# Yield Term Structure

It contains a base curve – such as a curve with a series of zero rate, and its own curve (this curve) that contains a bunch of instruments.

It passes itself to class IterativeBootstrap. In performCalculation(), it calls the latter class method to do the bootstrapping and save the results to the base curve.

**PiecewiseYieldCurve**

**<Traits, Interpolator, bootstrap>***instruments  
instance of IterativeBootstrap  
Bootstrap from instruments to data\_*

*this\_curve =itself,  
base\_curve =IinterpolatedZeroCuve*

FittedBondDiscountCurve*global fitting as opposed to bootsrap*

*class FittingMethod*

BootstrapHelper  
*Provide quoteError()  
and impliedQuote()*

BootstrapError  
*provide min error obj function  
by adjusting current spot rate*

IterativeBootstrap  
*Do real work in calculate()*

Curve

**Bootstraptraits**  
= Discount, ZeroYield, ForwardRate

Interpolator

InterpolatedForwardCurve

InterpolatedZeroCurve  
*dates\_, data\_=yields\_  
Initialize interpolator and use it in ZeroYieldImpl()*

InterpolatedDiscountCurve  
*discountImpl()  
data\_= discountFactor*

InterpolatedCurve  
*time\_, data\_, interpolator*

ForwardRateStructure  
annual continuous compounding

ZeroYieldStructure  
annual continuous compounding

YieldTermStructure

TermStructure

RateHelper

BondHelper

**Discount factor** is interpolated directly. Zero rate and forward rate are to converted to discount factor.

Everything comes down to discount factor.

YieldTermStructure has the following public functions which are inherited by its children classes:

* DiscountFactor discount(const Date& d, bool extrapolate = false), also available with Time instead of Date input. Extrapolation can be enabled optionally.
* InterestRate zeroRate(const Date& d, const DayCounter& resultDayCounter, Compounding comp,Frequency freq = Annual,bool extrapolate = false), also available with Time input.
* InterestRate forwardRate(const Date& d1,const Date& d2,const DayCounter& dc, Compounding comp, Frequency freq = Annual, bool extrapolate = false), also available with Time input.
* Rate parRate(const std::vector<Date>& dates, const DayCounter& dc, Frequency freq = Annual, bool extrapolate = false) returns the par rate for a bond which pays on the specified dates.

All of the concrete yield classes derive from YieldTermStructure and thus inherit automatically the functions above. The only function that they have to implement is the pure virtual discountImpl(Time) function. Given the discount factor, all other rates can be derived.

Most of the Ratehelper classes need only the forecastingTermStrucutre so their setupTermStructure can offer. The exception is SwapRateHelper that needs a discountingTermStructure and which is set to the forecastingTermStructure if empty by setupTermStructure.

Swap needs the forwardingTermStructure to decide the cash flows, and needs the discountingTermStructure to price and get the faire rate.

The index parameter in RateHelper class provides the tenor, calendar, dayCounter information but not the forward Term Structure, which is setup explicitly in setupTermStructure method. One exception is BMASwap, where the discounting curve is given in IborIndex (see BMA Swap for the reason).

Termstructure::impliedTermsStructure: the implied (future) term structure.

ForwardSpreadedTermStructure: add a spread on the forward rate

ZeroSpreadedTermStructure: add a spread on the zero rate

# Volatility Term Structure

BlackVolTermStructure stores market vol quotes.

Black Variance is defined as vol^2\*t

LocalVolTermStructure stores

local volatility model vol, which can be derived from BlackVol surface (market quotes)

TermStructure

VolatilityTermStructure

BlackVolTermStructure  
blackForwardVariance()

blackVarianceImpl(t,K)

blackVolImpl(t,K)

LocalVolTermStructure

Vol surface  
localVolImpl(t,S) returns   
local vol sigma

BlackVolatility  
TermStructure  
blackVolImpl()

BlackVariance  
TermStructure  
blackVarianceImpl

LocalVolCurve  
depend on t but not S

backout local vol from black variance (BlackVarianceCurve)

LocalVolSurface  
depends on (t,S)

back out local vol surface  
from quoted black vol implied surface (BlackVolTermStructure)

LocalConstantVol  
no t and S dependence

BlackVarianceSurface  
time and strike dependent surface

BlackVarianceCurve  
time dependent ATM curve

**BlackConstantVol**

ImpliedVolTermStructure

BlackVolTermStructure models the volatility quotes in the market.

* blackVol = (average) volatility in [0, t]
* blackVariance = Black Variance over [0, t], or sigma^2 \* t
* blackForwardVol = (average) vol between t1 and t2
* blackForwardVariance = variance between t1 and t2

The derived classes either implement blackVolImpl() or blackVarianceImpl().

The BlackVarianceCurve class models the time dependent ATM vol curve. In comparison, the InterpolatedSmileSection class models the smile/smirk for a strike K.

LocalVolTermStructure models local volatility from the market (Black) vol quotes. It is the derivative of Black Vol as

# Swaption Vol Term Structure

VolatilityTermStructure

SwaptionVolatilityStructure

smileSectionImpl

volatilityImpl

SwaptionVolatility  
Discrete

SpreadedSwaption

Volatility

ConstantSwaption  
Volatility

SmileSection  
  
a smile for a given T  
(option tenor and swap tenor)

SwaptionVolatilityMatrix (ATM)

So it has FlatSmileSection

InterpolatedSmile  
Section

FlatSmileSection

SwaptionVolatility  
Cube2

SwaptionVolatility  
Cube1

SwaptionVolatility  
Cube

# Cap/Floor Vol Term Structure

StrippedOptionletBase

VolatilityTermStructure

StrippedOptionletAdapter turns StrippedOptionletBase into OptionletVolatilityStructure

CapletVarianceCurve  
(depreciated)

StrippedOptionlet  
Adapter

SpreadedOptionlet  
Volatility

ConstantOptionlet  
Volatility

No time/strike dependency

OptionletVolatility  
Structure  
  
smileSectionImpl()  
volatilityImpl()

AbcdFunction  
the abcd curve

CapFloorTermVolSurface

interpolation2D\_

ConstantCapFloor  
TermVolatility

CapFloorTermVolCurve

ATM cap vol  
interpolation\_

CapFloorTerm  
VolatilityStructure

volatilityImpl(T, K)

OptionletStripper1

OptionletStripper2

OptionletStripper

StrippedOptionlet

# Default Probability Term Structure

HazardRateStructure

**PiecewiseDefaultCurve**

DefaultProbabilityHelper

(boostrap helper)

InterpolatedSurvivor  
ProbabilityStructure

SurvivorProbabilityStructure

InterpolatedDefaultDensity

Curve

DefaultDensityStructure

DefaultProbabilityTermStructure

survivalProbabilityImpl  
defaultDensityImpl

InterpolatedHazardRateCurve

FlatHazardRate

# Pricing Engine

It uses setupArgumnets() to setup arguments of PricingEngine.   
It uses fetchResuls() to retrieve the results from PricingEngine.

Instruments  
  
NPV\_  
engine\_  
  
setupArguments()  
fetchResults()

Specific PricingEngine

Bond::results

Bond::engine

Bond::arguments

Bond

results

arguments

PricingEngine

arguments\_ results\_

discretization

StochasticProcessArray

JointStochasticProces

HestonHullWhite – 3 factor

**HestonProcess**

GJRGARCHProcess

G2Process

ForwardMeasureProcess

BatesProcess

StochasticProcess  
  
discretization\_

# Stochastic Process

EulerDiscretization

**Process provides instantaneous drift and diffusion,**

**Discretization\_ converts it to discrete value (such as by multiplying dt)**

StochasticProcess1D

squareRootProcess

OrnsteinUhlenbeckProcess

Merton76—JumpDiffusion

HullWhiteProcess

GeometricBrownianMotionProcess

**Generalized**

**BlackScholesProcess**  
  
riskFreeRateTS\_  
dividendTS\_  
blackVolatilityTS\_  
localVolatilityTS\_  
localVolatility()

localVolatility() returns LocalVol Term Structure

Black(76)Process

BSMProcess

BlackScholesProcess

BlackCalculator

class StochasticProcess and class StochasticProcess1D model the following process

The member functions

|  |  |
| --- | --- |
| Real drift (Time t, Real x) |  |
|  |  |
| Real diffusion (Time t, Real x) |  |
|  |  |
| Real expectation (Time t0, Real x0, Time dt) |  |
|  |  |
| Real stdDeviation(Time t0, Real x0, Time dt) |  |
|  |  |
| Real variance (Time t0, Real x0, Time dt) |  |
|  |  |
| Real evolve (Time t0, Real x0, Time dt, Real dw) |  |
|  |  |
| Real apply (Real x0, Real dx) |  |

The discretization of a process is provided in the EulerDiscretization class such as multiplying the drift term by dt.

return process.drift(t0,x0)\*dt;

class GeneralizedBlackScholesProcess and its derived classes BlackScholesProcess, BlackScholesMertonProcess model the process

which is the log of the BSM stock price S. Therefore the apply member function is overridden to

Real GeneralizedBlackScholesProcess::apply(Real x0, Real dx) const {

return x0 \* std::exp(dx);

}

class OrnsteinUhlenbeckProcess models the process

where a is called drift, r is level, and sigma is vol.

# Tree

Used in **BinomialVanillaEngine<Tree>**

Tree provides up/down probabilities and underlying which are called by Lattice.  
Lattice provides tree structure. It rolls back, which in turn is called by DiscretizedAsset.

Current state price is stored in DiscretizedAsset

DiscretizedOption

DiscretizedVanillaOption

applySpecificCondition() = max(stop, continue)

OneFactorModel::  
ShortRateTree

DiscretizedAsset  
  
Array values  
Lattice\* method\_ used to call method\_rollback()

preAdjustedValues()  
postAdjustedValues() calls applySpecificConditions()

BlackScholesLattice<Tree>

pd\_, pu\_  
Tree tree\_, used to call underlying and get up/down probability  
stepback()

Lattice provides rollback logic.

Tree provides up/down probability and underlying evolution.

underlying(i,j) gives the S value at node (t=i, j)

TreeLattice1D<Lattice>  
  
grid(t) calls Tree::underlying()

EqualJumps  
BinomialTree  
underlying(i,j)

tree\_ provides up\_, down\_ probability.

In stepback(), newValues has dimension = values-1.

TreeLattice<Lattice>  
  
std::vector<Array> statePrices\_

stepback()

TimeGrid

Lattice  
(for Tree/FD)  
rollback()  
TimeGrid t\_

Tree

CoxRossRubinstein

EqualProbabilities  
BinomialTree

TrinomialTree

BinomialTree

Tree

# Finite Difference

Operator 🡪 MixedScheme

BC Condition control ParallelEvolver 🡪 FDModel

Step Condition variable

So in ParallelEvolver type = vector(2), bc\_set is vector of 2x2.

setTime() in L and BC offers time dependent features.

SampledCurve  
sample(payoff\*)  
grid and its value

Grid means

one column

of t (vertical)

global functions:

CenteredGrid()  
BoundedGrid()  
BoundedLogGrid()

intrinsicValue\_ records each colume / step of underlying and option price

OperatorFactory

Theta = 0 : explicit Theta = 1: implicit Theta = 0.5: CrankNicolson

FDVanillaEngine  
  
SampledCurve intrinsicValue\_  
initializeInitialCondition()  
TridiagonalOperator  
initializeOperator()

OneAssetOption::Engine

ParallelEvolver assembles L, BC, and StepCondition for the state variable and control variable.

FiniteDifferenceModel does the backward induction.

In each step

For each step there is a L operator, BC, etc. That is why this is a vector

OperatorTraits typedefs two sets:

BoundaryCondition set and

StepCondition set

FDVanillaEngine does the setup.

FDStepConditionEngine does the calculation.

These are not usefule.

Conditions applied at every time step

StepCondition  
  
applyTo()

AmericanCondition

CurveDependent  
StepCondition

FDEngineAdapter  
< base>  
  
calculate()

FDAmericanCondition  
<base>

OperatorFactory generates a BSMOperator.

These classes are Use in BSMOperator to construct BSMOperator

**Operator**  
TridiagonalOperator  
  
applyTo()  
solveFor()

ParallelEvolver  
<Scheme>  
  
step() calls Scheme::step()  
applyTo() calls condition

ParallelEvolverTraits  
<OperatorTraits>  
traits helper class (to vector)

FDAmericanEngine  
<Scheme>

FDStepConditionEngine  
<Scheme=CrankNicolson>  
  
**calculate()**

LogGrid

TransformedGrid

PdeSecondOrderParabolic  
  
generateOperator()

Mixed**Scheme**<Operator>  
  
explicitPart\_, implicitPart\_  
step()

FiniteDifferenceModel  
<Evolver=ParallelEvolver>  
  
  
rollback() calls Evolver::step()

OperatorTraits<Operator>  
traits helper class

ImplicitEuler

CrankNicolson

ExplicitEuler

DPlusMinus

DZero

DMinus

BSMOperator

DPlus

PdeConstantCoef  
<PdeClass = PdeBSM>

PdeBSM

FDEuropeanEngine  
<Scheme>

DirichletBC

NeumannBC

BoundaryCondition  
<Operator>

# Random Number and Monte Carlo

RSG produces a series of changes.

* PathGenerator generates a path based on it
* PathPricer prices the path.
* The sampleAccumulator in MonteCarloModel collects it.

RNG = Random Number Generator

URNG = Uniform RNG  
RSG = Random Sequence Generator  
IC = Inverse Cumulative (function)  
GPR = GenericPseudoRandom

Path

timeGrid\_value\_

Sample<Path>  
path\_, weight\_

PathPricer<Path>

PathGenerator<RSG>  
  
process\_, rsg\_  
  
Sample\* next() calls process\_evolve()

**MonteCarloModel**  
<MCTraits, GPR,S>  
  
**addSamples()**  
S sampleAccumulator

SingleVariant<GPR>  
(MCTraits helper class)

**GenericPseudoRandom**<URNG, IC>  
is a trait/make class

**GenericLowDiscrepency**<URSG,IC>

It type defines PseudoRandom and PoissonPseudoRandom

It type defines urgn\_type, rng\_type, ursg\_type, and rsg\_type; and the make\_sequence\_generator method

Random number generator in Math

One can also directly put RNG into RSG to get RSG of any distribution.

URNG

e.g. MersenneTwisterUniformRng  
next() and nextInt32()

MCEuropeanEngine  
<GPR,S>

MCVanillaEngine  
<MCTraits, GPR,S>  
  
calculate() calls MCSimulation::calculate()

EuropeanPathPricer

Pricing Engine in Engine

InverseCumulativeRsg<URSG,IC>  
= another RSG

Monte Carlo Methods in Methods

VanillaOption::engine

MCSimulation  
<MCTraits, GPR,S>  
  
mcModel\_  
value()

Calculate()

URSG =  
RSG<RNG>

RNG =  
InverseCumulativeRng<URNG,IC>

IC  
e.g. InverseCumulativeNormal

TimeGrid is used in class Path. A path is a time grid with attached array values on the grid. A sample is a path with sample weight.

A MultiPath holds a vector of paths -- std::vector<Path> and MultiPath::pathSize() = Path::length(), the length of the first path.

PathGenerator::next() generates the path of interest. It generates a random variable and pass it to Process::evolve(); The RSG should be set to the correct size. Especially in MultiPathGenerator, the size of RSG should be (path length or steps)x(number of random factors). **RSG generates uncorrelated random variables, it is the process::evolve()’s responsibility to add correlations**.

PathPricer class simply defines a function interface – operator().

**MonteCarloModel<MCTraits, GPR,S>** stands in the center of Monte Carlo simulation. It brings everything together in its addSamples() method. In the template parameters’ list, MCTraits is traits class, be it SingleVariate for single path or Multivariate for multi-path; GPR can be any RSG, usually set to PseudoRandom or LowDiscrepancy; S is statistics class, by default it is Statistics.

MCSimulation class drives the MonteCarloModel. It uses the value method or the valueWithSamples method to control the # of samples and the resulting estimated errors. Its calculate method runs a complete simulation by checking the stop condition and calling the MonteCarloModel.

# Cash Flow

If it has value, use its value;

Otherwise use the other. T 🡪 today in NPV 🡪 not occurred

Passing Parameter

Has value?  
Default = F

Y

Use the value

F

N

N

Y

T

Include Ref (NPV) Day?

Include Today?  
Has value?

# Index

**Index provides (forward) fixing.**

**CouponPricer provides (adjusted) fixed price/rate.**

**FloatingRateCoupon uses Index for the (original) fixing rate and CouponPricer for the rate adjustment.**

**FloatingRateBond holds a series /Leg of FloatingRateCoupons.**

IndexManager

InterestRateIndex  
*declare forecastFixing()*

Index

Eonia

USDLibor

Libor

BMIIndex

forwardingTermStructure  
*define forecastFixing()*

OvernightIndex

IborIndex

forwardingTermStructure  
*define forecastFixing()*

OvernightIndexedSwapIndex

overnightIndex\_

SwapIndex

forwardingTermStructure

=iborIndex::forwardingTS

discountingTermStructure

iborIndex\_

Sonia

# Coupon

FloatingRateBond contains a Leg of floating rate coupons

FloatingRateCoupon  
  
InterestRateIndex index\_ for indexFixing  
FloatingRateCouponPricer pricer\_ setPricer()  
**rate() {pricer\_->swapletRate()**

Coupon

CashFlow

LegBPS = Leg basis point = PV01

Fair spread = NPV/PV01 = # (bps) = # \* (0.0001)

OvernightIndexedCoupon

CmsCoupon

IborCoupon

CapFlooredCoupon

AverageBMACoupon

RangeAccrualFloatersCoupon

CapFlooredCmsCoupon

CapFlooredIborCoupon

FloatingRateCoupon

Pricer

(global) setCouponPricer(leg, Pricer)

IborCouponPricer

RangeAccrualPricer

OvernightIndexedCoupon Pricer

CmsCouponPricer

AverageBMACoupon Pricer

BlackIbor  
CouponPricer  
  
swapletPrice, capletPrice(), floorletPrice()  
adjustedFixing

HaganPricer

Leg

RangeAccuralLeg

OvernightLeg

CmsLeg

IborLeg

AverageBMALeg

Coupon::endDate is the date coupon gets paid. It needs to be a business date. refEndDate is the date when interest accrues. It needs not be to a business date. Therefore it’s possible that refEndDate is a weekend and gets paid the next Monday.

OvernightIndexCoupon is annually paid; But it has daily valueDates\_ member variable that is initialized in the constructor. Then in OvernightIndexedCouponPricer::swapletRate(), the annualized rates are deduced from daily compounding.

In a regular swap, a swaplet is fixed in advance and paid in arrear. In comparison, a in arrear swap fixes in arrear and pays in arrear (usually in two business days). In other words, on a fixing date, the swaplet sets the 3m Libor rate **for the next three month** but pays the corresponding interest immediately in two business days. The floating rate is a martingale in the forward measure of three month later but the cash flow is paid in the beginning of this three month tenor, thus a convexity adjustment is needed: the floating rate in the forward measure of the tenor start is the regular forward rate in the forward measure of the tenor end plus the convexity adjustment.

FloatingRateCoupon class models the above logic of forward rate F(t, T, S). Its indexFixing method returns the regular forward rate based on the forward measure of tenor end S. Its adjustedFixing method returns the forward rate based on the forward measure of tenor start T or convexity adjusted. The method has to call BlackIborCouponPricer::adjustFixing because volatility information is needed for convexity adjustment (CMS needs swaption volatility). Of course, if it is not in arrear, no adjustment is needed and adjustedFixing = indexFixing.

Therefore,

* FloatingRateCoupon::rate() = gearing \* adjustedFixing + spread = actual floating coupon rate = Pricer::swapletRate() which is convexity adjusted.
* Pricer::swapletRate() = payment of floating leg on one unit notional = fair fixed-leg rate for a swaplet
* Pricer::optionletPrice = caplet/floorlet Price
* Pricer::optionletRate = ?? makes no sense.

The **reference date** of a term structure is the date when the term structure starts, i.e., time t=0 or DF = 1. Usually the term structure reads the evaluationDate() and adjusts it by the settlement days to get the reference date.

An index needs a term structure which is used as **forecasting term structure** for forward fixing. This contrasts with the **discounting term structure** that is used for discounting.

In sum,

* An Index provides forward fixing
* A FloatingRateCoupon uses index for fixing
* A FloatingRateCouponPricer provides convexity adjustment for FloatingRateCoupon
* RateHelper helps the bootstrapping.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| IborIndex | OvernightIndex |  |  | Daily interest accumulated to 3m |
| SwapIndex | OvernightIndexedSwapIndex |  | BMAIndex | (e.g. 5Y) swap rate / OIS rate |
|  |  |  |  |  |
| IborCoupon | OvernightIndexedCoupon | CmsCoupon |  | A (3m) coupon |
| IborCouponPricer | OvernightIndexedCouponPricer | CmsCouponPricer/conumdrumPricer |  | Provide swapletRate() |
| VanillaSwap | OvernightIndexedSwap |  | BMASwap | IRS / OIS |
| makeVanillaSwap | makeOIS |  |  |  |

IborIndex gets to choose fixingCalendar (the one in charge of fixing, not the one from forwarding curve). In contrast, Libor class can only choose financial center calendar, which is joined by the default fixing calendar – UK::settlement. OvernightIndex is derived from IborIndex, so it gets to choose fixing calendar.

The BMA index is the short-term tax-exempt reference index of the Bond Market Association. It has tenor one week, is fixed weekly on Wednesdays and is applied with a one-day's fixing gap from Thursdays on for one week. It is the tax-exempt correspondent of the 1M USD-Libor.

<http://www.fincad.com/derivatives-resources/wiki/tax-exempt-swap-curve.aspx>

**Tax-Exempt (**Municipal**) Swap Curve**

BMA Swaps and BMA Swap Curve

A **BMA swap** is an [interest rate swap](http://www.fincad.com/derivatives-resources/wiki/interest-rate-swap.aspx) in which the payments of one leg are variable and are based upon fixings of the **US SIFMA Municipal Swap Index** (formerly the BMA Municipal Swap Index or "BMA Index"). This index is produced weekly, reflecting the average rate of issues of tax-exempt variable-rate debt, and serves as a benchmark floating rate in municipal swap transactions. The BMA index is usually 65%-70% of its taxable equivalent 1-month LIBOR. This ratio is subject to tax-risk, i.e., the risk that marginal tax rates will change or that there will be revisions to the US Tax Code.

The BMA Swap Curve represents the expected future values of the BMA index, where expectations are taken in the corresponding forward probability measure; the forward rates that are encoded in the curve can be used to calculate expected future cash flows for the purpose of valuing the BMA leg. Similar to other curve generation processes, the BMA Swap Curve is generated using a set of quoted cash rates and par rates for BMA fixed/floating swaps. Another important input is the risk-free discount factor curve (usually the LIBOR curve), which is used to calculate the present value of expected future cash flows. The par rate for a BMA fixed/floating swap of a particular maturity (e.g., 10 years) can be derived from the BMA Basis factor for that maturity (e.g., 75%) and the corresponding LIBOR swap rate. A BMA Basis factor of 75% means that a BMA/LIBOR basis swap, in which one leg pays 75% of LIBOR, and the other leg pays BMA, is a par swap. Thus, if the 10Y LIBOR swap rate (for a 10Y fixed/floating LIBOR swap) is, say, 4%, then the par rate for a BMA fixed/floating swap is 75% x 4% = 3%.

Technical Details

Bootstrapping starts with the shortest term swap and steps through them all in ascending order of maturity. At every step, forward rates inferred from the preceding swaps are considered as known, and subsequent forward rates are constrained to recover the price of the current swap. When valuing the BMA leg, the BMA Swap Curve is intended to be used as the "accruing curve" (used to calculate expected cash flows from its implied forward rates). The LIBOR curve is typically used as the "discounting curve".

The bootstrapping process iterates through the given par rates for BMA fixed/floating swaps and outputs discount factors and optionally forward rates. Note that the discount factor curve for discounting (i.e., the LIBOR curve) is already known, and therefore it is only necessary to build a curve that encodes implied forward rates.

In AverageBMACoupon, the rate is (annualized) accumulative average BMA rate that according to the interest you could get if one receives the weekly BMA interest rate.

GeneralStatistics

std::vector<std::pair<Real,Real> > samples\_

add()

GeneralStatistics stores all samples

IncrementalStatistics stores only statistics

IncrementalStatistics

void addSequence()

# Statistics

GenericGaussianStatistics <Stat>

-- add Gaussian assumption

-- add VaR, ES

-- typedef GenericGaussianStatistics<GeneralStatistics>

= **GaussianStatistics**

Decoration

DiscrepencyStatistics

GenericSequenceStatistics<S>

-- Multi-dimensional (sequence) data

-- typedef GenericSequenceStatistics<Statistics>

= SequenceStatistics

-- typedef GenericSequenceStatistics<IncrementalStatistics>

= SequenceStatisticsInc

ConvergenceStatistics<S>

GenericRiskStatistics <Stat>

**-- typedef GenericRiskStatistics<GaussianStatistics>**

**= RiskStatistics = Statistics**

# Heston Model

1. The objective is to minimize the least square of errors in class CalibrationFunction.
2. CalibratedModel provides constraints, parameters, and instruments to CalibrationFunction.
3. At each step, CalibrationFunction converts variable x into parameters in CalibratedModel, which updates the pricing engine that will give new prices.
4. **CalibratedModel** stays at the center of this mechanism. It provides information for both optimization and pricing.

CalibratedModel

OptimizationMethod minimize(Problem + EndCriteria)

GenericModelEngine  
  
model\_

OptimizationMethod  
  
minimize()

Parameters

HestonProcess

**CalibratedModel**  
  
get and set Params

calibrate()

AnalyticHestonEngine

Problem

PrivateConstraint  
  
test private constraints inherent in parameters

CalibrationHelper  
  
**engine\_**

CostFunction

Constraint

Problem

HestonModelHelper

HestonModel

HestonProcess

FellerConstraint

initial value and current value

EndCriteria

**CalibrationFunction**lease square error  
setParams()

EndCriteria

# Hull-White

Model  
Engine

How to  
update  
parameters

Calibration  
Helper

Model

setParams() is used to update its own parameters

generateArguments() is used to update parameters of underlying process

GenericEngine

CalibrationHelper  
  
pricingengine\_  
modelValue()

**CalibratedModel**  
  
argument\_  
constraint\_  
shortRateEndCriteria\_  
setParams()  
**calibrate()**  
value(CalibrationHelper\*)

Optimizer\_

AffineModel

GenericModelEngine

OptimizationMethod

TreeCapFloorEngine

TreeSwaptionEngine

TreeSwapEngine

LatticeShortRate  
ModelEngine

TermStructure  
ConsistentModel  
termStructure\_

When an array of parameters is passed into Calibration Function as usual, it calls model\_->setParams(), which sets the parameters and update the model.

**GenericModelEngine** observes the model, and updates as well.  
  
CalibrationHelper is set to the pricing engine. It uses which to calculate the model Value.

TermStructureFittingParameter (in HullWhite::FittingParameter) is Time-dependent parameter for yield-curve fitting.  
HullWhite::Dynamics determines true short rate movement by shifting x by (amount = fitting)  
  
HullWhite::tree() actually constructs the TreeLattice

ShortRateModel  
  
Lattice\* tree()

HullWhite  
  
Lattice\* tree()

CalibratedModel::  
PrivateConstraint  
*constraint  
test all the constraints*

EndCriteria

HullWhite::FittingParameter

HullWhite::Dynamics

Vasicek::Dynamics

TermStructure  
FittingParameter

Vasicek  
  
discountBondOption()

OneFactorAffineModel

OneFactorModel::  
ShortRateDynamics

OneFactorModel::  
ShortRateTree::  
Helper

OneFactorModel::  
ShortRateTree

OneFactorModel  
  
Lattice\* tree()

SwaptionHelper

CapHelper

Parameter

CalibratedModel::  
CalibrationFunction  
*cost function*

# Hull\_White (2)

State price depends on prob and discount of a tree.

Engine

CalibratedModel

ShortRateModel  
  
Lattice\* tree()

GenericModel  
Engine

TermStructureFittingParameter stores alpha(t) for yield-curve fitting.

It is used by HullWhite::Dynamics to determines true short rate movement r = x + alpha

1. BK::Helper class solves alpha with the aid of state price.

2. TrinomialTree from x0, using Brigo’s method

3. **HullWhite::tree() actually constructs the TreeLattice**, which includes 1 and 2.

In Lattice::initialize(), it resets the DiscretizedAsset.

Lattice

TreeLattice1D  
  
computeSatePrice()

LatticeShortRate  
ModelEngine

OneFactorModel  
  
discount()

TreeVanillaSwap  
Engine

TreeSwaption  
Engine

TermStrucutre  
FittingParameter

ShortRateDynamic  
  
transfer between alpha(t)+x(t)=r(t)

ShortRateTree

Helper class backs out alpha from state price

Tree

TrinomialTree

DiscretizedAsset provides mandatoryTimes().

It is acknowledged by Discretized Pricing Engine.

In DiscretizedOption:: postAdjustValuesImpl(), the 5y swap first is rolled back to 1y (in a 1y5y swaption).

In DiscretizeSwaption::reset(), it initializes underlying swap and **prices swap** in DiscretizedOption::postAdjustValuesImpl().

**In DiscretizedSwap::stepback(), it brings future cashflows to current step; then preAdjustValuesImpl() add current cash flows.**

**Continuous Compounding in OneFactorModel::discount(), and use r(t) to approximate R(t), see discount().**

DiscretizedAsset

DiscretizedOption  
  
DiscretizedAsset\* underlying\_  
postAdjustValuesImpl()

DiscretizedDiscountBond

The grid is given by:

mandatoryTime 🡪 Discritized Swap 🡪 Engine 🡪 TimeGrid 🡪 Lattice

DiscretizedSwaption

# Market Model

Model  
  
Product  
  
Evolver  
  
CurveState

Lognormal LIBOR market model  
  
Normal LIBOR market model

CMS Market Model

Coterminal Swap market model

AccountingEngine

ExerciseStrategy  
<CurveState>

PiecewiseConstant  
Correlation

SwapRateTrigger

Parametric  
ExerciseAdapter

LongstaffSchwartz  
ExerciseStrategy

TimeHomogeneous  
ForwardCorrelation

ExponentialForward  
Correlation

CotSwapFromFwd  
Correlation

CurveState

CoterminalSwap  
CurveState

LMMCurveState

CMSwapCurveState

LMMNormal  
DriftCalculator

LMM  
DriftCalculator

SMM  
DriftCalculator

CMSMM  
DriftCalculator

Pathwise  
AccountingEngine

AccountingEngine

EvolutionDescription

ProxyGreekEngine

**UpperBounded  
Engine**

MarketModel  
PathwiseDiscounter

MarketModel  
Discounter

NodeData

ParametricExercise

MarketModel  
NodeDataProvider

**collectNodeData()**

SwapForward  
Mapping

ForwardForward  
Mapping

MarketModel  
ParametricExercise

MarketModel  
BasisSystem

MarketModel  
ExerciseValue  
  
Early Exercise  
or Callability

Triggered  
SwapExercise

Nothing  
ExerciseValue

BermudanSwaption  
ExerciseValue

SwapForward  
BasisSystem

SwapBasisSystem

HistoricalForward  
RateAnalysis

Historical  
RateAnalysis

PiecewiseConstant  
Correlation

rateVolDifferences()

Pathwise Greeks

VegaBumpCluster

Swaption  
PseudoDerivative

RatePseudoRoot  
JacobianNumerical

VolatilityBump  
InstrumentJacobian

VolatilityInterpolation  
Specifierabcd

VolatilityInterpolation  
Specifier

PseudoRootFacade

PiecewiseConstant  
AbcdVariance

PiecewiseConstant  
Variance

FwdToCotSwapAdapter

FwdPeriodAdapter

FlatVol

CotSwapToFwdAdapter

CTSMMCaplet  
OriginalCalibration

CTSMMCapletCalibration

capletSwaption  
Periodic Calibration()

CTSMMCaplet  
MaxHomogeneityCalibration

CTSMMCaplet  
AlphaFormCalibration

MarketModel

AlphaForm  
LinearHyperbolic

AlphaForm  
InverseLinear

AlphaForm

AlphaFinder

AbcdVol

MarketProduct  
PathwiseWrapper

MarketModel  
MultiProduct

MarketModelPathwise  
MultiProduct

CallSpecified  
MultiProduct

MultiProduct  
MultiStep

MarketModel  
Composite

MarketModelPathwise  
MultiCaplet

CalledSpecifiedPathwise  
MultiProduct

MarketModelPathwise  
CashRebate

MultiProduct  
OneStep

MarketModelPathwise  
Swap

MarketModelPathwise  
InverseFloater

MarketModel  
CashRebate

MultiProduct  
Composite

SingleProduct  
Composite

MarketModelPathwise  
CoterminalSwaption  
Deflated

Exercise  
Adapter

MultiStep  
CoinitialSwaps

OneStep  
CoinitialSwaps

MultiStep  
CoterminalSwaptions

MultiStep  
CoterminalSwaps

OneStep  
CoterminalSwaps

OneStep  
Forwards

MultiStep  
InverseFloater

MultiStep  
Forwards

OneStep  
Optionlets

MultiStep  
Optionlets

MultiStep  
Nothing

MultiStep  
Ratchet

MultiStep  
PeriodCapletSwaptions

MultiStep  
Swaption

MultiStep  
Swap

SquareRoot  
Andersen

SVD  
FwdRatePc

Normal  
FwdRatePc

MarketModel  
VolProcess

LogNormal  
FwdRateEuller

LogNormal  
FwdRatePc

LogNormal  
FwdRateIpc

LogNormal  
FwdRateiBalland

LogNormal  
FwdRateEuller  
Constrained

ConstrainedEvolver

LogNormal  
FwdRateBalland

LogNormal  
CotSwapRatePc

LogNormal  
SwapRatePc

The evolver needs

MarketModel

Correlation

BrownianGenerator  
DriftCacluator  
EvolutionDescription  
CurveState

SobolBrownian  
Generator

MTBrownian  
Generator

BrownianGenerator

MC Simulation

MarketModelEvolver

LmCorrelation  
Model

LmLinearExponential  
CorrelationModel

LmConstWrapper  
CorrelationModel

LmExponential  
CorrelationModel

LFM Corelation

LFM Process

LmVolatility  
Model

LmConstantWrapper  
VolatilityModel

LmLinearExponential  
VolatilityModel

LmExtLinear  
ExponentialVolModel

LmFixed  
VolatilityModel

LFM Volatility

StochasticProcess

LfmForwardModelProcess  
  
fixingTimes()

Pricing Engine

Covariance Parameterization

LFM Model

LfmHullWhite  
Parameterization

LfmCovariance  
Proxy

LfmCovariance  
Parameterization

AffineModel

CalibratedModel

LiborForward

Model

GenericModel  
Engine

LfmSwaption  
Engine