

```
type PiecewiseYieldCurve <: InterpolatedCurve{P, T}
  lazyMixin::LazyMixin
  settlementDays::Int
  referenceDate::Date
  instruments::Vector{BootstrapHelper}
  dc::DayCount
  interp::Interpolation
  trait::BootstrapTrait
  accuracy::Float64
  boot::Bootstrap
  times::Vector{Float64}
  dates::Vector{Date}
  data::Vector{Float64}
  errors::Vector{Function}
  validCurve::Bool
end
```

13.3.4 FittedBondDiscountCurve

Discount curve fitted to a set of fixed-coupon bonds.

This class fits a discount function d(t) over a set of bonds, using a user defined fitting method. The discount function is fit in such a way so that all cashflows of all input bonds, when discounted using d(t), will reproduce the set of input bond prices in an optimized sense. Minimized price errors are weighted by the inverse of their respective bond duration.

The FittedBondDiscountCurve class acts as a generic wrapper, while its inner class FittingMethod provides the implementation details. Developers thus need only derive new fitting methods from the latter.

```
type FittedBondDiscountCurve <: Curve
  lazyMixin::LazyMixin
  settlementDays::Int
  referenceDate::Date
  calendar::BusinessCalendar
  bonds::Vector{BondHelper}
  dc::DayCount
```

(continues on next page)

(continued from previous page)

```
  fittingMethod::FittingMethod
  accuracy::Float64
  maxEvaluations::Int
  simplexLambda::Float64
end
```

FittedBondDiscountCurve(*settlementDays::Int*, *referenceDate::Date*, *calendar::BusinessCalendar*, *bonds::Vector{BondHelper}*, *dc::DayCount*, *fittingMethod::FittingMethod*, *accuracy::Float64*, *maxEvaluations::Int*, *simplexLambda::Float64*)

Constructor for the FittedBondDiscountCurve

13.3.5 Fitting Methods

Common interface for all fitting methods:

```
type FittingMethodCommons{T <: Real}
  solution::Vector{T}
  guessSolution::Vector{T}
  numberOfIterations::Int
  minimumCostValue::Float64
  weights::Vector{T}
  costFunction::FittingCost
end
```

ExponentialSplinesFitting

Exponential-splines fitting method

```
type ExponentialSplinesFitting <: FittingMethod
  constrainAtZero::Bool
  size::Int
  commons::FittingMethodCommons
end
```

ExponentialSplinesFitting(*constrainAtZero::Bool*, *size::Int*)

Constructor for the ExponentialSplines fitting method

SimplePolynomialFitting

Simple polynomial fitting method

```
type SimplePolynomialFitting <: FittingMethod
  constrainAtZero::Bool
  degree::Int
  size::Int
  commons::FittingMethodCommons
end
```

SimplePolynomialFitting(*constrainAtZero::Bool*, *degree::Int*, *size::Int*)

Constructor for the SimplePolynomial fitting method

NelsonSiegelFitting

Nelson-Siegel fitting method

```
type NelsonSiegelFitting <: FittingMethod
  constrainAtZero::Bool
  size::Int
  commons::FittingMethodCommons
end
```

NelsonSiegelFitting (*size::Int*)
Constructor for the Nelson Siegel fitting method

SvenssonFitting

Svensson Fitting method

```
type SvenssonFitting <: FittingMethod
  constrainAtZero::Bool
  size::Int
  commons::FittingMethodCommons
end
```

SvenssonFitting (*size::Int*)
Constructor for the Svensson fitting method

CubicBSplinesFitting

CubicSpline B-splines fitting method

```
type CubicBSplinesFitting <: FittingMethod
  constrainAtZero::Bool
  size::Int
  knots::Vector{Float64}
  splines::BSpline
  N::Int
  commons::FittingMethodCommons
end
```

CubicBSplinesFitting (*constrainAtZero::Bool*, *knots::Vector{Float64}*, *size::Int*)
Default constructor for the Cubic BSplines fitting method

13.4 Credit Term Structures

Term structures for calculating default probability in credit products

13.4.1 InterpolatedHazardRateCurve

Default probability term structure based on interpolation of hazard rates

```
type InterpolatedHazardRateCurve <: InterpolatedDefaultProbabilityCurve{P}
  settlementDays::Int
  referenceDate::Date
  dc::DayCount
  interp::Interpolation
  cal::BusinessCalendar
  dates::Vector{Date}
  times::Vector{Float64}
```

(continues on next page)

(continued from previous page)

```
  data::Vector{Float64}
end
```

InterpolatedHazardRateCurve(*dates::Vector{Date}*, *hazardRates::Vector{Float64}*, *dc::DayCount*, *interpolator::Interpolation*)

Constructor for the InterpolatedHazardRateCurve, given a set of dates and hazard rates

13.4.2 PiecewiseDefaultCurve

Piecewise default-probability term structure

This term structure is bootstrapped on a number of credit instruments which are passed as a vector of DefaultProbabilityHelper 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.

```
type PiecewiseDefaultCurve <: InterpolatedDefaultProbabilityCurve{P}
  lazyMixin::LazyMixin
  settlementDays::Int
  referenceDate::Date
  instruments::Vector{BootstrapHelper}
  dc::DayCount
  interp::Interpolation
  trait::BootstrapTrait
  accuracy::Float64
  boot::Bootstrap
  times::Vector{Float64}
  dates::Vector{Date}
  data::Vector{Float64}
  errors::Vector{Function}
  validCurve::Bool
end
```

PiecewiseDefaultCurve(*referenceDate::Date*, *instruments::Vector{BootstrapHelper}*, *dc::DayCount*, *interp::Interpolation*, *trait::BootstrapTrait*, *accuracy::Float64*, *boot::Bootstrap = IterativeBootstrap()*)

Constructor for a PiecewiseDefaultCurve

13.5 Volatility Term Structures

13.5.1 ConstantOptionVolatility

Constant caplet volatility, no time-strike dependence

```
type ConstantOptionVolatility <: OptionletVolatilityStructure
  settlementDays::Int
  referenceDate::Date
  calendar::BusinessCalendar
  bdc::BusinessDayConvention
  volatility::Float64
  dc::DayCount
end
```

ConstantOptionVolatility (*settlementDays::Int*, *calendar::BusinessCalendar*, *bdc::BusinessDayConvention*, *volatility::Float64*, *dc::DayCount*)
Constructor for ConstantOptionVolatility, with floating reference date

13.5.2 ConstantSwaptionVolatility

Constant swaption volatility, no time-strike dependence

```
type ConstantSwaptionVolatility <: SwaptionVolatilityStructure
  settlementDays::Int
  referenceDate::Date
  calendar::BusinessCalendar
  bdc::BusinessDayConvention
  volatility::Quote
  dc::DayCount
end
```

ConstantSwaptionVolatility (*settlementDays::Int*, *cal::BusinessCalendar*, *bdc::BusinessDayConvention*, *volatility::Quote*, *dc::DayCount*)
Constructor for ConstantSwaptionVolatility, with floating reference date

13.5.3 BlackConstantVol

Constant Black volatility, no time-strike dependence

```
type BlackConstantVol <: BlackVolTermStructure
  referenceDate::Date
  settlementDays::Int
  calendar::BusinessCalendar
  volatility::Quote
  dc::DayCount
end
```

BlackConstantVol (*refDate::Date*, *cal::BusinessCalendar*, *volatility::Float64*, *dc::DayCount*)
Constructor for BlackConstantVol term structure, fixed reference date

CHAPTER 14

Time Module

QuantLib.jl's Time sub-module provides calendars, day counters, tenor time periods, and other time related types and methods for pricing.

14.1 Misc. Time/Date methods

within_next_week (*d1::Date*, *d2::Date*)
Determines whether the second date is within the next week from the first date

within_next_week (*t1::Float64*, *t2::Float64*)
Determines whether the second year fraction is within the next week from the first year fraction

within_previous_week (*d1::Date*, *d2::Date*)
Determines whether the second date is within the previous week from the first date

14.2 Business Calendars

QuantLib.jl has a number of calendars based on region and asset-type. Some of this functionality is based off of Ito.jl and BusinessDays.jl

All calendars inherit from the abstract type:

```
abstract BusinessCalendar
```

14.2.1 General Calendars

A Null calendar exists which has no holidays and is used simply for date movement

```
type NullCalendar <: BusinessCalendar end
```

Also, a Target Calendar is available which has only basic holidays: * Saturdays * Sundays * New Year's Day (Jan 1) * Good Friday * Easter Monday * Labor Day (May 1) * Christmas (Dec 25) * Day of Goodwill (Dec 26)

```
type TargetCalendar <: BusinessCalendar end
```

A Joint Calendar construction exists that combines two calendars

```
type JointCalendar <: BusinessCalendar
  cal1::B
  cal2::C
end
```

Additional calendars are organized by geography and asset type

14.2.2 US Calendars

```
abstract UnitedStatesCalendar <: WesternCalendar

type USSettlementCalendar <: UnitedStatesCalendar; end
type USNYSECalendar <: UnitedStatesCalendar; end
type USNERCCalendar <: UnitedStatesCalendar; end
type USGovernmentBondCalendar <: UnitedStatesCalendar; end
```

USSettlementCalendar - General settlement calendar

USNYSECalendar - New York Stock Exchange calendar

USNERCCalendar - North American Energy Reliability Council calendar

USGovernmentBondCalendar - US government bond market

14.2.3 UK Calendars

```
abstract UnitedKingdomCalendar <: WesternCalendar

type UKSettlementCalendar <: UnitedKingdomCalendar end
type UKLSECalendar <: UnitedKingdomCalendar end
type UKLMECalendar <: UnitedKingdomCalendar end
```

UKSettlementCalendar - UK Settlement calendar

UKLSECalendar - London Stock Exchange calendar

UKLMECalendar - London Metals Exchange calendar

14.2.4 General Calendar methods

is_business_day (*cal::BusinessCalendar*, *dt::Date*)
Determines whether a given date is a business day, depending on the calendar used

easter_date (*y::Int*)
Returns the date of Easter Sunday based on year provided

is_holiday (*::BusinessCalendar*, *dt::Date*)
Determines whether a given date is a holiday, based on the calendar used

advance (*days::Day*, *cal::BusinessCalendar*, *dt::Date*, *biz_conv::BusinessDayConvention = Following()*)
Advance a number of days from the provided date

advance (*time_period::Union{Week, Month, Year}*, *cal::BusinessCalendar*, *dt::Date*, *biz_conv::BusinessDayConvention = Following()*)
Advance a number of weeks, months, or years from the provided date

adjust (*cal::BusinessCalendar*, *d::Date*)
Adjust a date based on a Following business day convention

adjust (*cal::BusinessCalendar*, *::BusinessDayConvention*, *d::Date*)
Adjust a date based on the provided business day convention

14.3 Business Day Conventions

These conventions specify the algorithm used to adjust a date in case it is not a valid business day.

```
abstract BusinessDayConvention

type Unadjusted <: BusinessDayConvention end
type ModifiedFollowing <: BusinessDayConvention end
type Following <: BusinessDayConvention end
```

Unadjusted - Do not adjust

Modified Following - Choose the first business day after the given holiday unless it belongs to a different month, in which case choose the first business day before the holiday.

Following - Choose the first business day after the given holiday.

14.4 Day Counters

These types provide methods for determining the length of a time period according to given market convention, both as a number of days and as a year fraction.

Adopted from Ito.jl and InterestRates.jl

All Day Counters inherit from this abstract type:

```
abstract DayCount
```

SimpleDayCount - Simple day counter for reproducing theoretical calculations.

```
type SimpleDayCount <: DayCount end
```

14.4.1 Day Counter methods

day_count (*c::DayCount*, *d_start::Date*, *d_end::Date*)
Returns the number of days between the two dates based off of the day counter method

year_fraction (*c::DayCount*, *d_start::Date*, *d_end::Date*)
Returns the fraction of year between the two dates based off of the day counter method

14.4.2 General Day Counters

```
type Actual360 <:DayCount ; end
type Actual365 <: DayCount ; end
```

Actual360 - Actual / 360 day count convention

Actual365 - Actual/365 (Fixed) day count convention

14.4.3 30/360 Day Counters

```
abstract Thirty360 <:DayCount

type BondThirty360 <: Thirty360; end
type EuroBondThirty360 <: Thirty360; end
type ItalianThirty360 <: Thirty360; end

typealias USAThirty360 BondThirty360
typealias EuroThirty360 EuroBondThirty360
```

USAThirty360 - 30/360 (Bond Basis)

EuroThirty360 - 30E/360 (Eurobond basis)

ItalianThirty360 - 30/360 (Italian)

14.4.4 Actual-Actual Day Counters

```
abstract ActualActual <: DayCount

type ISMAActualActual <: ActualActual; end
type ISDAActualActual <: ActualActual; end
type AFBActualActual <: ActualActual; end

typealias ActualActualBond ISMAActualActual
```

ISMAActualActual - the ISMA and US Treasury convention, also known as "Actual/Actual (Bond)"

ISDAActualActual - the ISDA convention, also known as "Actual/Actual (Historical)", "Actual/Actual", "Act/Act", and according to ISDA also "Actual/365", "Act/365", and "A/365"

AFBActualActual - the AFB convention, also known as "Actual/Actual (Euro)"

14.5 Frequency

Frequency of events

```
abstract Frequency

type NoFrequency <: Frequency end
type Once <: Frequency end
type Annual <: Frequency end
type Semiannual <: Frequency end
```

(continues on next page)

(continued from previous page)

```
type EveryFourthMonth <: Frequency end
type Quarterly <: Frequency end
type Bimonthly <: Frequency end
type Monthly <: Frequency end
type EveryFourthWeek <: Frequency end
type Biweekly <: Frequency end
type Weekly <: Frequency end
type Daily <: Frequency end
type OtherFrequency <: Frequency end
```

value(*::Frequency*)
Returns the number of times the event will occur in one year (e.g. 1 for Annual, 2 for Semiannual, 3 for EveryFourthMonth, etc)

period(*::Frequency*)
Returns the underlying time period of the frequency (e.g. 1 Year for Annual, 6 Months for Semiannual, etc)

14.6 Schedule

Payment schedule data structure

```
type Schedule
  effectiveDate::Date
  terminationDate::Date
  tenor::TenorPeriod
  convention::BusinessDayConvention
  termDateConvention::BusinessDayConvention
  rule::DateGenerationRule
  endOfMonth::Bool
  dates::Vector{Date}
  cal::BusinessCalendar
end
```

14.6.1 Date Generation methods

These conventions specify the rule used to generate dates in a Schedule.

```
abstract DateGenerationRule

type DateGenerationBackwards <: DateGenerationRule end
type DateGenerationForwards <: DateGenerationRule end
type DateGenerationTwentieth <: DateGenerationRule end
```

DateGenerationBackwards - Backward from termination date to effective date.

DateGenerationForwards - Forward from effective date to termination date.

DateGenerationTwentieth - All dates but the effective date are taken to be the twentieth of their month (used for CDS schedules in emerging markets.) The termination date is also modified.

14.7 Tenor Period

Data structure for a time period with frequency

```
type TenorPeriod
  period::Dates.Period
  freq::Frequency
end
```

TenorPeriod(*f::Frequency*)
Constructor for a TenorPeriod given a Frequency

TenorPeriod(*p::Dates.Period*)
Constructor for a TenorPeriod given a Julia Date Period

14.8 TimeGrid

A Time-Grid data structure

```
type TimeGrid
  times::Vector{Float64}
  dt::Vector{Float64}
  mandatoryTimes::Vector{Float64}
end
```

TimeGrid(*times::Vector{Float64}*, *steps::Int*)
Time Grid constructor given a vector of times and a number of steps

TimeGrid(*endTime::Float64*, *steps::Int*)
Time Grid constructor given an end time and a number of steps

closest_time(*tg::TimeGrid*, *t::Float64*)
Returns the time on the grid closest to the given t

return_index(*tg::TimeGrid*, *t::Float64*)
Returns the index i such that grid[i] = t

QuantLib.jl Settings

QuantLib.jl, upon loading, generates a Settings object which contains the global evaluation date.

```
type Settings
  evaluation_date::Date
end
```

set_eval_date!(sett::Settings, d::Date)
Update the global evaluation date

Index

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z