Certificate in Quantitative Finance

Jan 2011 Cohort

Module 6 Assessed Assignment

Luigi Sgro

HJM Model ... 3
Abstract ... 3
Model ... 3
Pricing methodology ... 3
Stochastic processes and volatility functions ... 4
Simulation of the forward rate curves ... 5
Pricing of caps/floors ... 5
Stopping condition ... 6
Implementation ... 7
PCA ... 7
Volatility functions ... 8
Monte Carlo simulation ... 8
Results ... 8
PCA ... 8
Monte Carlo ... 9
Sensitivity of the model to parameter changes ... 10
Uncertain Volatility and Static Hedge ... 11
Abstract ... 11
Model ... 11
Pricing methodology ... 12
Finite differences ... 12
Boundary conditions ... 12
Explicit method ... 12
Generalization for multiple maturities ... 13
Static hedging ... 13
Implementation ... 13
Optimization ... 13
Results ... 13
No hedge ... 14
Not optimized static hedge, long option ... 14
Not optimized static hedge, short option ... 14
Optimized static hedge, long option ... 14
Not optimized static hedge, short option ... 14
Price comparison ... 14
Appendix ... 16
Usage of the programs ... 16
Runtime ... 16
Syntax ... 16
Configuration ... 16
Building the code ... 17
Implementation notes ... 17
General packages ... 18
HJM packages ... 18
FDM packages ... 19
Layout of the project code ... 20

HJM Model

Abstract

An implementation of the HJM framework for pricing interest rate derivative is presented, with a description of the models used, and a discussion of sample results.
Historical forward rate data are analyzed with Principal Component Analysis, in order to find a reduced number of volatility functions ν_i which account for most volatility in the sample data; Monte Carlo simulation is performed, using multi-factor HJM with Musiela parameterization, to simulate future evolutions of the forward rate, and calculate a fair price for interest rate derivatives, like floors and caps.

Model

The multi-factor Heath, Jarrow & Morton framework models, for every maturity T on the forward curve, the instantaneous forward interest rate, defined as:

$$F(t,T) = -\frac{\partial}{\partial T}\log Z(t,T)$$

where $Z(t,T)$ is the value at time t of a zero-coupon bond with maturity T.

The Musiela parameterization, used in this implementation, formulates the same measure in terms of the time-to-maturity τ, rather than in terms of the maturity:

$$F(t,\tau) = -\frac{\partial}{\partial T}\log Z(t,\tau)$$

where $\tau = T - t$.

Within this framework, the forward rate curve evolves in time from a known state, measured at time t_0, following a random path driven by a C-valued stochastic process:

$$dF(t,\tau) = \left[\sum_{i=1}^{C} \nu_i(\tau) \int_0^{\tau} \nu_i(s)\, ds + \frac{\partial F}{\partial \tau}(t,\tau)\right] dt + \sum_{i=1}^{C} \nu_i(\tau)\, dX_i$$

where:

- $F(t,\tau)$ is the instantaneous forward rate at time $t+\tau$ evaluated at time t
- dX_i is the differential of the *i-th* Brownian process with unitary volatility
- $\nu_i(t,\tau)$ is the volatility function associated with the *i-th* Brownian process
- C is the dimension of the stochastic process

Pricing methodology

Instantaneous forward rate curve historical data is available as a realization of the discrete process:

$$F_{k,j} = F(t_j, \tau_k)$$

defined at each instant t_j with $j = \{1,2 .. N\}$ for the maturities τ_k with $k = \{1,2 .. M\}$.

These data are used as input to a process involving Monte Carlo simulation and pricing of interest rate derivatives.

The following sections provide a description of the processing steps of the pricing algorithm.

Stochastic processes and volatility functions

In order to define the optimal number C of stochastic processes underlying the forward rate curve evolution in time, and the volatility functions of these processes, the statistical method of Principal Component Analysis is used.

A general model for the evolution of this discrete multivariate process has M sources of randomness, one for each maturity of the curve.

The aim if PCA is to reduce the number of stochastic processes in the model, finding the minimum number of uncorrelated processes, with associated volatility functions, which are responsible for most volatility in the process.
To achieve this, the following steps are performed:

Difference vectors

M vectors of differences are calculated from the original data, isolating for each process sample the increment over the unitary time period:

$$\vec{D}_p: \{D_{k,j} = F_{k,j} - F_{k,j-1}\}$$

where k = {1,2 .. M}.

Covariance matrix

The covariance matrix is calculated for the vectors of differences, yielding the covariance of the stochastic increments for each couple of processes associated to the different maturities:

$$\Sigma_{p,q} = \mathrm{Cov}(\vec{D}_p, \vec{D}_q)$$

To obtain the yearly covariance, the covariance matrix is multiplied by the number of samples in the year (e.g. the number of days, if the sample was daily).

$$\Sigma_{year} = \mathrm{k}\Sigma_{day}$$

where:

- k is the number of daily samples per year

Covariance matrix eigenvectors

The eigenvectors and eigenvalues of the covariance matrix are found:

$$\Sigma = V \Lambda V^T$$

where:

- $V = \{\vec{V}_1 | \vec{V}_2 | \dots \vec{V}_M\}$ is the matrix composed by the M eigenvectors
- Λ is a diagonal matrix with the corresponding eigenvalues Λ_i on the diagonal

The shape of the various volatility components is represented by the normalized vectors $\vec{V}_i$, and the intensity of the volatility for each component is expressed by the square root of the corresponding eigenvalue $\sqrt{\Lambda_i}$.

Principal components of volatility

The contribution of each component to the total variance of the sample can be calculated by the ratio:

$$\frac{\Lambda_i}{\Lambda_1 + \Lambda_2 + \cdots + \Lambda_M}$$

By ranking the volatility components based on the eigenvalues, from the biggest to the smallest, it is possible to select the minimum number of components that account for the wanted percentage of variance, e.g. 90%.

Volatility functions

Once the minimum set of components has been selected, they are used to build the volatility functions to be used:

$$v_{i,k} = \sqrt{\Lambda_i} V_{i,k}$$

where:

- $v_{i,k}$ is the value of volatility for the i-th component at maturity τ_k
- $V_{i,k}$ is the k-th element of the i-th eigenvalue, corresponding to maturity τ_k

To obtain the continuous functions $v_i(\tau)$ from the discrete data available, a 3rd degree polynomial regression is used:

$$v_i(\tau) = a_i\tau^3 + b_i\tau^2 + c_i\tau + d$$

Simulation of the forward rate curves

Once the C stochastic components have been determined, the risk-neutral process can be repeatedly simulated with a discrete approximation of the model:

$$\hat{F}_{k,j+1} = \hat{F}_{k,j} + \left[\sum_{i=1}^{C} v_i(\tau_j) \int_0^{\tau_j} v_i(s)\, ds + \frac{\partial F}{\partial \tau}(t,\tau)\right]\delta t + \sum_{i=1}^{C} v_i(\tau_j)\, dX_i$$

Starting from the observed values $\hat{F}_{k,0} = F_{k,N}$ chosen to correspond to the last sample of historical data, up to a number of steps M reaching the maturity date of the last payoff of the instruments that must be valued.

Pricing of caps/floors

The value of a derivative can be calculated as the conditional expectation of the present value of its payoffs in a risk-neutral framework.
The payoffs of a cap/floor are distributed through the life of the instrument, and are calculated as:

$$CF_c(t_j) = max(L_m(t_j) - K_c, 0) \times m \text{ (caplet)}$$
$$CF_f(t_j) = max(K_f - L_m(t_j), 0) \times m \text{ (floorlet)}$$

where:

- m is the tenor of the reference interest rate
- t_j is the j-*th* payoff calculation reference times, with $t_{j+1} - t_j = m$

To calculate the fair value for a floor or a cap on the interest rate L_m, the sum the expected present value of all the caplet/floorlet individual cash flows is calculated:

$$V = \sum_{j=0}^{Q} \mathrm{E}\left[e^{\int_{t_0}^{t_{j+1}} F(s,0)ds} CF(t_j)\right]$$

where Q is the number of cash flows.

In the current framework, each simulation of the forward rate curve is defined as:

$$F_p(t,\tau)$$

where p is the simulation index, ranging from 1 to P.
The following process is used:

1. For every simulation p:
   a. For each payoff reference time t_j:
      i. Calculate the forward interest rate with maturity m at time t_j:
      $$L_m(t_j) = \int_0^m F_p(t_j, s)ds$$
      ii. Calculate the individual cash flow CF
      iii. Calculate the discount from present to cash flow payment time t_{j+1}:
      $$Z(t_0, t_{j+1}) = e^{-\int_{t_0}^{t_{j+1}} F_p(s,0)ds}$$
   b. Sum all the individual discounted cash flows to obtain one realization of the instrument value:
   $$V_p = \sum_j Z(t_0, t_{j+1})\, CF(t_j)$$
2. Calculate the average of the realized instrument values:
   $$V = \frac{1}{P}\sum_{p=1}^{P} V_p$$
3. Check the stopping condition of the algorithm.

Stopping condition

The algorithm can be configured to run:

- to a fixed number of iterations
- until a desired level of accuracy is reached

In the second case, an estimation of the standard error of the P-*th* calculated value is:

$$\epsilon_P = \sqrt{\frac{\left(\frac{1}{P}\sum_{p=1}^{P} V_p^2\right) - \left(\frac{1}{P}\sum_{p=1}^{P} V\right)^2}{P}}$$

This value decreases with the increase in number of the simulations, and represents the standard deviation of the P-*th* calculated value. In order to reach a given level of accuracy, in terms of maximum standard error ϵ_{max}, the following criteria can be used:

$$\epsilon_{P<}\epsilon_{max}$$

Implementation

In the following, the methodologies and algorithms used in the implementation of the HJM model are described.

PCA

Once the curve data time series has been loaded, a covariance matrix is calculated.
In order to find the eigenvalues and eigenvectors of the covariance matrix, the Jacobi method has been implemented.
It is an iterative method that proceeds by performing rotations of 2-dimensional subspaces in a symmetric matrix, zeroing an off-diagonal element for each rotation.
For each rotation:

$$A_{i+1} = Q_i^T A_i Q_i$$

where:
A_i is the i-th transformation of the original matrix
Q is the Jacobi rotation matrix:

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 & \cdots & 0 \\ 0 & 0 & \cos\varphi & \cdots & \sin\varphi & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 & \cdots & 0 \\ 0 & 0 & -\sin\varphi & \ldots & \cos\varphi & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}$$

It is an identical matrix, with columns (rows) p and q modified as above, where the angle φ is calculated:

$$\cot\varphi = \frac{a_{qq} - a_{pp}}{2a_{pq}}$$

The algorithm used performs iterations called *sweeps* by inspecting all the non-diagonal elements, looking for elements with absolute value bigger than a given threshold, which is configured as an algorithm parameter. For each element found, a rotation is performed, zeroing it.
The *sweeps* are repeated until all the non-diagonal elements are below the configured threshold.

When all the non-diagonal elements are close enough to zero, the algorithm stops, and the resulting matrix is approximately diagonal.
The eigenvectors are found by multiplying all the rotation matrices:

$$E = Q_1 Q_2 Q_3 \dots Q_D$$

When the process is complete the matrix E columns are the eigenvectors of matrix A.

Volatility functions

The volatility functions are implemented by fitting a 3rd degree polynomial to the chosen eigenvectors.
To perform the fitting, a library performing the the Ordinary Least Square algorithm has been used (Commons Math *OLSMultipleLinearRegression* [web: http://commons.apache.org/math/]).
The volatility functions have been integrated analytically, generating another polynomial function to be evaluated as integral.

Monte Carlo simulation

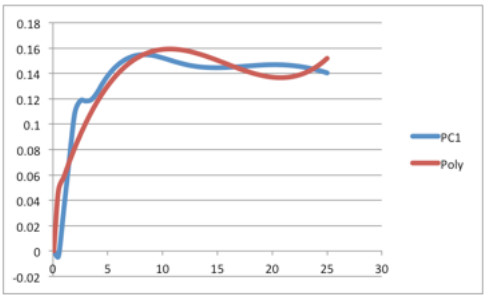
A quasi-random generator, the Mersenne Twister, has been used to generate the uniformly distributed vectors that are in turn transformed to Gaussian distributed vectors.
To generate Mersenne Twister vectors, a suitable library (Commons Math *MersenneTwister*) has been used.

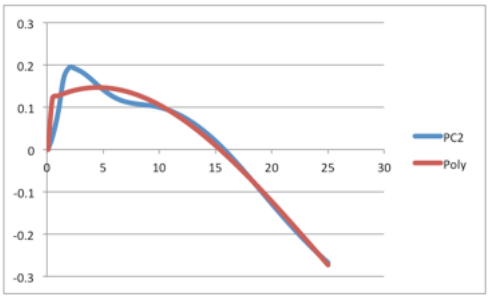
Results

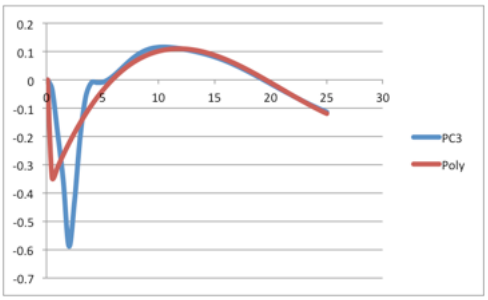
The input data for the simulation process is the historical instantaneous forward interest rate available from the Bank of England, from January 2005 to May 2011 (http://www.bankofengland.co.uk/statistics/yieldcurve/index.htm). The tenors used are 1 month, 6 months and each 6 months to 25 years.

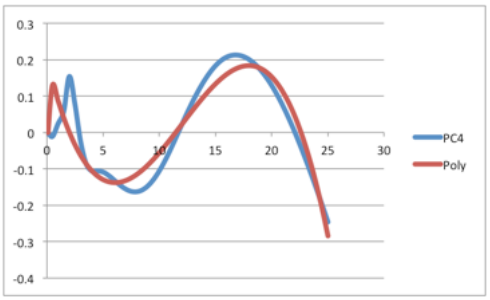
PCA

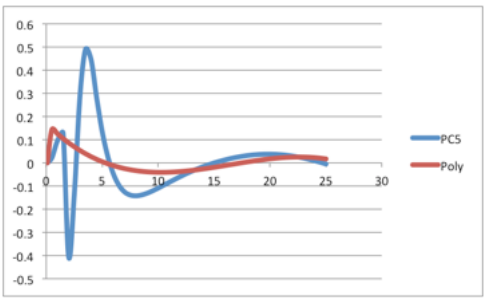
The PCA has been used to select the components responsible for at least 90% of the total variance of the sample. This resulted in 5 components, responsible for more than 94% of the total variance.











Monte Carlo

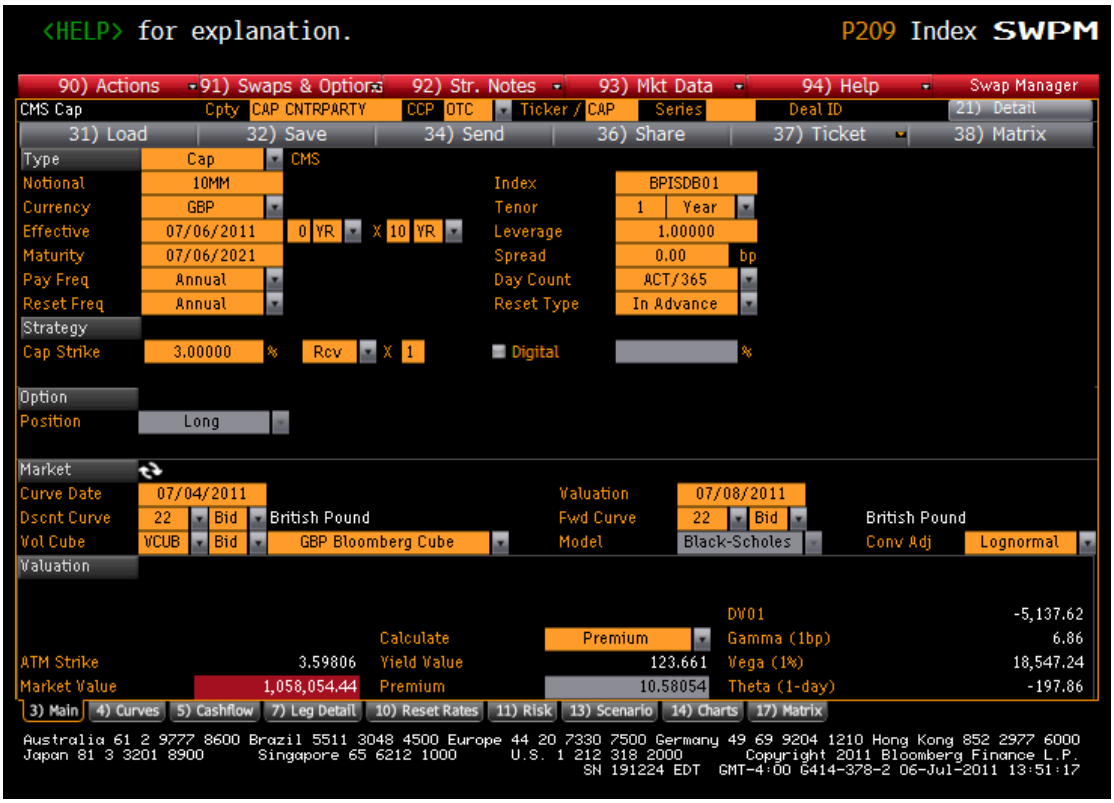
All the following results have been obtained with the following parameters:

- Max standard error: 5.0E-4
- Number of components: 5
- Degree of polynomial for the volatility components: 3

The table shows the results of the simulation side by side with corresponding results obtained from Bloomberg.

Sim #	type	maturity	tenor	strike	ATM strike	Bloomberg	Simulation	%diff
1	CAP	5	0.25	0.02	0.0223	0.037919	**0.042042**	10.31%
2	CAP	5	0.5	0.03	0.0249	0.026867	**0.018993**	-34.34%
3	CAP	5	0.5	0.04	0.0249	0.016293	**0.007116**	-78.41%
4	CAP	10	1	0.03	0.0359	0.105805	**0.089466**	-16.73%
5	CAP	10	1	0.04	0.0359	0.066559	**0.049906**	-28.60%
6	FLOOR	5	0.25	0.02	0.0223	0.026622	**0.014511**	-58.89%
7	FLOOR	5	0.25	0.03	0.0223	0.058166	**0.040027**	-36.95%
8	FLOOR	5	0.5	0.03	0.0249	0.050941	**0.039664**	-24.89%
9	FLOOR	5	0.5	0.04	0.0249	0.087781	**0.075185**	-15.46%
10	FLOOR	10	1	0.03	0.0359	0.054602	**0.046894**	-15.19%
11	FLOOR	10	1	0.04	0.0359	0.101367	**0.093083**	-8.52%

The table shows that the evaluation made with Monte Carlo simulation is in general lower compared with the results obtained with the default settings on Bloomberg, especially for options OTM, like #3. This can be due to the shape of volatility used by the Bloomberg screen, which is probably backed out from other traded instrument on the same underlying.



This picture shows the setup used for the evaluation of a cap on Bloomberg.

Sensitivity of the model to parameter changes

By improving the amount of volatility accounted for, and therefore the number of stochastic components used, and by raising the degree of the polynomials, the results have drifted slightly, showing a certain degree of sensitivity of the model to a more accurate representation of the volatility.

% variance	# components	degree of polynomial	value
94%	5	6	0.00854
95%	6	8	0.00942
97%	7	10	0.01024
98%	8	20	0.01027
94%	***5***	***20***	***0.00965***

The results for simulation #3, with a different choice of number of components and polynomial degree, show a marked drift upward with the improving of the accuracy of the volatility representation.

Uncertain Volatility and Static Hedge

Abstract

A generalization of the Black Scholes model is used, where the model volatility parameter is not assumed as deterministic, and no probability distribution is given for it.
Based on the simple assumption that volatility of the underlying will stay within a specified interval, it is possible to calculate a *worst-case* and a *best-case* evaluation, by taking, for each step of the calculation, the value of the volatility which yields the lowest or highest value of the derivative.
By leveraging the non-linearity of the pricing model, which implies that changes to the hedging portfolio will have different effects on the two sides of the trade, a strategy of static hedging can be implemented, that can be further improved by means of numerical optimization, to find the hedging ratios that yield the best value for the derivative.

Model

The model used is Avellaneda, Levy & Parás and Lyons, which extends the Black Scholes model for the evaluation of a derivative to cases when the volatility of the underlying asset is uncertain.
The original Black Scholes equation is modified by making the volatility function $\sigma(.)$ depend on the second partial derivative of the instrument value.
There are two complementary equations: one for the lowest value that the derivative can take (worst-case), one for the highest (best-case), given an interval for the underlying asset volatility during the life of the instrument to be priced.
Equation for the worst case:

$$\frac{\partial V^-}{\partial t} + \frac{1}{2}\sigma(\Gamma)^2 S^2 \frac{\partial^2 V^-}{\partial S^2} + rS\frac{\partial V^-}{\partial S} - rV^- = 0$$

where:

- $V^-$is the lowest possible value of the derivative within the volatility interval
- $\Gamma \equiv \frac{\partial^2 V^-}{\partial S^2}$
- the volatility function:

$$\sigma(\Gamma) = \begin{cases} \sigma^+ if\ \Gamma < 0 \\ \sigma^- if\ \Gamma > 0 \end{cases}$$

Best case:

$$\frac{\partial V^+}{\partial t} + \frac{1}{2}\sigma(\Gamma)^2 S^2 \frac{\partial^2 V^+}{\partial S^2} + rS\frac{\partial V^+}{\partial S} - rV^+ = 0$$

where:

- $V^+$is the highest possible value of the derivative within the volatility interval
- $\Gamma \equiv \frac{\partial^2 V^+}{\partial S^2}$
- the volatility function:

$$(\Gamma) = \begin{cases} \sigma^{+} if\ \Gamma > 0 \\ \sigma^{-} if\ \Gamma < 0 \end{cases}$$

Pricing methodology

The method used is Explicit Finite Difference, which numerically integrates a PDE by approximating the dynamics of the portfolio value by means of finite differences calculation.

Finite differences

Given a grid of points (S_i, t_k) spanning over a range of values of the underlying asset S and a time interval interesting for the evaluation $[t, T]$, the value of the portfolio is approximated by:

$$V_i^k \simeq V(S_0 + i\delta S, t_0 + k\delta t,)$$

The members of the model equations can be approximated as follows.
Theta

$$\frac{\partial V}{\partial t}(S,t) = \frac{V_i^k - V_i^{k+1}}{\delta t} + O(\delta t)$$

Delta

$$\frac{\partial V}{\partial S}(S,t) = \frac{V_{i+1}^k - V_{i-1}^k}{2\delta S} + O(\delta S^2)$$

Gamma

$$\frac{\partial^2 V}{\partial S^2}(S,t) = \frac{V_{i+1}^k - 2V_i^k + V_{i-1}^k}{\delta S^2} + O(\delta S^2)$$

Boundary conditions

The range of the underlying asset S is chosen large enough to approximate the values of each instrument of the portfolio with a simple expression of S and t.
A call vanilla option, for example, at S=0 can be approximated with 0. For values of S much higher of the option strike E (e.g. 2E) its value can be approximated by:

$$V_I^k = I\delta S - Ee^{-rk\delta t}$$

The boundary condition at maturity is represented by the payoff function of the instrument:

$$V_i^K = CF(S_i, T)$$

Explicit method

When using the explicit method, the calculations are performed backward in time from the boundary at maturity towards the present time, where the present value of the portfolio will be calculated for each value of S.
To calculate the value of the portfolio at one grid point (S_i, t_k) from the points in the previous step, the delta and gamma are calculated at (S_i, t_{k+1}), then the model equation is used to calculate the theta.

$$\frac{\partial V_i^{k+1}}{\partial t} = -\frac{1}{2}\sigma(\Gamma)^2 S_i^2 \frac{\partial^2 V_i^{k+1}}{\partial S^2} - rS_i \frac{\partial V_i^{k+1}}{\partial S} + rV_i^{k+1}$$

From the value of theta calculate V at (S_i, t_k):

$$V_i^k = \frac{\partial V_i^{k+1}}{\partial t}\delta t$$

Generalization for multiple maturities

To evaluate a portfolio containing instruments with different maturities, the cash flows corresponding to each instrument payoff function must be taken into account into the PDE integration.
For maturity T_p of the p-th instrument, an amount corresponding to the payoff cash flow must be added to the value of the portfolio. The equation above becomes:

$$V_i^k = \frac{\partial V_i^{k+1}}{\partial t}\delta t + \sum_{p=1}^{P} CF_p\left(S_i, T_p\right)\mathbf{1}_{\{t_k = T_p\}}$$

Static hedging

By hedging an exotic instrument with vanilla options, thanks to the non-linearity of the pricing model, it is possible to vary substantially the value of the instrument by varying the hedge components. In fact the value of the portfolio is in general not the same as the sum of the value of the individual instrument contained in it.
By leveraging this fact, it is possible to build portfolios that yield an instrument value optimized for a specific trading strategy.
For example, in case of a long trade, the worst-case value of the instrument can be maximized, by varying the quantity of the hedge components.
In a short trade, instead, a different quantity can be chosen for the hedge components, to minimize the best-case evaluation of it.
These two different prices can then be used to build a *bid-ask* quote.

Implementation

The Finite Difference – Static Hedging model has been implemented from scratch without any external library, except for the optimization stage.

Optimization

In order to optimize the static hedging, since the non-linear function has potentially several non-differentiable points, a heuristic technique has been used: Nelder Mead optimization.
A library (Commons Math *NelderMead* [web: http://commons.apache.org/math]) has been used for this purpose.

Results

The algorithm has passed a sanity check by pricing correctly vanilla options with parameters:

- Interest rate: 0.04
- Volatility: 0.22

- Stock price: 100
- Maturity: 1y

	Call @100	Call @80	Put @100	Put @110
Black-Scholes	10.688	24.223	11.544	12.133
FDM	10.686	24.222	11.542	12.131

Pricing of a "cash-or-nothing" binary option with static hedge, with these parameters:

- Interest rate: 0.04
- Volatility Min: 0.2
- Volatility Max: 0.3
- Stock price: 100
- Maturity: 1y

No hedge

Call binary option	1
Worst case long position (bid)	0.4036
Worst case short position (ask)	0.6030

Not optimized static hedge, long option

Call binary option	1
Call vanilla @90	-0.05
Call vanilla @110	0.05
Worst case value	0.4549

Not optimized static hedge, short option

Call binary option	-1
Call vanilla @90	0.05
Call vanilla @110	-0.05
Worst case value	-0.5525

Optimized static hedge, long option

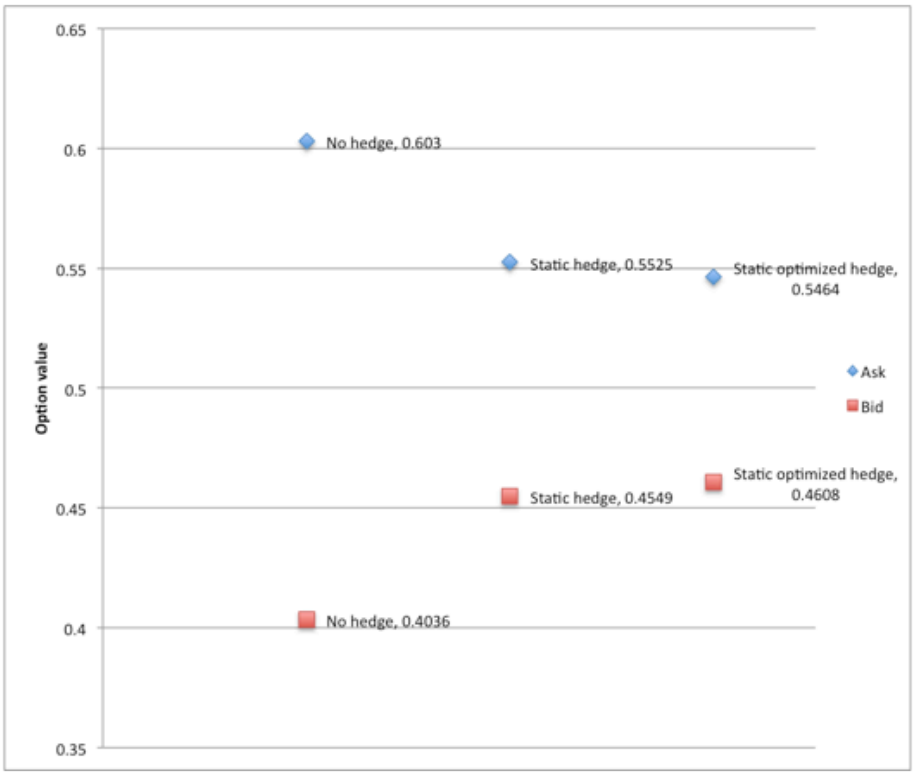
Call binary option	1
Call vanilla @90	-0.061
Call vanilla @110	0.056
Worst case value	0.4608

Not optimized static hedge, short option

Call binary option	-1
Call vanilla @90	0.055
Call vanilla @110	-0.059
Worst case value	-0.5464

Price comparison

	No hedge	Static hedge	Opt. static hedge
Ask	0.6030	0.5525	0.5464
Bid	0.4036	0.4549	0.4608


0.65
0.6
0.55
0.5
0.45
0.4
0.35
Option value
No hedge, 0.603
Static hedge, 0.5525
Static optimized hedge, 0.5464
Static hedge, 0.4549
Static optimized hedge, 0.4608
No hedge, 0.4036
Ask
Bid

Appendix

Usage of the programs

Runtime

To run the code, a Java Virtual Machine version 1.6 or above is required. The latest version of the JVM can be downloaded at:
http://www.java.com/en/download/
Both programs, HJM simulation and FDM static pricing are contained in a single JAR file: `project6.jar`
Make sure that in the working directory there is a `lib/` directory containing:
`commons-cli-1.2.jar`
`commons-math-2.0.jar`

Syntax

This is the syntax of the command:
```
java -jar project6.jar <property_file_name>
```
Where <property_file_name> is the name of a configuration file provided, as per the following section.

Configuration

The configuration must be provided in a properties file, where the program specific parameters are assigned a value on separate lines, with the following syntax:
```
# comment
name=value
```

List of parameters for the HJM program:

job.type	hjm
input.file	name of the file containing the interest rate input data
input.loader.class	class name to be used for the specific input file format. For BoE: com.luigisgro.cqf.curve.loaders.BoECurveTimeSeriesFileLoader
input.timestep	fraction of year used as time step for input data
pca.eigenvectors.output.file	optional output file for ordered eigenvectors output of PCA and corresponding eigenvalues
pca.eigenvectors.output.file.separator	separator character for the fields of the output file (e.g. ";")
pca.variance.threshold	min variance to be reached when selecting principal components (0.0 – 1.0)
mc.evaluator.type	stderr or fixed
mc.iterations.max	max number of Monte Carlo iterations
mc.evaluator.stderr	max error in case stderr is used
mv.calendar.months	time horizon of the simulation
mc.timestep	number of days in a year
mc.random.generator.class	class name of the random generator com.luigisgro.cqf.montecarlo.JDKUncorrelatedStandardVectorGenerator or com.luigisgro.cqf.montecarlo.MersenneTwisterStandardVectorGenerator
hjm.volatility.polynomial.degree	degree of the polynomials fitting the eigenvectors used as volatility functions
hjm.derivative.type	zcb, fwd (forward rate), cap or floor
hjm.derivative.strike	strike for caps or floors
hjm.derivative.tenor	tenor of the caps, floors
hjm.derivative.maturity	maturity of the instrument

List of parameters for the FDM static hedging program:

job.type	fdm
fdm.underlying.volmin	minimum volatility used in the model
fdm.underlying.volmax	maximum volatility used in the model
fdm.underlying.value	value of the stock
fdm.underlying.step	delta stock
fdm.timestep	delta time
fdm.ir	interest rate
fdm.volatility.type	constant or uncertain (strategy to be used)
fdm.option.type	binary or vanilla (type of the option to value)
fdm.option.maturity	maturity of the option
fdm.option.position	number of options
fdm.option.direction	call or put
fdm.option.strike	option strike
fdm.hedge.type.N	type of option for the N-th hedging option
fdm.hedge.maturity.N	maturity of the N-th hedging option
fdm.hedge.position.N	position of the N-th hedging option
fdm.hedge.direction.N	call or put (direction of the N-th hedging option)
fdm.hedge.strike.N	strike of the N-th hedging option
fdm.hedge.price.type.N	automatic (Black-Scholes) or fixed
fdm.hedge.price.N	optional price for the N-th hedging option

Building the code

The project code has been developed in Java with the Eclipse 3.6 (Helios) developer suite.
An Ant build file build.xml is provided, to compile the classes and generate a JAR file.
The Ant file expects the following environment:

src/	directory containing the Java source
lib/commons-cli-1.2.jar	library for command line interface (http://commons.apache.org/cli/)
lib/commons-math-2.0.jar	math library (http://commons.apache.org/math/)

These are the Ant targets provided:

clean	removes compiled code and JAR file
compile	compiles only
dist	compile and generate JAR file (default target)

To start a default build:

```
ant
```

If the build is successful, the project6.jar JAR file will be created in `dist/`.

Implementation notes

The programs have been developed following an Object Oriented methodology, with the aim of writing correct code, which is also understandable, easy to test and to modify.
The main goals in the development have been:

- Correctness: stick to the theoretical model as much as possible.
- Readability: make clear what each class does with a careful choice of names and avoiding excessive optimization.
- Efficiency: write fast code avoiding needless creation and copy of objects, use native floating-point arithmetic.
- Extensibility: write small classes and interfaces, highly cohesive and loosely coupled. Also, a good level of JavaDoc documentation has been provided.

The code is divided in packages, which reflect the functional areas in which the code can be divided.

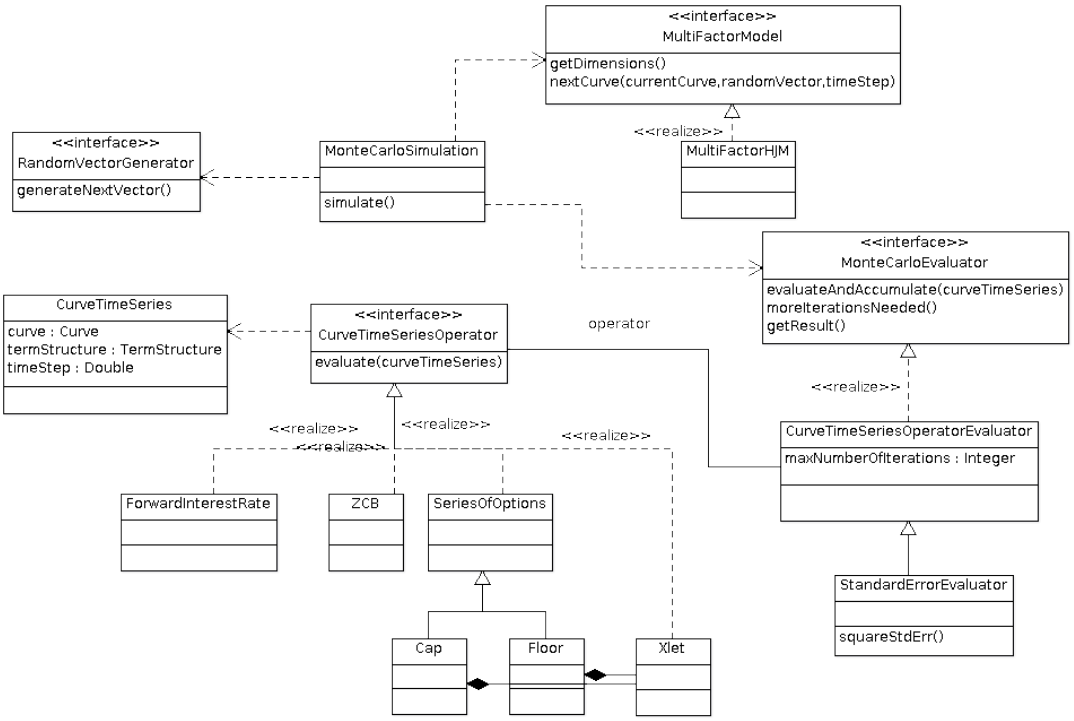
General packages

- `com.luigisgro.cqf.job`: This package contains the high-level implementation of the complete processing for: HJM: `HJMJob`; FDM: `FDMJob`, together with support classes to read job configuration from property files.
- `com.luigisgro.cqf.util`: A few utility interfaces and classes generic enough to not fit in any other package.

HJM packages

- `com.luigisgro.cqf.hjm`: Core package with the implementation of the Monte Carlo simulation framework specifically for the HJM model (`MultiFactorHJM`) and the implementation of some interest rate derivative models to be used with it.
- `com.luigisgro.cqf.curve`: Interest rate curves and curve time-series data types for HJM processing.
- `com.luigisgro.cqf.curve.loaders`: A collection of utility classes that can parse interest rate historical data and generate new `CurveTimeSeries`.
- `com.luigisgro.cqf.function`: A light model of scalar functions, used within the HJM framework to model volatility functions.
- `com.luigisgro.cqf.linalg`: A light linear algebra library, relying on lazy evaluation and immutable object where convenient.
- `com.luigisgro.cqf.montecarlo`: The Monte Carlo framework, in a generic implementation for curve time series, with a generic number of factors.
- `com.luigisgro.cqf.pca`: Implementation of the Principal Component Analysis needed for the HJM model.

The following UML class diagram shows the main classes of the HJM implementation and their relations.

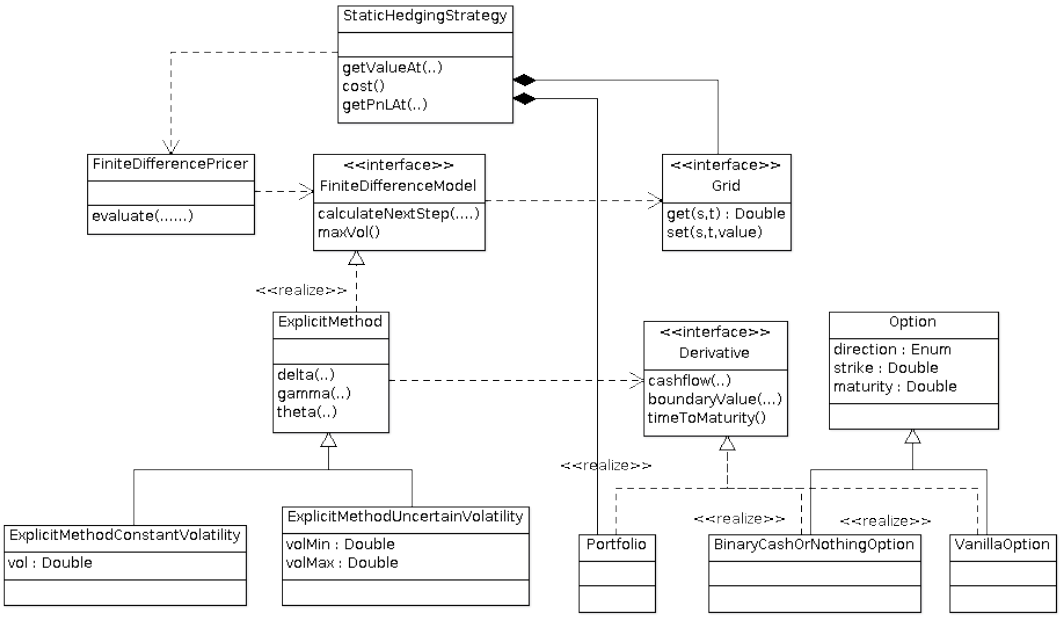


A complete documentation of all the classes is available in HTML format in the `doc/` directory of the project.

FDM packages

- `com.luigisgro.cqf.fdm`: All the Finite Difference Method code is contained in this package.

The following UML class diagram shows the main classes of the HJM implementation and their relations.



A complete documentation of all the classes is available in HTML format in the `doc/` directory of the project.

Layout of the project code

The project source code is organized with the following layout:

build.xml	Ant build file
src/	Java code of the programs
test/	Java code for the test suite
resources/	other resources needed to build the programs
data/	input data for the HJM simulations
diagrams/	UML diagrams
dist/	directory where the JAR file is generated and the needed libraries are copied
doc/	JavaDoc root
jobs/	collection of example configurations for HJM and FDM
lib/	libraries needed to compile and test the code