MASTER'S DEGREE THESIS

MSc in Mathematics Engineering

Title: Uncertain Volatility in QuantLib

Author: Carles Jou

Advisor: Josep J. Masdemont

Department: Matemàtica Aplicada 1

Academic year: 2009

Facultat de Matemàtiques i Estadística

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Universitat Politècnica de Catalunya
Master en Enginyeria Matemàtica

Uncertain Volatility Pricing in QuantLib

Master Project Dissertation

by Carles Jou

Advisor: Josep J. Masdemont

June 2009

Abstract

Quantitative finance has acquired a significant role in the markets during the past years with financial institutions increasingly hiring scientists, so-called quants, to discover patterns or relationships in prices and to implement strategies to exploit them.

One area with significant quants presence is derivative pricing. In this domain, the main goal is to consistently evaluate different instruments that depend on the same sources of risk. With this purpose, models of the risk sources are elaborated and arbitrage pricing theory is used to determine prices and replicating strategies. However, as far as the modelled risk is controlled and measurable, the risks associated to the specification of the model remain important and out of its scope. For the most widely used model class, diffusion models, we observe a significant dependency on the volatility parameter.

On the theoretical side, this project is focused on the study of an extension of diffusion models that integrates the volatility risk: the uncertain volatility model [ALP95]. In this model, the volatility parameter is no longer specified as a single value but as an interval. From this volatility interval, an interval for the derivative price is determined. The price envelope gives a consistent and reliable quantitative measure of the volatility exposure. A relevant feature of the model is that, in presence of changing convexity, the pricing equation becomes non-linear, making the valuation portfolio-dependent.

On the practical side, the aim of the project is to introduce QuantLib, an open source C++ library for quantitative finance and use it in the implementation of the uncertain volatility model. Given the (provisional) lack of official support documentation, an introduction to the structure and main features of QuantLib has been elaborated. The functioning of the library is illustrated with the integration of Uncertain Volatility pricing functionalities and their use in applications.

As a whole, the goal of the project is to provide an overview of the techniques and tools used in derivatives pricing and illustrate all the stages of a project in this field: from the generation of a theoretical model, to its implementation in pricing applications.

Contents

Chapter 1

Introduction

1.1 Motivation

Capitalist economies rely on capital markets as meeting points between financial resources and projects. Since the outcomes of the projects remain uncertain, capital holders will require their investments to be remunerated according to their risk-taking. Efficient capital markets should match the risk profile of investors and the projects in the market. To articulate this market of risk, financial securities provide at a certain price the rights over a share of the returns of a project. The valuation of these financial instruments will then be critical: in primary markets, it will determine the funding capabilities for new projects whereas in secondary markets it will adjust the risk allocation according to the information flow. To make the capital markets more dynamic and in some way more efficient, financial intermediaries issue new and more complex financial securities to adapt the existing projects to investors risk profiles.

In general, the evaluation of financial securities will not be objective in that it will heavily rely on the risk appreciation by the potential buyer. However, there is one particular class of financial securities that can be consistently valued and whose price can be enforced by arbitrage mechanisms, that is, potentially unlimited risk-less profits are possible until the price converges to its natural value. These are derivative securities: securities whose value depends on the price of a specified underlying. The idea is simple: we have only one source of risk and two traded securities depending on it; since their value should only depend on that same source of risk, they should be somehow related. Otherwise, one

could exploit this incoherence by buying at a low price, repackaging the risk at no cost and selling it at a higher price.

Derivative instruments have grown in importance in the last decades because they allow tailor-made products that fit the needs of asset managers and fund allocators. The issuers and traders of such products will need to evaluate them accurately so that no free money is given away by mispricing. At this point, the industry heavily relies on mathematical models mostly coming from the work of Black and Scholes. Generally these models make some assumptions on the behaviour of the underlying and then consistently price a set of securities depending on that underlying. Their main feature is that they provide hedging strategies to enforce a (range of) price. However, valuation is far from being a risk-less business and the inability of models to fully capture reality generates an additional risk.

In this context, the mission of financial institutions will be to package the risk, evaluate and hedge it in a particular framework, and take speculative positions with respect to the residual unhedgeable risks. To handle the complexity of the models, financial institutions heavily rely on information technologies. The growth of computing capabilities has boosted the size derivatives markets.

1.2 Aim and organization of the project

The aim of this project is to focus on a particular class of valuation models and study it both from a theoretical and from a practical point of view.

The first part of the dissertation focusses on the basics of securities pricing and particularly on derivatives pricing and hedging. In this part we will identify the main sources of risk and distinguish hedgeable and unhedgeable risk, particularly in the context of diffusion models.

Since volatility turns out to be the critical parameter that conveys most of the specification risk, special attention will be devoted to its modelling in the chapter 3 and we will particularly focus on a model that integrates part of this risk in the prices: the uncertain volatility model by Avellaneda, Levy and Paras.

For the practical side, we will explore an open source solution, QuantLib, very promising in terms of its functionalities and potential for the future of the industry. In particular,

chapter 4 will focus on the motivations and design of the library.

Finally, in chapter 5, we implement the model presented in the previous sections as a QuantLib module and test some applications of the model.

Chapter 2

Background

2.1 Securities pricing overview and modelling philosophy

Our first aim is to provide a framework to evaluate investments. According to basic financial theory, the main postulate for investment valuation is that the present value of an investment should be calculated as the sum of its future cash flows, discounted at a certain rate dependent on the time at which they occur. The discounting process conveys the opportunity cost of other investments such as cash deposits or, otherwise, a relative valuation of risk; for simplicity, we will present a basic modelling framework omitting this feature. The main issue remains that the future cash flows may be uncertain. We should then move to *expected* cash flows. We will need an estimate of possible future scenarios and a suitable measure over these scenarios to validly compute the expectations. But what future scenarios will we consider? What do we mean by a suitable measure? This is where models come in.

The key point in securities pricing theory is to assume that market prices contain all known information about the future and that markets are efficient in that the prices are determined so as to make any systematic profit impossible. This Efficient Market Hypothesis could be restated as follows: market prices are the expected prices taking into account all the available information. The market is thus implicitly using a measure that contains all this information about the future. It is natural then to value any claim using this same measure. By doing so, the achieved valuation could only be outperformed by luck. From

this point of view, securities pricing is merely the recovery of a market measure.

But first things first. We are talking about measures but we first need a probability space, a set on which to establish a measure. The modelling task starts by specifying future scenarios. But how will the future turn out? If asked about what will happen tomorrow, the first natural question should be: about what? A choice should be made about the drivers of uncertainty and how these may turn out. We may regard this uncertainty drivers as natural state variables which determine future scenarios.

Next, one should be able to describe the price of traded securities, in each of the scenarios, deterministically from the state variables.

Once we have specified the future scenarios and the value of basic securities in each of them, we want to recover the measure that gives the present price as the sum of the expected cash flows. In our simplified model, one could ask if such a measure will actually exist. It turns out that considering an arbitrage-free model will suffice to ensure the existence of suitable pricing measures.

With the measures obtained we can price other securities by computing and adding their expected cash-flows over a time interval. We are implicitly assuming that future scenarios of the modelled securities will be relevant for the new securities we are pricing. This turns out to be especially true in the case of contingent claims, whose value depend on the values of the underlyings and thus naturally share the set of future possible scenarios. The issue comes when we are left with various measures that fulfill the pricing requirements. These will be incomplete markets models as opposed to complete market models where only one measure satisfies the pricing requirements. In this situation, the best we can do is to provide a range for the price.

One of the most celebrated features of this whole pricing theory is that it provides hedging methods: the key point is that if the behaviour of different securities is known under different scenarios then one can build a portfolio that replicates to some extent one security from the others. The more closely related securities are, the easier it is to achieve a good

hedge. Thus, modelling, as stated by Derman [Der96], will be about finding similar securities to the ones we want to value. If the hedging is perfect, then the price can be enforced through arbitrage. Logically, a model will only provide a perfect hedge if it provides a single price. The converse is trivially true: if a model provides the means for a perfect hedge, then only one price will be admissible: the cost of the replicating strategy (this is the law of one price: two securities having the same payoffs must have the same initial value). So perfect hedging will be possible in and only in complete market models. In incomplete markets, we will be left with some additional risk. The different prices provided by the different measures can be understood as different ways to price this leftover risk.

In brief, a model should provide:

- a description of future scenarios according to some state variables,
- the market price of traded assets in these scenarios,
- an effective way to recover the suitable measures in order to compute expectations,
- an effective way to design hedging strategies.

2.2 Models and Fundamental Asset Pricing Theorem

We now turn to the mathematical formulation of such models.

Let $\tilde{\Omega}$ be the set of possible realizations of our world. Since this set is too abstract to work upon, the first idea is to restrict it through the use of a number of time dependent state variables whose evolution we can model. We first define a time interval $I \subset \mathbb{R}^+$, either discrete or continuous. We then define $\tilde{X}_i : \tilde{\Omega} \times I \to \mathbb{R}$ to be our state processes, $i = 1..n$. As we mentioned, we want to restrict our probability set to the behavior of our state variables, thus it is natural to define the following equivalence relationship: $\tilde{\omega} \equiv \tilde{\omega}' \Leftrightarrow \tilde{X}_i(\tilde{\omega}, t) = \tilde{X}_i(\tilde{\omega}', t)$ for all $i = 1..n$ and $t \in I$. We finally define $\Omega = \tilde{\Omega}/\equiv$ as our working probability set. The state processes X_i will be naturally defined as $X_i : \Omega \times I \to \mathbb{R}$, where $X_i(\omega) = \tilde{X}_i(\tilde{\omega})$, $\tilde{\omega}$ being one representative of the ω class. These new variables are well defined since the value of the state variable will not be dependent on the choice of the representative. We

note that Ω can be thought of as the different paths the state variables can jointly follow.

Next thing to do is to describe the behaviour of these state variables under a given probability. To do so, we will need a probability space, for which we need a σ-algebra $\mathscr{F}$ over Ω. The σ-algebra models in some way the information that we may get from the realization of Ω. It is then natural to make this σ-algebra time-dependent, as the more time elapses, the more able we will be to distinguish between different realizations. To model this feature, we will consider a filtration, that is, a sequence of σ-algebras $\{\mathscr{F}_t | t \in I\}$ such that $\mathscr{F}_i \subseteq \mathscr{F}_j$ for $i \leq j$. The resulting $(\Omega, \mathscr{F}, \mathscr{F}_i)$ is a filtered probability space. We then can define a probability $\mathbb{P}$ on this space such that the distribution of the state variables is known.

Now that we have a grip on what our newly defined world is like, we turn to model the asset prices in the scenarios we just defined. For each traded asset, we will model its price through a stochastic process defined as a deterministic function of the state variables: $S_j = f_j(X_1, \ldots, X_n)$ for $j = 1..m$. We note that sometimes we will be modelling other quantities, such as interest rates, and the derived asset prices will depend on this intermediate modelled variable.

If we turn to the recovery of the pricing measure, the first thing we should logically impose is that the new pricing measure $\mathbb{Q}$ can be used to retrieve the described pricing processes. This means that, at any time, their value can be computed as the expected future value with the known information. In mathematical terms: $S_j(s) = \mathbb{E}_{\mathbb{Q}}(S_j(t)|\mathscr{F}_s)$ for all $s \leq t$. This is the definition of martingales. So, for a measure to be acceptable to price securities, we will ask it to make the modelled pricing processes martingales. In fact, here we have local martingales; to have proper martingales we should also impose the technical condition $\mathbb{E}_{\mathbb{Q}}(|S_j|) < \infty$ for all $j = 1..m$. The measures that fulfil this requirement will be called martingale measures. In particular, we will be interested in equivalent martingale measures (EMM), those that preserve the views on possibility introduced by the original measure $\mathbb{P}$.

To find the pricing process for other claims, all we need to do is to find the expectation of their future value with respect to our pricing measures. If we denote by V_t the pricing process for claim X, we have, for a given pricing measure $\mathbb{Q}$, $V_t^{\mathbb{Q}} = \mathbb{E}_{\mathbb{Q}}(X|\mathscr{F}_t)$.

Finally, to cover the last modelling requirement, we must talk about hedging. The key point here is the use of martingale representation results. General martingale theory provides us with representation results for square integrable martingales. By considering the martingales set as a Hilbert space with a scalar product induced by the quadratic variation, one can represent a martingale in terms of a series of orthogonal martingales, with predictable coefficients with respect to the filtration. If the number of martingales needed to achieve the representation of the first martingale is finite, we then talk about the predictable representation property. Thus, in markets whose securities satisfy this basic property, we will be capable to achieve perfect hedges. Otherwise, the finite number of securities will not allow us to completely reproduce the new security and so we will be left with an unhedgeable risk. It goes without saying that complete markets will satisfy the predictable representation property.

We close this section with the most important result in pricing theory: the Fundamental Asset Pricing Theorem that provides necessary and sufficient conditions for the existence and unicity of pricing measures (EMMs).

1. The market is arbitrage-free if and only if there is at least one EMM; and
2. in which case, the market is complete if and only if there is exactly one such EMM and no other.

An interesting remark is that for no arbitrage opportunities to exist, one must have at least as much independent securities as sources of uncertainty. On the other side, completeness requires the converse. So there will only be room for efficient complete market models when the number of independent securities equals the number of sources of randomness.

When it comes to the market practice, it is reasonable to assume that no arbitrage opportunities exist, since if they existed, they would rapidly be materialized and disappear in the dynamics of the market. As for the completeness, there is empirical evidence that this condition does not hold[CGM01]. From the previous remark, this is quite reasonable as to properly model the behaviour of equities, one will need to introduce many sources of ran-

domness. Besides, another source of markets incompleteness is that trading strategies are limited: discrete trading and transaction costs stand against the completeness in practice.

To deal with market incompleteness, one can stick to Arbitrage Pricing Theory (APT) that we just described and that provides bounds for the prices (though generally this bounds will be insufficiently tight) or otherwise move to the expected utility maximization (EUM) framework, where the choice of the final pricing measure is made by optimizing a utility function of the investor (the drawback in this case is that the theory does not take into account the views of other market participants; besides, traders do not write down their utility function).

2.3 Diffusion models

The most popular class of models is the one that assumes that the state variables' behaviour can be modelled by diffusion processes.

The appeal of diffusion models comes from their analytical tractability and their use in the work of Black-Scholes and previously Bachelier for equity modelling.

The key feature of these models is their use of Brownian motion (or Wiener processes) to describe the randomness.

Definition: The process $W = (W_t : t \geq 0)$ is a $\mathbb{P}$-Brownian motion if and only if:

1. $W_0 = 0$
2. W_t is almost surely continuous
3. W_t has independent increments with $W_t - W_s \sim N(0, t-s)$ (for $0 \leq s < t$).

Following the modelling agenda established in the previous sections, we start by definig our state processes: let $W = (W^1, \ldots, W^n)$ where the W_i's are independent Brownian motions. Vector W collects all the randomness of our model. By repeating the construction of the previous section, we get a probability space formed by the paths that these state

variables may jointly follow. Over this probability space, the Brownian motions are implicitly defining a probability measure. This measure can be recovered using joint likelihood functions: suppose we take a time-mesh and we set the values of the variables at the times of the mesh. Then, since we know the distribution of each variable at each time, we can recover a joint likelihood for the variables to take such values. Taking the mesh to the limit, we can compute the likelihood of any possible path, which can be then extended to any set of paths. Thus, by saying that our state processes behave like Brownian motions, we are implicitly defining a whole probability space along with a measure which we will call $\mathbb{P}$.

Next step in our construction is to describe the behaviour of the basic assets from the state variables. We will model their pricing processes as stochastic processes adapted to the n-dimensional brownian motion of W.

> Definition: A stochastic process adapted to n-dimensional Brownian motion is a continuous process S_t, $t \geq 0$, such that S_t can be written as
>
> $$S_t = S_0 + \sum_{i=1}^{n} \int_0^t \sigma_i(s) dW_s^i + \int_0^t \mu_s ds,$$
>
> where $\sigma_1, \ldots, \sigma_n$ and μ are random $\mathscr{F}$-previsible processes such that the integral $\int_0^t (\sum_i \sigma_i^2(s) + |\mu_s|) ds$ is finite for all times t (with probability 1). The differential form of this equation can be written
>
> $$dS_t = \sum_{i=1}^{n} \sigma_i(t) dW_t^i + \mu_t dt.$$

Once we have the description of the behaviour of our basic assets, we turn to the specification of equivalent measures that make these price processes martingales. The Cameron-Martin-Girsanov (CMG) theorem plays a key role in this step as it provides an equivalence, up to technical conditions, between drift and measures. This result proves to be particularly useful as one can characterize martingales in terms of the drift (martingales are driftless processes up to a technical condition on the quadratic variation). So the CMG theorem reduces the problem of finding suitable measures to the problem of finding the changes of measure that eliminate the drift. It provides us thus with an "effective way to recover the pricing measures".

> Theorem (Cameron-Martin-Girsanov): Let $W = (W^1, \ldots, W^n)$ be n-dimensional $\mathbb{P}$-Brownian motion. Suppose that $\gamma_t = (\gamma_t^1, \ldots, \gamma_t^n)$ is an $\mathscr{F}$-previsible n-vector

process which satisfies the general growth condition $\mathbb{E}_{\mathbb{P}} \exp(\frac{1}{2}\int_0^T |\gamma_t|^2 dt < \infty$, and we set $\tilde{W}_t^i = W_t^i + \int_0^t \gamma_s^i ds$. Then there is a new measure $\mathbb{Q}$, equivalent to $\mathbb{P}$ up to time T, such that $\tilde{W} := (\tilde{W}^1, \ldots, \tilde{W}^n)$ is n-dimensional $\mathbb{Q}$-Brownian motion up to time T. The Radon-Nikodym derivative of $\mathbb{Q}$ with respect to $\mathbb{P}$ is

$$\frac{d\mathbb{Q}}{d\mathbb{P}} = \exp\Big(-\sum_{i=1}^n \int_0^T \gamma_t^i dW_t^i - \frac{1}{2}\int_0^T |\gamma_t|^2 dt \Big).$$

The converse is also true. Since we want equivalent measures $\mathbb{Q}$ under which all the discounted asset prices are $\mathbb{Q}$-martingales simultaneously, all we have to do is to find the γ that eliminates the drift in our processes. Then, we will be able to recover the correspondent pricing measure through the Radon-Nikodym derivative characterized in the CMG theorem.

The drift γ we have to add in this case must satisfy:

$$\sum_{i=1}^n \sigma_{ij}(t)\gamma_t^j = \mu_t^i, \text{ for all } t,\ i = 1, \ldots, n.$$

If we add here a discounting process with respect to a (riskless) security with drift r_t and we require the discounted assets to be martingales, this amounts to impose on the assets processes to have the same drift as the numeraire security. In such case, the drift we have to add must satisfy:

$$\sum_{i=1}^n \sigma_{ij}(t)\gamma_t^j = \mu_t^i - r_t, \text{ for all } t,\ i = 1, \ldots, n,$$

or in matrix terms, where we define Σ_t to be the matrix $(\sigma_{ij}(t))_{ij}$:

$$\Sigma_t \gamma_t = \mu_t - r_t \mathbf{1}.$$

This formulation is particularly interesting if one tries to draw some similarities with the Capital Asset Pricing Theory (CAPM): in this theory, risk is measured as variance and it has to be rewarded with returns over the riskless assets. The ratio between risk and reward is the market price of risk and is measured as the extra return over the riskless asset per variance unit. This is exactly the same formula we have obtained for γ_t (should it exist). It is then natural to call γ_t the "market price of risk". If we push further this observation, we notice that once we make a change of measure, γ_t is fixed and so, every asset in the economy should become a martingale under the same change of measure (otherwise, the no-arbitrage condition would be violated). This amounts to say that all traded securities

should have the same market price of risk in order to avoid arbitrage.

We must remark that the existence and unicity of γ_t depend on the actual values of the parameters Σ_t, μ_t and r_t. The existence of several solutions for this equation is equivalent to having several pricing measures, that is, an incomplete market model. One could then say that the different prices that we can obtain correspond to the different prices of risk and so to the risk aversion of the investor. We note that if Σ_t is invertible, then there is a unique γ_t and under the martingale measure, γ_t is zero; so the pricing measure is the one that has null market price of risk and so is known as risk-neutral.

Once we have the pricing measures, one can obtain the price by computing the expectations with this measure. The ability to obtain closed forms will mostly depend on the actual values of the model parameters. Later on, we provide a couple of examples.

We finally turn to the issue of hedging. Here, we focus on those market models where a complete hedge is possible, that is, those that satisfy the predictable representation property. This property can be stated as follows in the case of diffusion processes:

> Theorem (Martingale representation n-factor):
> Let $\tilde{W}$ be n-dimensional $\mathbb{Q}$-brownian motion, and suppose that M_t is an n-dimensional $\mathbb{Q}$-martingale process, $M_t = (M_1(t), \ldots, M_n(t))$, which has volatility matrix $\left(\sigma_{ij}(t)\right)$, in that $dM_j(t) = \sum_i \sigma_{ij}(t) d\tilde{W}_i(t)$, and the matrix satisfies the additional condition that (with probability one) it is always non-singular. Then, if N_t is another one-dimensional $\mathbb{Q}$-martingale, there exists an n-dimensional $\mathscr{F}$-previsible process $\phi_t = (\phi_1(t), \ldots, \phi_n(t))$ such that $\int_0^T (\sum \sigma_{ij}\phi_j(t))^2 dt < \infty$ with probability one and N can be written as
>
> $$N_t = N_0 + \int_0^t \phi_s dM_s.$$
>
> Furthermore, ϕ is (essentially) unique.

Let X be a derivative maturing at time T and let E_t be the $\mathbb{Q}$-martingale $E_t = \mathbb{E}_{\mathbb{Q}}(B_{T-1}X|\mathscr{F}_t)$ and $Z_t = B_t^{-1}S_t$. Here, B represents the discounting process mentioned in the introduction. If the matrix Σ_t is always invertible, then the n-factor martingale representation

theorem gives us a volatility vector process $\phi_t = (\phi_t^1, \ldots, \phi_t^n)$ such that

$$E_t = E_0 + \sum_{j=1}^{n} \int_0^t \phi_s^j dZ_s^j.$$

The invertibility of Σ_t is essential at this stage. Our hedging strategy will be $(\phi_t^1, \ldots, \phi_t^n, \psi_t)$ where ϕ_t^i is the holding of security i at time t and ψ_t is the bond holding. As usual, the bond holding ψ is

$$\psi_t = E_t - \sum_{j=1}^{n} \phi_t^j Z_t^j,$$

so that the value of the protfolio is $V_t = B_t E_t$. The portfolio is self financing in that

$$dV_t = \sum_{j=1}^{n} \phi_t^j dS_t^j + \psi_t dB_t.$$

To summarize, we have seen that diffusion models are easily tractable. They depend basically on two parameters: drifts and volatilities. One can look at these parameters as first and second order parameters. What we are doing in some way is letting the market fix the first order parameters (to make processes driftless) while we have a choice on the second order parameters. Since the measure changes to adjust the drift do not affect the volatility, pricing securities under diffusion models will be about speciying the volatities.

Examples

1) Black-Scholes

Probably the most famous example of diffusion processes is the Black-Scholes model for equities. In the simplest version of the model we have one source of uncertainty and one stock. The price process for the stock is $dS_t = S_t(\mu dt + \sigma dW_t)$. With these assumptions, stock returns are normally distributed. The model's first application was the valuation of contingent claims such as plain vanilla options for which it has proven to be extremenly successful. Under these assumptions, the model provides closed expressions for calls and puts whose popularity has grown to the extent that some products are quoted in terms of their implied volatility.

2) Hull and White

This model applies diffusion processes to describe the evolution of interest rates. In particular, it focusses on the evolution of the short rate, the interest for an instantaneous spot borrowing. One noticeable feature for this model is the mean reverting drift $dr_t = (\theta_t - ar_t)dt + \sigma dW_t$. This short-rate process is assumed to be given under the risk-neutral measure. The price of bonds can be obtained by calculating forward rates so as to make arbitrage impossible and then integrating on these forward rates. Pricing of other interest rate contingent claims can be obtained in a similar fashion.

We close this section with an ouverture to other models. If we consider the family of probability distributions that fulfill:

1. $X_0 = 0$;
2. X has independent and stationary increments;
3. $X_{t+s} - X_t$ has an infinitely divisible distribution.

If we take this distribution to be a normal, we obtain the browninan motion case we have just presented. We could have instead chosen any other distribution such as Poisson, Gamma, Inverse Gamma, CMY, ... Distributions with fatter tails are becoming increasingly popular, especially in the field of credit derivatives.

2.4 A complementary approach: PDE

For diffusion models there exists one complementary approach for contingent claims pricing based on partial differential equations (PDEs). We present the one-factor case, as the extension to multi-factor is straightforward and only adds complexity to the formulae.

We begin by assuming that we have one basic asset S (the extension to several assets could be easily done, taking care that the no-arbitrage condition still holds) and that the be-

haviour of this asset is given by the following equation:

$$dS_t = \mu_t dt + \sigma_t dW_t \tag{2.1}$$

Then, for a derivative security Z, its behaviour can be written as follows (using Itô calculus):

$$\begin{aligned} dZ_t &= \frac{\partial Z_t}{\partial t}dt + \frac{\partial Z_t}{\partial S_t}dS_t + \frac{1}{2}\frac{\partial^2 Z_t}{\partial S_t^2}(dS_t)^2 \\ &= \left(\frac{\partial Z_t}{\partial t} + \mu_t \frac{\partial Z_t}{\partial S_t} + \frac{1}{2}\sigma_t^2 \frac{\partial^2 Z_t}{\partial S_t^2}\right)dt + \sigma_t \frac{\partial Z_t}{\partial S_t}dW_t \end{aligned}$$

The idea is to form a portfolio such that the random part disappears. To do so, we take one unit of our derivative security and $-\partial Z_t/\partial S_t$ units of the underlying (the negative sign means we are holding a short position). The portfolio Π satisfies:

$$d\Pi = dZ_t - \frac{\partial Z_t}{\partial S_t}dS_t = \left(\frac{\partial Z_t}{\partial t} + \frac{1}{2}\sigma_t^2 \frac{\partial^2 Z_t}{\partial S_t^2}\right)dt$$

Since this portfolio is riskless, it follows from no-arbitrage arguments that it should have the same return as the riskless assets of our economy. If we set this riskless return to r_t, we get:

$$\begin{aligned} r_t dt = d\Pi/\Pi \quad &\Leftrightarrow \quad \left(Z_t - \frac{\partial Z_t}{\partial S_t}S_t\right)r_t dt = \left(\frac{\partial Z_t}{\partial t} + \frac{1}{2}\sigma_t^2 \frac{\partial^2 Z_t}{\partial S_t^2}\right)dt \\ &\Leftrightarrow \quad \frac{\partial Z_t}{\partial t} + r_t S_t \frac{\partial Z_t}{\partial S_t} + \frac{1}{2}\sigma_t^2 \frac{\partial^2 Z_t}{\partial S_t^2} - Z_t r_t = 0 \end{aligned}$$

This last equation is known as the Black-Scholes equation for the particular case where $\sigma_t = \sigma S$ and $r_t = r$. We note there is no dependence on the drift of the asset in this formula. To compute the price of the derivative, all we need to do is to solve this equation with the appropriate boundary conditions (final pay-offs). This can be done using numerical techinques such as finite differences which are proven to be equivalent to compute expectations on trinomial trees. In fact, the equivalence of the two approaches is given by the Feynman-Kac theorem that expresses the solution to this PDE in terms of expectations [Hau05].

2.5 Models and risk

As we have pointed out in the previous sections, one key feature of this pricing theory is that it provides the means to quantify and, to some extent, hedge the risk. In this section we review the practice of risk-management through the greeks and we finally make an ouverture to model-risk.

The basic idea of hedging has been already introduced in the previous sections, especially in the last one with the idea of a riskless portfolio. When we were building our riskless portfolio, we expressed the randomness of the derivative in terms of the randomness of the underlying. In this way, one can cover the risk of the derivative by holding a position in the underlying. This is what we call Δ-hedging and it corresponds more generally to the first order derivatives of the contingent claim with respect to the underlyings. The general idea is to represent the security we want to price in terms of other traded securities (this was what we were doing through the martingale representation theorem). An effective way to do so in models where we have a differential calculus is to compute the Taylor expansion of a contingent claim in terms of other traded assets and eliminate, by appropriately weighting the elements of the portfolio, as many random terms as possible. In models such as the classic Black-Scholes, a perfect hedge can be achieved because of the completeness. However, other situations (incomplete market models) do not allow the trader to completely hedge his position. Then, as we mentionned earlier, there is a leftover unhedgeable risk that will need to be priced.

Up to this point, we have been talking about in-model risk. We supposed that the market behaved exactly as predicted by our models and hedged in consequence. However, markets rarely behave as modelled and so other variables need to be taken into account. Thus, it will not be unusual to see traders computing derivatives with respect to theoretically constant parameters (such as the volatility in the Black-Scholes equation: this measure is known as vega ν). In a derivatives desk, traders will be Δ-hedging to cover the primary exposure but they will also be closely watching these other parameters and trade other securities so as to keep them in pre-specified limits.

One more remark needs to me made about hedging: our models will often assume (as it is

the case for diffusion models) a continuous time interval. The re-balancing of portfolios should be made continuously if we want the representation properties to hold and make the replicating portfolio strictly self-financing. However, hedging in continuous time is clearly impossible - not to mention the transaction costs. This introduces a hedging error, once again not contemplated by our models. The impact of hedging in discrete time has been studied by several authors and optimal strategies have been proposed, all of them though, leaving an unhedged risk [Wil06].

Finally we turn to the important issue of model-risk. We have been asuming that our market model was fairly reproducing the market. However, this may turn not to be true. Suppose we are working with a diffusion model. We calibrate that model at time t_0. At time t_1, we observe that prices differ from what we had been expecting; the implied volatility could be, for example, different that the one obtained in our first calibration. We are thus confronted to the following dilemma: on the one hand, we can believe that the difference in the implied volatility is due to a market mispricing; this consideration would lead us to invest all our capital in what we are seeing as an arbitrage opportunity. On the other hand, though, we can admit that our model needs to be recalibrated. With the new volatility obtained from the market, the prices are now different than those we expected and so there could be room for unexpected moves in our P&L. Wether we have to go for arbitrage (pseudo-arbitrage would be a more suitable term since we are in a model-dependent framework) or writedowns is an extremely important and unresolvable issue. In practice, the position will depend on the views of the trader.

To conclude, it is interesting to remark that the risk-neutral pricing theory is all but risk-neutral. In its essence, it starts from an arguable hypothesis (the market efficiency) and tries to recover a measure from the market with insufficient information (we don't have as much securities as possible outcomes for the future). To overcome this lack of information, restrictions are made over the possible outcomes and more assumptions are needed on the nature of the probability distribution. Essentially, we are setting a whole probability distribution and leave only the first order parameters to be fixed by the market. The risk that any of these assumptions turn out wrong is huge. Still, the theory has had an enormous appeal over practitioners for its underlying ideas (hedge derivatives exposure with underlyings or similar securities) and is currently accepted as a standard market practice. However, when used, one has to have in mind all the assumptions that are being made

and act consequently when these no longer hold (a recent example: the short selling ban approved by the Fed to face the markets crash invaliates most of Black-Scholes models as some arbitrage strategies depend on this possibility).

2.6 Further reading

For a general overview of securities pricing, the book by Baxter and Rennie [BR96] provides a short and complete introduction with a clear intuition and an elegant mathematical formulation. For a practitioner-oriented insight, the three-volume text-book by Wilmott [Wil06] explores the valuation theory and contrasts it to problems faced by market participants. In a similar approach, the book by Hull [Hul05] is an inescapable reference in this field. On the theoretical side, despite being more focused on interest rate derivatives, Rebonato [Reb98] is also an excellent reference.

Very interesting materials can also be found in the personal web page of Emmanuel Derman, former Head of Quantitative Strategies at Goldman Sachs. Particularly relevant in the context of this project are the comments on modelling and model risk as in [Der96]. Also on model risk, Rebonato's article [Reb03] provides a complete overview of the problem and its practical consequences.

Chapter 3

Volatility and uncertainty

3.1 Volatility and pricing

As mentioned in the previous chapter, most pricing models are built on top of Black and Scholes work and, as such, most are variations of diffusion models. We have also pointed out that diffusion models are governed by a two-parameter (drift and volatility) equation but only one parameter turns out to be relevant: namely, the drift becomes irrelevant as the measure is set to make the underlying processes martingales. Thus, volatility is left as the only source of uncertainty in the specification of the model[1].

This ultimately means that, once the model has been chosen to be a diffusion model, the prices will only depend on the specification of the volatility. Moreover, being a parameter rather than a variable, the risk of misspecification lies beyond the scope of the model, which means that it cannot be properly accounted for or hedged in that context. To quantify and make decisions on volatility issues we need models that integrate volatility. The goal of this chapter is to present one of such models, the uncertain volatility model; we start by briefly reviewing the basics of volatility and the main modelling approaches to situate the new model in context.

Before making any attempt to model volatility, we should have a clear idea of what it represents. Volatility is not a directly observable magnitude such as a price[2]. In fact, there are

[1] In the discounting process, another parameter usually comes in: the interest rate. However, most of the issues discussed in the context of volatility modelling can be translated in the context of interest rates.

[2] Although attempts have been made to establish volatility markets such as the VIX.

several definitions or types of volatilities that we review hereafter.

As defined in the context of diffusion models (see Chapter 2), volatility is the scaling factor applied to the Brownian motions that convey the uncertainty of the underlyings. In this context, it is merely a measure of the amount of uncertainty at a given time. We refer to this type of volatility as instantaneous volatility and, as such, it varies at every moment. Instantaneous volatility is the concept we need to model to be fully consistent with our approach.

Instantaneous volatility however is not observable, an important drawback when we need to model it. To overcome this difficulty we turn to the implied volatility. The idea here is to observe the market prices (of convex instruments) and then solve the inverse problem to determine the volatility. In this way, we can obtain volatility samples. We note that the relationship between prices and volatility is subject to a number of assumptions and so will be the validity of the samples obtained by these means. The main assumption is that the market prices are governed by the Black Scholes model, a statement that has often been questioned. A widely used argument against it is the volatility smile. Volatiliy smile is a phenomenon observed in the markets by which, at a given time, 'at the money' options have lower implied volatility than 'in the money' or 'out of the money' options. This finding is inconsistent with the model which assumes that all the instruments relating to a same underlying should be priced consistently, that is, their implied volatility should be the same. But still, the implied volatility plays a strong role in the markets and some option prices are directly quoted as implied volatilities.

These first two concepts are closely related to the construction of the models. However, instantaneous magnitudes are not easy to deal with and, often, practitioners turn to interval measures.

Historical volatility is an interval measure that gives the amount of randomness over a past period of time. This figure is generally computed as the standard deviation of the price over a time interval. As opposed to this, we have the forward volatility which is computed as the expected average for a future period of time and generally taken to be equal to the historical volatility adjusted to some predictions. This concept is very important in practice as it is the common input to practicioners Black Scholes model. The rationale behind this simplification is that one may elaborate an estimate for the volatility over a certain time period and then assume that the instantaneous volatilities are equal to that mean over time. It has to be noted that this approach is not always suitable, especially not

in the case of path-dependent options that can be triggered or de-activated by volatility peaks.

This said, the distinctions made above are only useful to establish a common vocabulary but they give in no way an indication on how volatility should be accounted for in practice. The following section covers this issue by giving several modelling approaches.

3.2 Modelling approaches

In the following we present the two most established modelling approaches and we discuss their main features. Then, a the uncertain volatility modelling approach, that can be considered as a midpoint of the previous two, is reviewed. The presentation is similar to the one that can be found in [Wil06].

3.2.1 Deterministic volatility surfaces

The first modelling approach is a purely deterministic one. It starts from the assumption that we know with certainty the future behaviour of volatility.

The simplest assumption in this context is to consider that the volatility will be constant over time. Although we should keep in mind that the real input to the Black Scholes model is the instantaneous volatility, this approach is close to estimating the average volatility in a similar way to that of the forward volatility measure. As far as this approach can be judged to be valid in an averaging context, it is clearly unacceptable for instruments with uncertain expiration or exercise such as barrier options.

Still in the deterministic approach, we can increase the complexity by specifying the volatility as a function of time. In this way, on one side, we can account for changes over time, particularly relevant for some instruments, but on the other side we face the problem of further specification of an unknown magnitude. In the same spirit, we can introduce a dependency on the price of the underlying to capture the observed smiles and skews.

In calibrating models, we always find a trade-off between the fidelity to past data and the adjustment to future predictions. The more we want a model to capture the features of historical data, the more parameters will have to be introduced and calibrated to this

data. The problem is that by introducing more parameters in the model, we also introduce more constraints for the future dynamics and, as models become increasingly complex, they easily diverge from observed values and more (complex) recalibrations are needed.

Deterministic models have enjoyed a great popularity with an increasing number of methodologies to model and calibrate volatility[Wil06]. Probably, the main advantage of this modelling philosophy is that it does not introduce another source of uncertainty and the models remain complete, with a unique price output in the end. Deterministic models give us the freedom to fully specify the volatility but it is this same freedom that constitutes their main flaw: good accurate predictions cannot be made by market agents according to efficiency hypothesis. Hence, these predictions will often turn to be wrong and so a reflexion is needed on the convenience to model in detail something that we do not even know in broad outlines.

3.2.2 Stochastic volatility

Stochastic volatility starts from the opposite premise: instead of eliminating a source of uncertainty by specifying a quantity that we do not fully know about, this approach suggests to model volatility in the same way that we model the underlying asset, that is through a stochastic process.

Generally, stochastic volatility models are also built upon diffusion processes. However, there is market evidence that volatility is not normally distributed. Volatility presents some particular features such as dependence on the underlying's price level or mean reversion that will need to be properly accounted for in the model. All these diffusion-based models share the same general equation for the volatility dynamics:

$$d\sigma = p(S,\sigma,t)\,dt + q(S,\sigma,t)\,dX \tag{3.1}$$

An important remark needs to be made at this point: we are introducing another state variable to our system and so, as mentioned in the previous chapter, we must specify the correlation between the two Brownian motions, the one that carries uncertainty on the underlying W_t and the one that drives the volatility process X_t. A model will be fully specified when this correlation ρ is given, along with the deterministic functions $p(S,\sigma,t)$ and $q(S,\sigma,t)$.

Regardless of the choices made for these parameters, we can obtain a general pricing

equation by considering a portfolio formed by the instrument we want to price, the underlying and an additional instrument to cover the new source of uncertainty and keep the model complete. Interestingly enough, the choice of this second instrument is theoretically irrelevant as the final equation will not depend on it but on a more general underlying magnitude: the market price of risk for volatility.

The resulting pricing equation is:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + \rho\sigma S q \frac{\partial^2 V}{\partial S \partial \sigma} + \frac{1}{2} q^2 \frac{\partial^2 V}{\partial \sigma^2} + rS\frac{\partial V}{\partial S} + (p - \lambda q)\frac{\partial V}{\partial \sigma} - rV = 0. \tag{3.2}$$

The equation is very similar to the Black Scholes equation 2.2. This suggests the existence of a risk-neutral drift $(p - \lambda q)$ for the volatility process which will actually be used for pricing.

The most popular models are Hull and White [HW87] which assumes uncorrelated uncertainty sources and risk-neutral dynamics for volatility given by $d(\sigma^2) = a(b - \sigma^2)dt + c\sigma^2 dX$; and Heston's [Hes93] where the volatility dynamics is given by $d(\sigma^2) = (a - b\sigma^2)dt + c\sigma dX$ and arbitrary correlation between the underlying and its volatility.

This approach to volatility modelling is appealing as it comes quite naturally from empirical studies of volatility. It seems much more reasonable to assume that volatility is not known in advance and that it is governed by another stochastic process. This approach is particularly suitable for options such as barriers. However, this approach has at least two important drawbacks: the first is that we need to specify more parameters and that these parameters become less and less observable. If we cannot even observe volatility, how can we hope to specify a volatility of the volatility? The second is that, unless we introduce other securities in our analysis, the model becomes incomplete.

3.2.3 Uncertain parameters

Finally, there exists an interesting intermediate approach that somehow covers the space between the previous two methodologies. This consists in considering that volatility is an uncertain parameter, which means that we do know in advance some bounds for the volatility and that we allow it to fluctuate within these bounds, regardless of any probability distribution. With this approach we have a deterministic part that sets the bounds and then some uncertainty left in that, within the bounds, anything can happen.

This approach looks very much like a best and worst case analysis and it will indeed provide two different prices that account for the uncertainty left. These prices are an upper and a lower bound but, more interestingly, they could be interpreted as some kind of bid-ask quote. As long as purchases are made at the lower price and sales at the higher price, delta hedging will suffice to eliminate volatility risk for $\sigma \in [\sigma_{min}, \sigma_{max}]$.

This approach would be merely a sum of best/worst scenarios if it was not for the non-linearity of the pricing equation. By taking advantage of this non-linearity, the bounds provided for the derivatives prices will depend on the portfolio held. This feature will be important in risk-management.

The following section is devoted to a more detailed review of the uncertain volatility model.

3.3 The uncertain volatility model

The uncertain volatility model was introduced by Avellaneda, Levy and Paras [ALP95]. As previously stated, this approach starts by specifying an upper and a lower bound for volatility that can be viewed as a confidence interval for the parameter: $\sigma \in [\sigma_{min}, \sigma_{max}]$. These bounds may be dependent on several magnitudes, namely time or underlying price level. For simplicity, we will, in the following, sometimes omit these possible dependences although generalizations will be straightforward. This model can be easily extended to any other parameters such as interest rates or correlations.

We recall from the first chapter on valuation that the key step in pricing derivatives is to find a measure that makes the underlying process a martingale and then compute prices as expectations with respect to this measure. In this case, we do not have one single underlying process but a parametric family depending on the volatility parameter that ranges over the specified interval. Therefore, we have one measure for each process and we find ourselves with a whole set of pricing measures.

Since we are interested in obtaining bounds for the derivative prices, it makes sense to state the problem as follows: the value of the derivative security should be comprised between

$$W^+(S_t, t) = \sup_P \mathbb{E}_t^P [\sum_{j=1}^{N} \exp^{-r(t_j - t)} F_j(S_t)],$$

and

$$W^{-}(S_t,t) = \inf_{P} \mathbb{E}_t^P [\sum_{j=1}^{N} \exp^{-r(t_j-t)} F_j(S_t)],$$

where P ranges over $\mathbb{P}$, the class of all probability measures on the set of paths $\{S_t, 0 \leq t \leq T\}$ and $F_j(S_t)$ are the streams of future cash-flows, known deterministic functions of the underlying.

The key observation of the authors is that this problem can be formulated in terms of stochastic control theory with control variable σ_t and that the solution can be achieved by solving dynamical programming partial differential equations. In the case of one single maturity date, the two extreme functions are obtained by solving the final value problem:

$$\frac{\partial W(S,t)}{\partial t} + r\Big(S\frac{\partial W(S,t)}{\partial S} - W(S,t)\Big) + \frac{1}{2}\sigma^2\Big[\frac{\partial^2 W(S,t)}{\partial S^2}\Big]S^2\frac{\partial^2 W(S,t)}{\partial S^2} = 0, \tag{3.3}$$

with $W(S,T) = F(S)$, where W^{+} is obtained by setting

$$\sigma\Big[\frac{\partial^2 W(S,t)}{\partial S^2}\Big] = \begin{cases} \sigma_{max} & \text{if } \frac{\partial^2 W}{\partial S^2} \geq 0, \\ \sigma_{min} & \text{if } \frac{\partial^2 W}{\partial S^2} < 0, \end{cases}$$

and W^{-} with

$$\sigma\Big[\frac{\partial^2 W(S,t)}{\partial S^2}\Big] = \begin{cases} \sigma_{max} & \text{if } \frac{\partial^2 W}{\partial S^2} \leq 0, \\ \sigma_{min} & \text{if } \frac{\partial^2 W}{\partial S^2} > 0. \end{cases}$$

The case of multiple maturities is similar. The equation is solved by time intervals, setting the final condition to the values obtained in the previous step.

The non-linear PDE that we have been solving is known as the Black-Scholes-Barenblatt (BSB) equation. It is noteworthy that it is merely a generalization of the Black Scholes equation and that the latter can be recovered as a special case when the interval reduces to a single point: $\sigma_{max} = \sigma_{min}$.

The non-linearity of the pricing equation causes the valuation of instruments to be portfolio dependent: as long as the residual payoff presents sign variations of $\Gamma = \partial^2 W/\partial S^2$, the resulting valuation will be different from the sum of the values of the instruments contained in the portfolio. Moreover, this new value proves to be lower than the value obtained by separately considering the instruments. The proof is straightforward from supremum and infimum inequalities. More interesting is the interpretation: if we take each instrument separately, the worst case analysis will take for each of them the value of the volatility that gives it a higher price. However, when we evaluate several derivative instruments on the same underlying altogether, we are constrained to use one single

volatility and, although this volatility may cause some instruments to be valued at their worst, it will happen that not all the instruments will be at their worst and hence the value of the portfolio will be lower than the sum of its instruments values.

In the paper, the authors provide an example that illustrates this point: we consider two options on the same underlying. We take European call options with the same maturity but different strike prices so as to generate a bull call spread, that is, buying the option with lower strike and selling the option with higher strike. The payoff diagram of this strategy is given in figure 3.1.

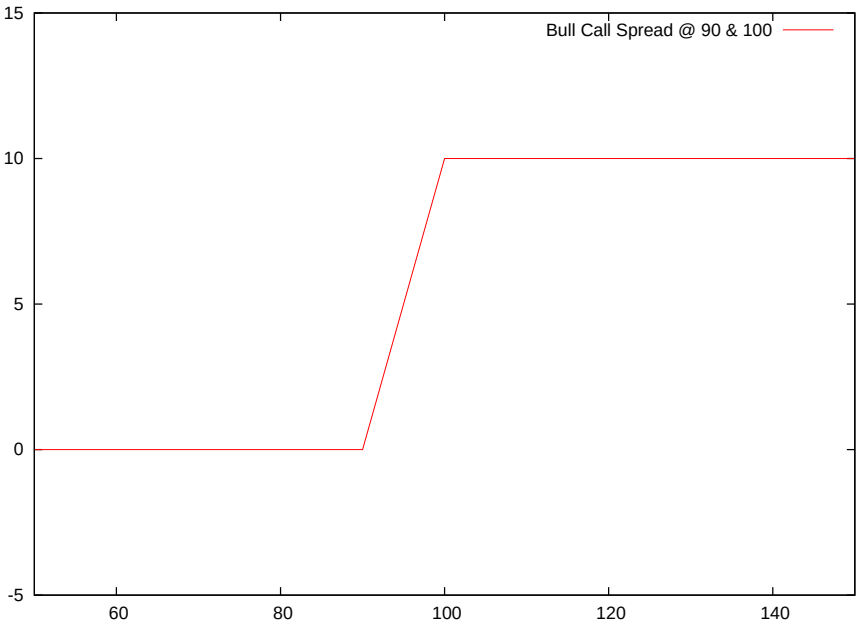

Figure 3.1: Bull call spread payoff diagram

The interest of this strategy is to profit from upward markets. Here, the assumption made is that markets will not go above a certain threshold and so part of the upward potential of the call option strategy would be useless and expensive. Instead of paying for it, we rather sell it by selling another call option. Hence, we only benefit from the upward move until the strike of the sold option is reached. The advantage is that we obtain money by selling an asset that is not valuable in our views of the market. Let us consider for the example the strikes to be at $K_1 = 90$ and $K_2 = 100$. This means that we would be expecting an appreciation of the asset from 90 to 100.

Now, we turn to the valuation of this option portfolio. The first thing to do is to determine the dynamics of the underlying. Here, we will assume it to follow a Black Scholes process with a risk free rate of 5%. Moreover, we assume the volatility to be comprised between σ_{min} = 10% and σ_{max} = 40%. With these assumption for the dynamics of the underlying we provide in the following several valuations of this bull call spread.

The first possible approach is to take a best/worst case analysis. By doing so, we value the option that we are buying with the highest volatility (which gives the highest price) and the option that we sell with the lowest volatility. However, we clearly see that this approach is not consistent, since we would be using different volatilities to price options on the same underlying at the same point in time. This could only make sense if we considered a strike-dependent volatility (see volatility smile theories).

The new approach proposed in the paper consists in valuing the portfolio as a whole. Using the methodology suggested by the authors, we roll-back the final scenarios according to the measures defined by the underlying processes taking at each step the volatility that values the portfolio at its worst. This approach, as previously discussed, will give tighter bounds for the bid and ask prices that will hedge the volatility risk away.

Finally, the authors also provide as benchmark the portfolio evaluated by standard Black Scholes method using one single volatility: the midpoint of the interval. The results obtained by this approach are presented in figure 3.2

The thick lines correspond to the upper and lower bounds provided by the proposed method. The outer dotted lines are the prices obtained using a best/worst case approach with the Black Scholes formula and the middle line is a Black Scholes price with mid volatility. Here we observe the anticipated features: the price range that is needed to integrate volatility risk in the model is tighter that a simple best/worst case analysis.

Let us turn now to the applications of the model. In the previous chapter we talked about model risk; the uncertain volatility model helps to quantify and deal with the risk related to the specification of volatility.

First, and straightforward, the quantification of the risk comes from the bid-ask quotes produced by the model. These quotes give information on extreme scenarios contrary to the original Black Scholes model which produces a simple quote with no reference whatsoever to its variability to volatility changes -the only measure is the artificial vega, theoretically inconsistent and misleading in practice.

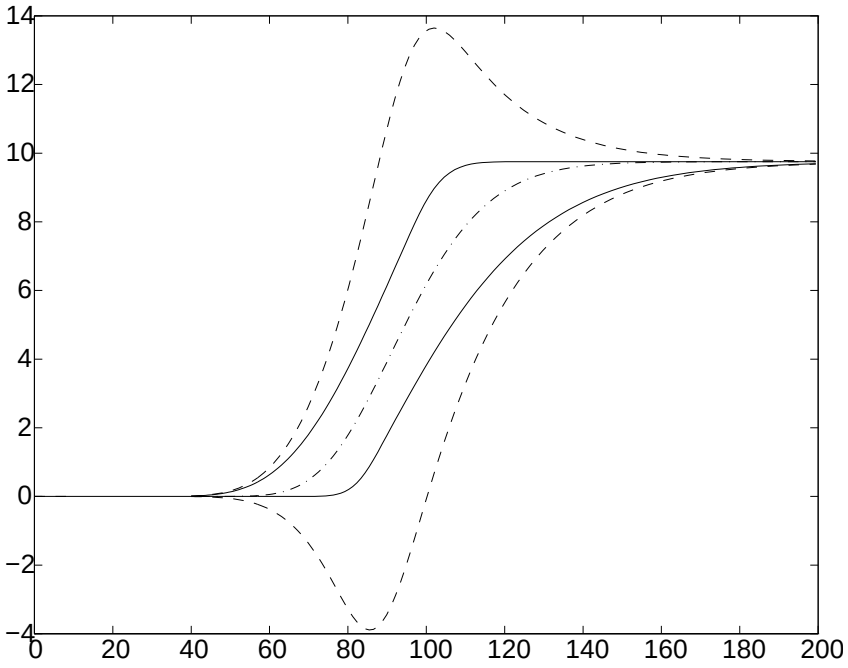

Figure 3.2: Bull call spread valuation

The second application is somehow a consequence of the quantification of the risk. The Lagrangian Uncertain Volatility model [APEK+96] will allow us to decide whether to statically hedge a position with other traded instruments. The rationale of such hedging is to fully eliminate the volatility risk as the payoffs of the hedging instruments offset those of the liabilities. However, often static hedging is expensive and in a classical framework it is difficult to decide if it is worthy. The Lagrangian Uncertain Volatility model uses the quantification of the risk made by the Uncertain Volatility Model to determine the optimal hedging policy.

We illustrate this application in the following with an example. Let us consider, for instance, a financial institution that has to quote a price for a given instrument. Whenever the quoted price is accepted by the counterparty, the financial institution will acquire an obligation or liability. If we consider the financial institution to be dealing with several clients, in the end, this will result in a residual liability which is the sum of all the liabilities generated by the contracts.

This residual liability may be reduced through static hedging, that is, by taking positions

on other liquidly traded instruments whose prices are exogenously given by the market and keeping these positions to maturity so as to offset the liabilities generated by the other operations. This comes at a cost, the cost of taking the positions on the hedging instruments. Whenever the hedging instruments are cheaper than the hedged liability, there is a (pseudo)-arbitrage opportunity. The key difference here with respect to Black Scholes classic arguments is that considering different volatility scenarios may alter the prices and, in some cases, the volatility risk transfer will be enough to justify a transaction.

We conclude this section with an example to illustrate the Lagrangian Uncertain Volatility. Consider a bank that has sold a European option. The bank has then a liability corresponding to this sold call. When evaluating this liability in the uncertain volatility framework, we would take the highest volatility as it is the one that corresponds to the worst scenario. Now, we consider that there is another call trading at an implied volatility higher than our worst case volatility. If the bank buys this second call, it will (statically) hedge a big part of the residual liability. The value of the residual liability will thus be diminished. The lagrangian uncertain volatility model will indicate the optimal quantity of hedging instruments to acquire or short taking into account a given price for the hedging instruments and a residual liability.

Chapter 4

QuantLib: an introduction

4.1 Aim and scope of the project

Finance is essentially an applied field where having solid theories about prices is essential but where the implementation of such theories is also of vital importance. In the context of constantly changing markets, quoting prices and detecting mispricings must take place fast and accurately. Errors may result in important losses while delays may imply losing clients. The words for this field are therefore fast and reliable.

In this section we focus on the implementation of financial models.

The majority of industry participants believe the implementations to be a competitive advantage factor. Accordingly, they use proprietary software, own-developed or trough closely watched externalization. This translates in an important amount of redundancy in the tasks as algorithms are implemented again and again and often not in the most efficient ways.

Here is where open source can play a relevant role in the years to come.

The idea is to build a common base, available to all practitioners, academics, students, regulatory bodies, and build upon this base. In this way, the standard implementations can be improved and best practices made available to everyone, thus favoring market efficiency which should be the final objective of all the players.

It has to be said that finance is basically about predicting the future and that it makes no

sense to standardize predictions. What is aimed for instead is to standardize the ways to translate, through a given model, a prediction into a price or into any other market measure. The fact of having one efficient implementation of Black Scholes does by no way mean that all banks will use the same input parameters. Ultimately, each financial institution will have to chose what models to use and what parameters to input in the model. By offering standards in implementations of widely used models, one allows the industry participants to focus on what should be their main task. Besides, keeping the solution open-source allows anyone to develop extensions suited to particular needs. Finally, it is easier to build regulations on standards.

It is with this whole perspective that a number of projects have been developed. The one that has gone further down the track is QuantLib.

According to the objectives stated in the project's website,

> The QuantLib project is aimed at providing comprehensive software framework for quantitative finance. QuantLib is a free/open-source library for modelling trading and risk management in real-life.

The scope of the project is restricted to the implementation of the most used financial models, leaving aside the obtention of market data, whose availability is another issue currently under discussion, especially as open-source gains presence in the financial community.

The project is maintained and developed by a community of programmers. There is a division between authors and contributors, the first being responsible for most of the design and releases while the latter basically submit code or comments according to some specifications. Contributors may also participate in the design through forums in which the authors regularly post and reply. A few companies have devoted significant resources to the development of this library, notably StatPro, a leading international risk-management provider, where the QuantLib project was born.

The QuantLib license is a modified BSD (Berkeley Software Distribution) suitable for use in both free software and proprietary applications, imposing no constraints at all on the use of the library.

4.2 Organization: an overview

In this section we provide an overview of the library structure and organization along with some quick guidelines to develop applications on top of it.

As of January 2009, QuantLib is in its 0.9.7 version and has practically no documentation. This makes it very hard for newcomers to understand how the project is built and it may discourage many from using this solution. The code comments compilated in a pseudo-documentation support are useless to obtain a general view of the structure of the library. However, efforts are being made to put remedy to this issue: one of the developers is writing a book that explains the main design choices and gives a broader view on the library's structure. However, this documentation is still incomplete and the following section aims at providing the reader with a general understanding of QuantLib from an "undocumented" approach.

QuantLib is written in C++, an object oriented programming language. In Appendix A we provide a short review of the main concepts of object oriented programming while in this section we focus on how they are put together to structure the library.

4.2.1 From the main requirements to the basic class structure

The main objective of QuantLib is to provide means to price financial instruments.

The two main requirements for the library are that it should be able to:

- support extensions of the type of instruments priced;
- support extensions of the means of pricing the instruments.

These two requirements naturally translate into classes: Instrument and Pricing Engine. These two classes are abstract classes that should be kept as general as possible and only include the attributes and methods shared by all financial instruments and pricing algorithms respectively.

This double class structure is interesting as it permits to separate the objectivable description of the instrument from the subjective pricing features such as possible dynamics or probability distributions.

Through inheritance we can account for the complexity of the multiple financial instruments and pricing engines while preserving some common interfacing features that will be interesting to design neatly some applications.

The basic structure built from these premises can schematized as follows:

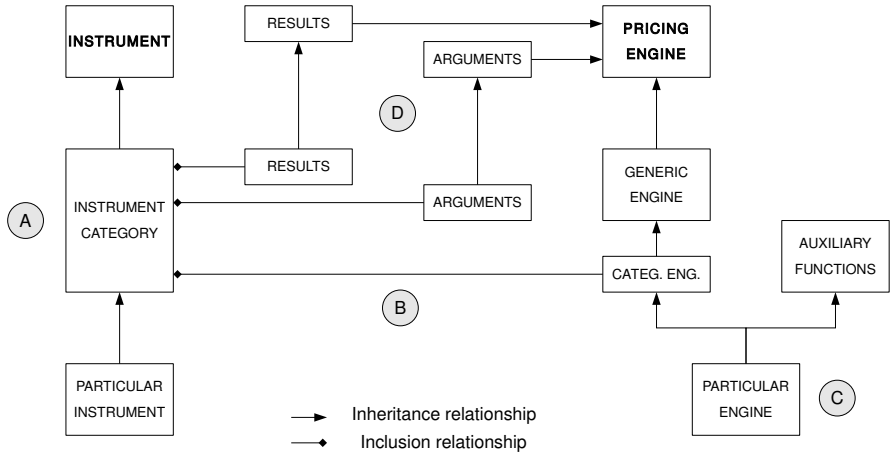

Figure 4.1: QuantLib structure diagram.

(A) Specific instruments are defined as inheriting from the basic instrument class, often with multiple layers representing wider instrument categories;

(B) Pricing engines will only make sense for some given instrument categories. A Category Engine, inheriting from Pricing Engine is embedded in each Instrument Category class;

(C) To facilitate the reuse of code, sometimes the pricing functions are coded in a separate class and the role of the actual pricing engine will be to combine these auxiliary functions into a pricing algorithm;

(D) Parameter passing: each engine will take two templates (arguments and results) that will specify the format of the interactions with the instrument. The classes that will

be passed as templates are thus be embedded in the instrument description while inheriting from the base pricing engine to acquire interaction functionalities.

4.2.2 From the basic class structure to program design

When building applications, one should think in terms of the previously reviewed scheme. As seen, QuantLib's structure can be thought of as mainly two classes, instrument and pricing engine, that interact. Since it is desirable to structure the design process in some way, we propose here to think in terms of these two main entities and perform a class requirement analysis for each of them and then finally proceed to the implementation of the required classes.

Once we have all the classes available, we focus on using the code in an application. Typically, QuantLib programs will have the same basic underlying structure that is schematized hereafter:

```
#include<QuantLib>

int main ()
{
  declare context variables

  declare instrument parameters
  declare instruments

  declare pricing parameters
  declare pricing engines

  link instruments to pricing engines

  evaluate instruments

  additional features (eg. portfolio rebalancing for delta hedging)
}
```

With this design, QuantLib takes full advantage of Object Oriented Programming: we start from the abstraction of the main theoretical concepts of the field and translating them into the main classes. Then, through inheritance, we account for the complexity of the subject while maintaining common interfaces, the polymorphism idea. Besides, the separation of the pricing algorithms and their auxiliary functions is a perfect example of encapsulation where changes can be made on specific parts of the program without altering what is built upon the modified modules.

A number of design patterns are also used to integrate more features such as result caching that introduces important time saving by storing results and recomputing them only when one of the variables underlying in the pricing process has been modified. These are discussed in more detail in Appendix C.

Up to this point, we have provided a very general outlook of the library's structure and use. The choice has been made to keep the presentation simple as a more detailed analysis could hide the forest behind the trees and still be incomplete. The reader is invited to find the details in the code itself and through the examples provided in the next sections.

Finally, it is interesting to have a broad view on what has been already implemented as this is the main strength of open source programming, the possibility of building up from different contributions. This general view is sought in Appendix C, where we provide a guided tour through of the directories that form QuantLib.

As we will see, often the most difficult choices in designing a new program will be on the way that the existing contributions are put together to achieve new functionalities. For this, it is critical to have a clear understanding of the underlying structure to be able to identify whether some code can be re-used or new features need to be introduced.

4.3 Developing in QuantLib

In this section we focus on the details of a particular implementation to see how all the structure translates into real applications. Besides, the example will also be useful to understand the implementation of the uncertain volatility model that is provided in the next chapter.

The analysis is presented as a reverse engineering analysis. The main reason for this is that when dealing with open source code with multiple contributions, it is essential to understand what has been done before to build on top of it. This understanding process may sometimes be very complicated, especially if there is no documentation and we are faced to the raw code, which is the case here. The proposed methodology consists the following steps: first, elaborate a class diagram (standard modeling languages are useful in that they provide a standardized methodology and the result is easily understandable for others); after that, identify the basic structure of the library, that is, in our case, the division between instrument and pricing engine; each part can then be separately analyzed to have

a complete understanding of the classes involved, their inheritance structures and the way they may interact. After performing this analysis, one should be able to understand and modify the original code and to build applications that use that code.

4.3.1 Class diagram

The class diagram presented here has been built according to the Unified Modelling Language (UML), a general-purpose modelling language widely used for the specification and design of object oriented programming applications. The official specification of the language can be found at [UML]. The diagram has been elaborated with MetaUML. The advantage of using such a language is that it helps to properly structure the diagram and the documentation of the language itself constitutes a usable tutorial of UML for our purpose. The project can be found in [Met]. It has to be pointed out that some applications exist that automatically crop the code into a diagram although none of them has been used for the project.

4.3.2 Instrument

The first step is to focus on the description of the instrument. As seen in the previous section, the class describing the European vanilla call should stem from the basic instrument class. Instead of a direct derivation, it is interesting to implement some intermediate class layers that can be reused for other instruments. The chain of inheritances for that particular case can be easily retrieved from the diagram. We provide here the motivation behind each intermediate level.

- First, the Instrument class derives from a *Lazy Object* class, whose only purpose is to permit results caching, a feature more extensively discussed in appendix C.
- The *Instrument* class provides the generic attributes and methods common to all financial instruments. The attributes judged to be general enough to be included in this base class are some related to NPV (such as the value itself or the relative error when applicable), expiration of the instrument, and finally a pointer to the pricing engine that should be used at a given moment to compute the NPV. As for the methods, we basically find those necessary to access and manipulate the mentioned data structures. Most of the methods are left virtual, which means that they

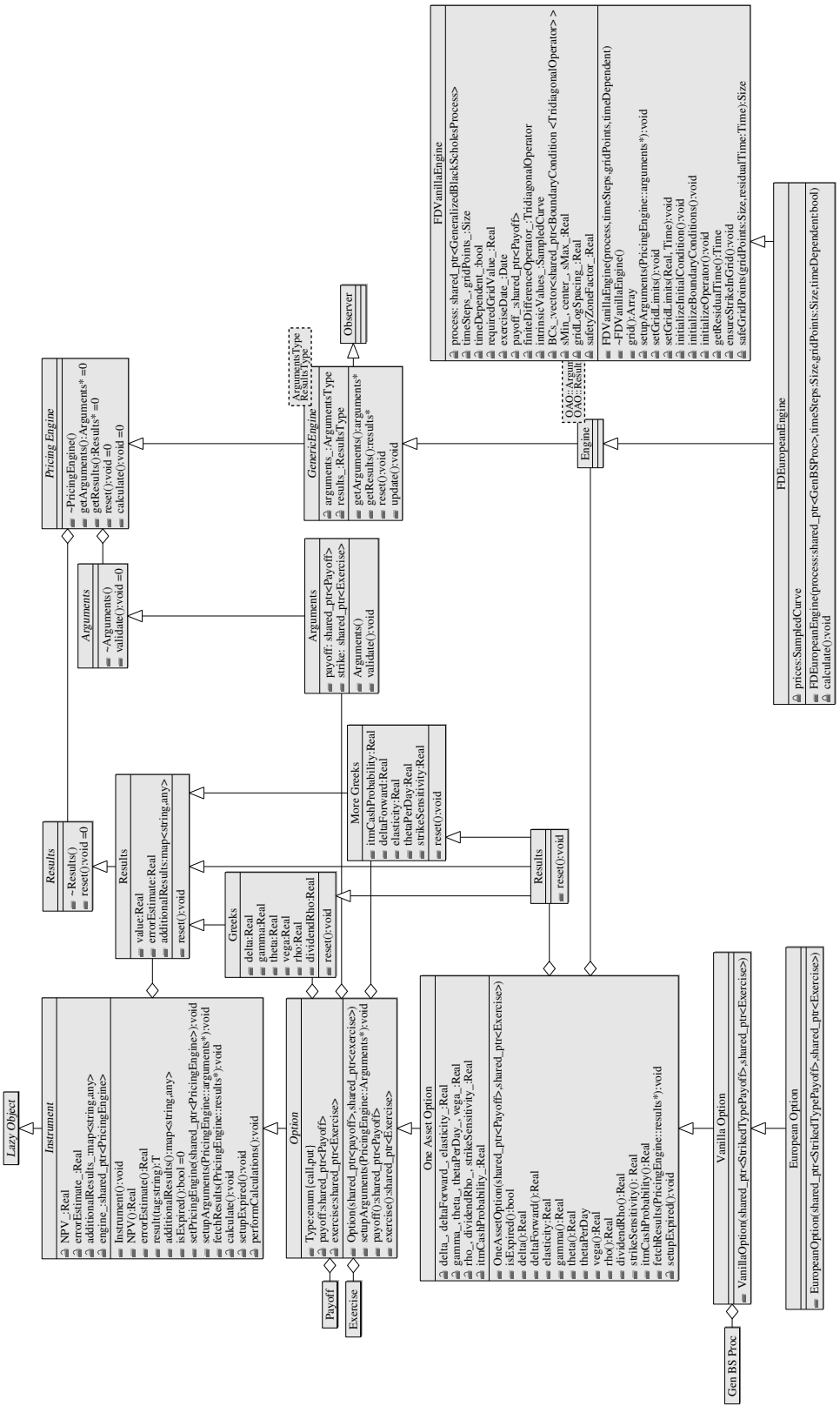

Figure 4.2: Class structure for European option pricing

support reimplementations in derived classes, to leave freedom for further specification tailored to each instrument. More than this, some are pure virtual, so that no implementation is provided in the base class. This means in particular that this base class is abstract and we will not be able to instantiate it directly, that is, no object of type Instrument can be declared in a program.

- In the following layer, the *Option* class are introduced the concepts of payoff and exercise that are common to all options. As these will characterize an option, the embedded Arguments class can already be implemented at this level. As for the Results, the Greeks are defined (standard and non-standard). Some of these measures may however vary if the option is written on several underlyings. This justifies the existence of the next layer.

- The *One Asset Option* restricts the range of possibilities for the option and makes it possible to definitely settle an embedded Results class (derived from the previous intermediary classes). Since both the Arguments and Results classes have been satisfactorily defined to cover the needs for any One Asset Option, we can define at this step a base pricing engine for One Asset Options. This pricing engine will take as template parameters the Arguments and Results classes and will be embedded in the Option class.

- The last intermediate layer reflects a widely spread financial term: *Vanilla Options* are in the financial jargon the most traded options that do not include exotic conditions such as barriers or other structuring features. The authors of the library consider vanilla options those that have a striked type payoff, that is, whose payoff is determined with respect to a given strike. A more controversial choice is the one to include the Generalized Black Scholes Process as an embedded class in the Vanilla Option class. This choice is not coherent with the general philosophy of separating the objectivable from the subjective evaluation part, therefore, we expect this to be revised in future versions of the library.

- Finally, the *European Option* class is derived from Vanilla Option so as to differentiate it from American Options or Bermudan Options. It is not necessary from a design point of view to implement call and put classes as the valuation algorithms will apply generally to both and any call or put. It is this class that we will instantiate in the application when declaring for instance a European Vanilla Call.

Although some of the intermediate class choices may appear arbitrary, they actually follow quite naturally if one thinks about it in terms of first specifying the arguments and results classes to be able to define the base engine that takes these as template parameters and then finally refining to capture the complexity of the financial instruments in the market. This class structure allows easy extensions or ramifications to elegantly integrate similar instruments in the framework.

4.3.3 Pricing engine

Once the instrument has been defined objectively, we focus on the pricing algorithm. One same instrument can generally be evaluated with different algorithms. In the example, the European vanilla call could be evaluated either by using the analytical Black Scholes formula, or by solving the pricing PDE or by simulating random realizations of the underlying, etc. Each of these should define a different pricing engine. With the QuantLib structure, we will be able to dynamically link our European vanilla call option object to a realization of any of these pricing engines.

We begin by analyzing the class breakdown as we did in the previous subsection:

- We start from the base *Pricing Engine* class that provides the means to interact with the Arguments and Results classes and to calculate these results with some given arguments. It is to be noted that both the Arguments and Results base classes are embedded in the Pricing Engine class so as to facilitate the interaction.

- The following layer is the *Generic Engine*, where the main difference with respect to the base Pricing Engine class is the template parameters. The generic engine class establishes the structure that every pricing engine will have while keeping it general enough to adapt to any financial instrument.

- The adaptation is done in the following step: the pricing *Engine* specific to the instrument category is derived from the *Generic Engine*. This construction translates the natural limitation of pricing engine's scope: we will not be able to use the same pricing engine to price different types of instruments such as, for instance, bonds or swaps. The idea is to include the pricing engine in the instrument category class to structurally capture this restriction.

- Finally, the *FD European Engine*, in this case, implements the actual pricing algorithm. Usually, this is done at a very high and readable level so another class containing the implementation of the auxiliary functions is needed. In the example, this will be the *FD Vanilla Engine* class. The benefit from this structure is double: on one hand, it allows encapsulation as the implementation of the functions may vary without having to alter anything that is built upon it and, on the other hand, these functions may be re-used for instance in the context of an American FD Engine.

We now focus on the pricing algorithm itself. We begin by recalling that option pricing theory can be formulated in terms of partial differential equations. The Black-Scholes PDE comes when forcing the derivative price process to be driftless. The resulting equation gives the dynamics of the payoff curve with respect to the underlying. When evaluating a European call option, we know the function that links the final payoff to the price of the underlying. This provides a boundary condition. Then, we make this curve evolve backwards in time with the Black-Scholes dynamics, ie we solve the PDE, and the curve obtained for time zero will determine the current price of the option according to the price level of the underlying. Since we restrict ourselves to a numerical solution of the PDE, we turn to finite differences to account for the curve's evolution over time.

This background will now allow us to understand the actual code implemented in the pricing engine:

```
setupArguments(&arguments_);
setGridLimits();
initializeInitialCondition();
initializeOperator();
initializeBoundaryConditions();

StandardFiniteDifferenceModel model(finiteDifferenceOperator_, BCs_);

prices_ = intrinsicValues_;

model.rollback(prices_.values(), getResidualTime(), 0, timeSteps_);

results_.value = prices_.valueAtCenter();
```

The code has been directly extracted from the QuantLib file FDEuropeanEngine. Each function called in this process is implemented in superior layers of the inheritance structure.

In the rest of this subsection, we will analyze each of these steps to understand what they do and how they do it, that is, to express the algorithm in terms of the classes that are used in the implementation. We note once again that the design process should work in the opposite way: that is, by choosing an algorithm first, then analyzing the requirements in terms of classes and then working through the implementation. However, we insist on the reverse engineering approach that is suitable to analyze the existing code, an essential task in open-source development.

1. The first step in the algorithm, after retrieving the parameters, is to initialize the grid over which the discretization of the pricing equation will be done. The grid can be thought of as an Array of values of the underlying. It has to be noted that for efficiency purposes a logarithmic grid is preferred since most relationships become linear by doing this simple assumption. We also remark that, since the process is considered to be driftless under the pricing measure, the same grid can be used throughout all the process; the grid is centered on the current price of the underlying as it is the most relevant value for pricing. One should also ensure that the strike of the option is contained in the grid, a function that is performed by an auxiliary method. The particular parameters for the specification of the grid depend on the algorithm. Generally, for Black Scholes equations, the volatility will be taken into account in the determination of the grid.

2. The second step is to evaluate the initial condition. This initial condition is the payoff at maturity that we will make evolve through time to retrieve the present value of the instrument. In fact, in the discretized finite difference approach, we only need the values of the initial condition over the grid. That is why Sampled Curves that combine two Arrays, one for the grid, one for the values, are used all across the algorithm;

3. The following step is to initialize the operator that encapsulates the relationship (time derivative) between W_t^i and W_{t+1}, where the subindex denotes the time variable and the superindex the grid position. The Operator, is merely the translation of the pricing equation, in that case the PDE BSM Partial Differential Equation of Black Scholes and Merton. Such operators are allowed to be time dependent through the use of a Time Setter that changes the coefficients according to a time variable. Finally, we observe that, in this case, the operator turns out to be expressable as a tridiagonal matrix since W_t^i is only dependent on W_{t+1}^{i-1}, W_{t+1}^i and W_{t+1}^{i+1}. Initializ-

ing the operator will amount to defining the weights of these three factors in the tridiagonal operator.

4. Next, the Boundary Conditions are initialized. Boundary conditions are needed because for extreme values we will not be able to use the same tridiagonal operator: for W_t^0 we lack W_{t+1}^{-1} and, similarly, for W_t^N we don't have W_t^{N+1}, where N is the grid size. The boundary condition used for this algorithm is the Neumann BC that sets a value for the normal derivative.

5. Then, the operator and the boundary conditions can be used to discount the initial condition. This is done through a Finite Difference Model. The function of such a model is to set the steps, adapt the operator and the boundary conditions to the the step and iteratively apply a given scheme to obtain the discounted curve, eventually applying some extra conditions at each step (this feature is basic for American options pricing). The scheme chosen for this particular engine is the Crank-Nicholson scheme, which is a particular case of a Mixed Scheme. A mixed model performs the roll back by subdividing the step length and using the operator explicitly in the first part and implicitly in the second part; such manipulations are performed to ensure the robustness and convergence of the scheme.

6. Finally, the net present value of the derivative is computed by evaluating the sampled curve at its center.

4.3.4 Building an application

We finally turn to the part of building an application that uses the described pricing engine to evaluate the instrument. We focus for the example on an application that will replicate a European option. The program will compare the performance of a self-financed portfolio, dynamically Δ-adjusted with respect to the actual performance of option prices.

Following the structure described in the previous section, we first define some context variables such as dates and calendars to be used.

Next step is to define the instruments. For the example, we focus on a European call option. Given the model presented above, to fully determine an option we need to specify its payoff and maturity. For the payoff, it will depend on whether it is a call or a put and on its strike. For the maturity, it will suffice to know the maturity date and the type, ie.

European, American, etc. Once payoff and maturity have been declared, the instrument can be instantiated.

Up to this moment, we have defined the objective characteristics of the instrument. We now focus on the declaration of the pricing engine. Again, we follow the same logic: declaration of the parameters and then the declaration of the engine itself based on those parameters. However, here there is an intermediate step: the valuation may depend on market inputs that will make our parameters change. Namely, as time goes by, we will need to modify our assumptions on the evaluation date and on the underlying's price. We want all these changes to be accounted for in each new valuation. To do so, we take advantage of the observer pattern, discussed more in detail in the appendix C. Basically, the idea is that through the handles structures, we keep track on the modifications and make the changes cascade to all appropriate levels.

Once all the parameters have been initialized and stored through handles, we can finally declare the pricing engine.

The following step is to associate the pricing engine to the instrument to indicate that all calculations should be performed with the selected pricing engine.

The rest of the program will be more dependent on what specific task is aimed for. For the case of the example, we read a data set containing the prices of the underlying at specific dates. Namely, the input consists of standard market data, that is, date, open, high, low, close and volume. The objective here is to keep track of a Δ-hedging strategy. So the first step is to build a portfolio containing both the underlying and cash to replicate the option. The initial amount to be allocated to this portfolio is the theoretical value of the option. As time passes, we rebalance the portfolio, keeping Δ shares per option, where Δ is the delta of the option with the current price. We also account for the growth of cash (as an interest charge or payment).

This simple program will allow us to replicate options with self-financing portfolios and keep track of the replication error through time.

4.3.5 Conclusion

We have reviewed in this section a particular pricing engine from a reverse engineering approach. This allows us to understand how the different classes interact and with this

vision we can elaborate programs such as the one presented. The following step is to be able to create new classes that properly interact with the existing ones. This will allow us to extend the library's functionalities and develop new applications.

Chapter 5

Implementing uncertain volatility in QuantLib

5.1 Overview

This chapter puts together the theoretical review of chapter 3 and the analysis of QuantLib in chapter 4 to produce an implementation of the Uncertain Volatility Model. The presentation is organized to reflect the whole process: first, the basic design choices and trade-offs are discussed; then, a class requirement analysis is carried out to determine some guides for the implementation; according to these specifications, classes are coded; and finally some applications are built, first to test the implementation and then to provide support to traders, which is the ultimate goal of a quantitative finance project.

We start by by briefly recalling the main points of the two previous chapters:

On one hand, we have a model that provides a portfolio-dependent option valuation by solving a Black-Scholes derived differential equation that dynamically choses the volatility within a pre-specified range so as to maximize (minimize) the overall portfolio value.

On the other hand, we have a library that builds on a dichotomy of instrument and pricing engines. Pricing engines take inputs such as the dynamics of the underlying, and produce an evaluation, which in the case of Black-Scholes models will amount to the computation of the expected price. This is achieved either analytically, where a closed form solution is available, or numerically (Montecarlo, finite difference or trees).

From this, we want to build a QuantLib application to price a portfolio of simple options (European type for the example) included in a portfolio of such securities in an uncertain volatility scenario.

We proceed following the guidelines established in the previous chapter, that is, by firstly establishing the class requirements in terms of instrument and pricing engine and then implementing the classes.

5.2 Class requirements

In the identification of the class needs, we always face a trade-off between integration and flexibility: on one hand, the more we use existing code, the better the integration in the library will be and this will show clear advantages in its use and maintenance. On the other hand, implementations from scratch avoid the non-elegant adjustments needed to fit the new functionalities in the existing code. For this case, we will tend to favor integrated code to elegant solutions as a correct integration enables the user to take advantage of the whole potential of the library.

To examine the class requirements, we base our analysis on the example discussed in the previous section. From this example, we look at the concepts introduced by the uncertain volatility model and we evaluate the changes needed to integrate those new concetps. Again, for clarity purposes, the analysis separates the instrument and the pricing engine.

On the instrument side, a new concept comes into the model: the portfolio. In the uncertain volatility model, all the valuations are portfolio dependent. It is reasonable so to define this new concept with a class of its own. It is natural to make this class derive from the instrument class as the portfolio can be seen as an aggregate instrument such as the coded "composite instrument".

On the pricing side, there is one major change: the introduction of the Uncertain Volatility. First of all, this new volatility neatly appears as a new class. Here, the question of derivation is trickier and we find an illustration of the described trade-off between adaptation and elegance of the solution: the natural choice in a QuantLib perspective should be to derive this new volatility from the volatility term structure class. However, term structures are designed to return at any particular time a value for the volatility, a feature that does not make sense in the uncertain volatility context, since all that is known at a given time is

a range for the volatility and no particular values. Thus, we find the problem of having to adjust the library for a function that it is not meant to support. However, if we choose not to use this inheritance structure, our new volatility cannot be used as input for sotchastic processes which would then in turn have to be redifined to be adapted. We opt thus for making the necessary adjustments to make things work under the QL natural inheritance structure.

The next natural step is to define a stochastic process that we shall call Black Scholes Barenblatt process and that will take as input an uncertain volatility.

If we track forward the changes triggered by this new volatility, we find that the partial differential equation is changed. We no longer have to consider a Black Scholes Merton (BSM) PDE but rather a Black Scholes Barenblatt (BSB) PDE as the one described in the uncertain volatility paper. Basically, this class has to account for the new volatility in the specification of the coefficients of the operator. To do it, it will also need to keep track of the payoff function as the choice of the volatility will depend on the curvature of that function.

The operator itself also is changed. We need the operator to change the coefficients according to the curvature of the payoff function. This will be done in through a BSB Operator class.

The finite difference model used for the roll-back process will also need to be modified in that it only allows to discount one sampled curve while we need to discount both the sampled curve of the individual payoff and the one of the portfolio payoff as the latter determines the volatility at each step. Since the roll-back must be done simultaneously for the two curves, separate calls of the discounting process will not solve the problem. The class needed then is a Modified Finite Difference Model.

Finally, we are creating a new pricing engine that should have a class of its own, the FD Uncertain European Engine. We also note that most of the auxiliary functions contained in the FDVanillaEngine class need to be recoded after the proposed class changes; this is done in the FD Uncertain Engine.

For the rest, the structure of the algortihm is still valid and no additional changes need to be made.

We can sum up our class requirements in the following list:

- Portfolio instrument class
- Uncertain volatility class
- Black Scholes Barenblatt process
- Black Scholes Barenblatt PDE
- Black Scholes Barenblatt Operator
- Modified Finite Difference Model
- Finite Difference Uncertain Volatility Pricing Engine
- Finite Difference Uncertain Volatility Pricing Engine for European Options

5.3 Implementation

This section is devoted to review the implementation details of the classes stated in the previous section.

Portfolio instrument class

The implementation of this class is inspired on the composite instrument class that can be found in QuantLib. However, a reimplementation has been preferred because of undesirable features observed in the composite instrument implementation such as the non consolidation of registries for a same security (that is, if one buys a call and the sells it, the code would generate two entries: a positive one and then a negative instead of cancelling both).

As for the implementation itself, the chosen data structure has been the list to efficiently handle changes in the portfolio composition. The type of security has been restricted to one asset options. In effect, for our case, only portfolios based on the same underlying make sense. To avoid misunderstandings, the class has been named One Asset Option Portfolio (or OAOPortfolio). For the methods, they are basically the same as the ones that can be found in the composite instrument with the appropriate adjustments to correct the detected flaws.

One last thing has to be noted for this class: normally, we would like to register this portfolio to all of the options that are included in it. However, when we define the portfolio-based engine, this can produce a circular loop as changes in composition of the portfolio would trigger a change in the pricing engine which would in turn update the instruments and the effect would be a second notification to the portfolio and so on. We prefer thus to make the updates manually in case an instrument integrating the portfolio suffers a change (since this possibility is rather unrealistic, we prefer to make the adjustment here rather than in the portfolio-engine relationship.

Uncertain volatility class

The implementation of this class is mostly based on the Black constant volatility class (we assume that the interval bounds for the volatility are constant over time; taking another base class, one could easily implement time-varying bounds).

There are two main changes with respect to the original implementation: first, the blackVol function, responsible for returning the value of the volatility at a given time, is disabled with the `QL_FAIL` instruction. The reason is purely theoretical as the function does no longer make sense in the uncertain volatility framework. To avoid a misuse of the function or un-noticed errors, it is preferrable to disable the command; if any ohter program calls this function, the application will be terminated with the correpsonding error message.

The second change is the addition of a blackVol2 function to replace the disabled blackVol. This new function returns either the higher or the lower bound for the volatility depending on an input parameter. Both σ_{min} and σ_{max} are stored through Handles, a data structure that enables caching mechanisms to trigger as discussed in the Appendix C.

Black Scholes Barenblatt process

The Black Scholes Barenblatt process class is built as a derived class of the one-dimensional stochastic process class. Its main distinctive feature when compared to the Black Scholes process is that the volatility term structure is taken to be of the uncertain type defined above. In fact, the implementation here is a little tricky because of the C++ derivation rules that do not allow us in this case to simply redefine the type of the volatility data

structure. Upon call of the constructor, the uncertain volatility parameter is taken to be a more general Black volatility. This has the effect of blocking the uncertain volatility specific functions, such as the ones that retrieve the upper and lower bounds of the parameter interval at a given moment. What we do to overcome this problem is to define a pointer through which we will access the parameter. This pointer identifies the volatility term structure as of uncertain type and hence we can freely operate. Some type casts need to be done to fully comply with the compiler requirements.

Black Scholes Barenblatt PDE

The PDE class implementation is based on the BSM PDE class. The main change is the adaptation of the generate operator function to the uncertain volatility.

We remark here the pointer parameter that is passed for the generate operator method. This reference is essential in the new model as it points to the portfolio payoff, which determines which of the two bonds has to be used at every time for the coefficients of the operator.

Black Scholes Barenblatt Operator

The class is again inspired on the BSM Operator. The main changes required come from the time setter tat requires an additional parameter, that is the reference to the portfolio payoff. This simple change is made only to provide the PDE class with the appropriate parameters to generate the operators.

Modified Finite Difference Model

This class is programmed to take into account the parallel discounting process for both the reference portfolio payoff and the evaluated instrument payoff. The code consists of a slight variation of the original Finite Difference Model class, where instead of applying the discounting process to one vector, it is done simultaneously to the two vectors.

Finite Difference Uncertain Volatility Pricing Engine

As pointed previously, the class FDVanillaEngine is merely a compilation of the pricing functions shared by all finite difference pricing engines. In our case, however, because of the modifications carried out in the other classes, some changes are needed.

Namely, in the definition of the grid, the volatility is used. Since we no longer have a single value for the volatility, we must specify here which of the values should be used (we take the same approach as the authors of the paper and always use the highest volatility to set the grid). We also need to modify the call to the operator initialization since it takes the additional portfolio reference parameter.

Finite Difference Uncertain Volatility Pricing Engine for European Options

This class contains almost the same code as the standard FDEuropeanEngine with the only modifications to adjust for the portfolio references and the calls to the modified classes and functions.

5.4 Applications

In this section we will review two applications of the uncertain volatility model that will use the implemented classes discussed above.

5.4.1 Portfolio pricing

The first program is a straightforward application of the model, that is, pricing a portfolio of European options. In particular, we present a program that evaluates a bull call spread as the one presented in the original paper. The aim here is double: first, we can illustrate the use of the new functions that we have built and the second is to test the performance of the algorithm by comparing the obtained results with those obtained by the authors.

The instrument priced is a bull call spread, that is, a long call combined with a short call. Here, the long call has a strike of 90 and the short call a strike of 100. All options are taken to be European and mature in a six month horizon. The additional parameters for the

valuation, that is, those that describe the dynamics of the underlying, are set as follows: interest rate $r = 5\%$; volatility comprised between $\sigma_{\min} = 10\%$ and $\sigma_{\max} = 40\%$.

The code presented in Appendix F is organized as described in the previous chapter: first, the context variables such as dates or calendar to be used are defined. Next, we focus on the instrument definition, setting all the parameters. Most of them are defined through handles in order to make the caching mechanism account for any changes in their value. Then, the pricing engines are defined. Again, the parameters are defined as handles to allow for online changes and recalculations. Finally, the application, is programmed.

In this case, the application consists of a loop that goes over several prices of the underlying and evaluates the instrument. It compares the price obtained with the uncertain volatility model with the sum of the highest price approach for each of the instruments (that is, considering the maximum volatility for the long call and the low volatility for the short call).

The program is run two times, the first to determine the worst-case price. Then, a second run is performed to obtain a best price. To obtain the best price, one needs only to swap $\sigma_{\max}$ and $\sigma_{\min}$ as can be seen from the algorithm presented in chapter 3 (the algorithm is symmetric (except for the grid construction, a detail that is already accounted for in the pricing engine code).

The results obtained are plotted in figure 5.1:

We can observe that the enveloppe obtained by the uncertain volatility model narrows the one computed by separately considering each instrument. Moreover, the graph obtained is identical to the one that can be found in the original paper (see chapter 3), which means that the implementation stands the basic test.

5.4.2 Static hedging

The second application is an implementation of the lambda uncertain volatility model described by Avellaneda and Paras in [APEK+96].

The example considered here is the static hedging of a long European call option struck at the money, at 100, with 6 months maturity. The hedging instrument considered here is a liquidly traded European option on the same underlying, with strike at 110 and same

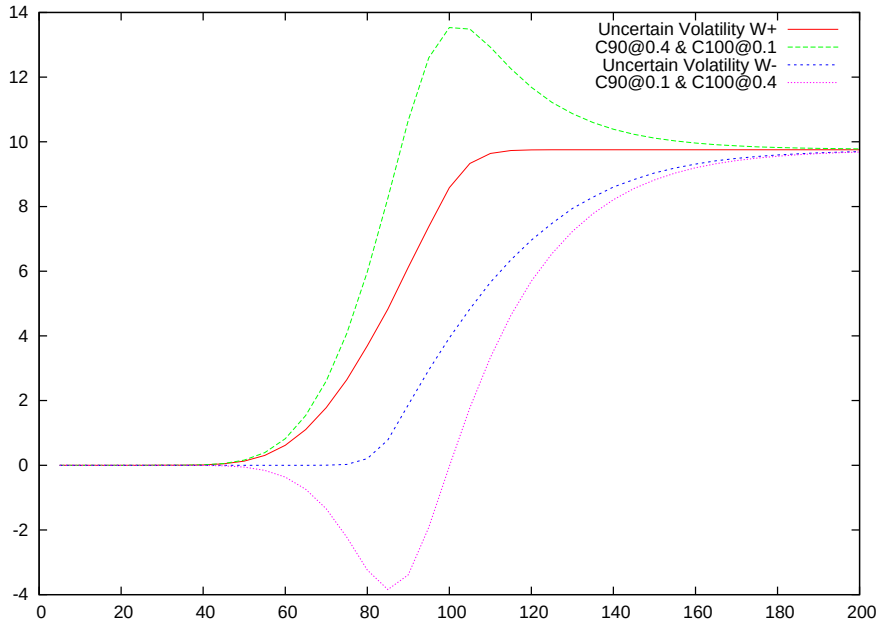

Figure 5.1: Bull call spread valuation results

maturity. The problem is to decide the optimal amount of the hedging instrument to be included in the portfolio to hedge the exposure. The parameters considered for the valuation are an interest rate of $r = 7\%$, and a volatility comprised between $\sigma_{\min} = 2\%$ and $\sigma_{\max} = 32\%$. Finally, the hedging instrument is considered to trade at 5.00. The program is run with 500 iterations.

The code presented in Appendix G is structured to carry out this optimization with a very simple optimization algorithm (gradient). The commented part of the code produces the results to visualize the effect of the changes in lambda on the overall portfolio hedging value.

The results of the commented part of the code are presented in the following table and plotted in figure 5.2.

The first column of the table represents the value of the residual liability. The next two columns represent the amount of each instrument in the portfolio; then, the prices of each of these instruments according to the uncertain voilatility model; and finally the cost of taking a long position on the hedging assets and the difference between the residual liability and this cost.

Residual liability	Q(call1)	λ = Q(call2)	Call1 value	Call2 value	Cost	Overall liability
-10.587	-1	0	10.587	6.42711	0	-10.587
-10.2735	-1	0.05	10.5847	6.22397	-0.25	-10.5235
-9.96496	-1	0.1	10.5781	6.13137	-0.5	-10.465
-9.66098	-1	0.15	10.5668	6.03879	-0.75	-10.411
-9.36198	-1	0.2	10.5478	5.92892	-1	-10.362
-9.06877	-1	0.25	10.5191	5.80147	-1.25	-10.3188
-8.78255	-1	0.3	10.479	5.65494	-1.5	-10.2826
-8.50512	-1	0.35	10.414	5.45404	-1.75	-10.2551
-8.23893	-1	0.4	10.3204	5.20365	-2	-10.2389
-7.98764	-1	0.45	10.176	4.86307	-2.25	-10.2376
-7.75677	-1	0.5	9.94794	4.38236	-2.5	-10.2568
-7.55535	-1	0.55	9.57437	3.67095	-2.75	-10.3053
-7.40007	-1	0.6	8.92509	2.54171	-3	-10.4001
-7.30399	-1	0.65	8.44677	1.75813	-3.25	-10.554
-7.21608	-1	0.7	8.44677	1.75813	-3.5	-10.7161
-7.15549	-1	0.75	8.43731	1.7091	-3.75	-10.9055
-7.07003	-1	0.8	8.43731	1.7091	-4	-11.07
-6.98458	-1	0.85	8.43731	1.7091	-4.25	-11.2346
-6.89912	-1	0.9	8.43731	1.7091	-4.5	-11.3991
-6.87284	-1	0.95	8.32954	1.53337	-4.75	-11.6228
-6.87956	-1	1	8.19881	1.31925	-5	-11.8796
-6.86746	-1	1.05	8.11511	1.18824	-5.25	-12.1175
-6.85225	-1	1.1	8.05015	1.08899	-5.5	-12.3523
-6.83929	-1	1.15	7.99132	1.00177	-5.75	-12.5893
-6.82275	-1	1.2	7.94646	0.93643	-6	-12.8227

Table 5.1: Results of the Lagrangian Uncertain Volatility example

We note that the residual liability consists at first only of the sold call. We focus here on worst case valuations and hence it is evaluated at the highest volatility. As we add the hedging call to our portfolio, the value of the residual liability decreases as a result of the static hedging effects. The cost however, which is obtained by multiplying the quantity of the hedging option with its market price, increases as we overweight this instrument in the portfolio. The difference between the residual liability and the cost is the total liability. This figure reaches an optimum for $\lambda \simeq 0.435$, which corresponds to a price for the hedging call of 5.00, that is, equal to the market price, which confirms the no-arbitrage remarks of chapter 3.

We conclude this section with some remarks on the computational time. Times quoted relate to the following configuration: Ubuntu OS run virtually on Windows Vista through

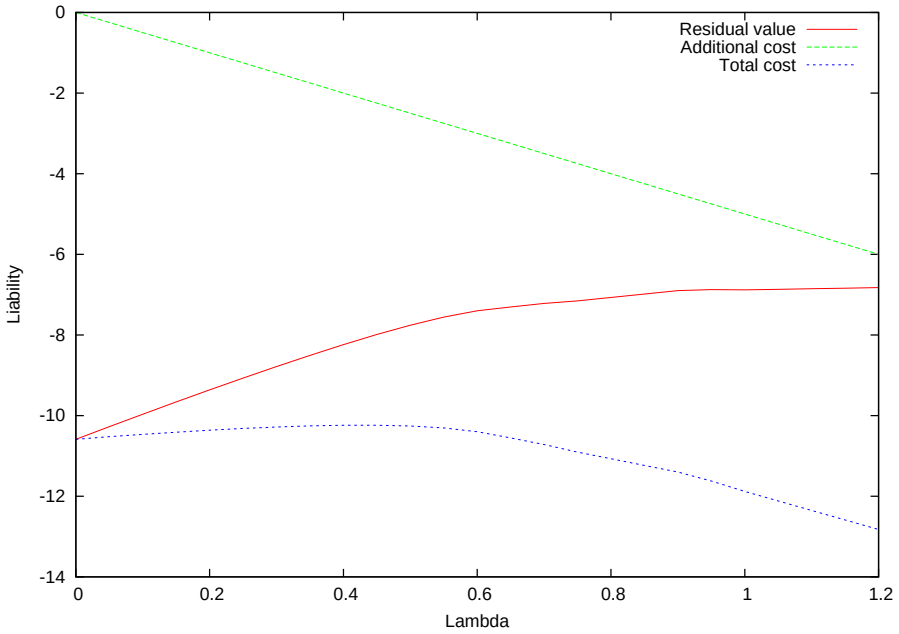

Figure 5.2: Results of the Lagrangian Uncertain Volatility example

Sun VirtualBox. The base CPU is Intel Core2 Duo at 2GHz and the virtual machine takes 512MB of RAM memory. In this case, the valuation of a 2 European option portfolio with 500 iterations takes aproximately 1s while the separate evaluation of both instruments through QuantLib finite difference schemes (also with 500 iterations) takes aproximately 10ms. This difference grows at a more than linear rate as the number of securities increase. The overhead can be partly explained by the following remarks: first of all, there is a first step of payoff agregation that we do not need in the standard model; then, the main overhead comes from the finite difference operator, which has to be recomputed at each iteration in the case of the uncertain volatility model. However, this would only explain a linear factor. A finer analysis permits to track down the most important costs to memory access which suggests that there is room for optimization in this aspect but this is beyond the scope of this project.

Chapter 6

Conclusions

In this project we have developed a tool for quantitative finance, starting from the analysis of a model to its implementation and integration in an existing framework.

From the theoretical point of view, we have presented a model that builds on the Black Scholes approach and integrates part of the parameter specification risk by considering the parameter to be in a pre-specified interval. The model shows that the parameter specification risk can be limited through diversification. The models provide us with the basic insights of the criteria of the diversification: changing convexity of payoffs should be sought to take advantage of the non-linearity features of the pricing equations. The model proves to have realistic applications in static hedging where the existence of other liquidly traded instruments can be used to narrow the price envelope and produce a competitive spread.

On the technical side, we have shown how to implement and integrate uncertain volatility related algorithms in QuantLib. On this point, the bulk of the work remains out of this report. First, there is a whole work of understanding an undocumented code that requires a methodology and a deep knowledge of the subject. The introductory chapter that may appear somehow disconnected is essential to the understanding of the library. Besides, despite the few references in the report, the modeling with UML has also proved to be an essential part in the study of the library.

As the project remains highly focused on QuantLib it is also interesting to provide a short review of this tool. QuantLib has opened a very interesting path in the future of quanti-

tative finance, both in the creation of standards and in the elimination of obsolete competitive barriers in this field. However, despite of its many impressive features, QuantLib may fail to accomplish its objectives and remain a tool restricted to a narrow community as it has overlooked in its development the creation of a supporting documentation, a key feature for an open source solution. Although some efforts are being made in this direction, the role of the authors still seems to remain too focused on the development of non-fundamental extensions, neglecting the opening to a wider community.

In this project we have built tools to price and hedge financial derivatives in the presence of model risk. The paradox is that we dealt with model risk by using new models. It is because of this fundamental role of models in quantitative finance that it may be interesting to conclude with a remark on their use in this field. However difficult it might be to summarize the importance and the dangers of models in finance, it is important to always keep in mind that models are made to guide the thinking and not to establish absolute truths. Prices produced by models should always be looked at with a certain skepticism and the limits of any model should be put forward every time they are used. Some may blame quantitative finance for the current crisis; still, it remains only a tool, it is how it is used that will determine its effects. We would like to end by eagerly suggesting the reader to take a look at the Financial Modeler's Manifesto [DW] written by Wilmott and Derman that brilliantly and succinctly exposes these ideas.

Appendix A

C++

QuantLib is written in C++, an object-oriented programming language. In this appendix, we provide a short review of the basic concepts of object oriented programming as they are extensively used in the construction of QuantLib. We base this review on the work carried out by D. Armstrong [Arm06] and its presentation in [OOP]. We present hereafter an adapted version of the latter with QL-related examples.

Object oriented programming is a programming paradigm that is built on objects and their interactions. The basic features of an object oriented programming language are the following:

A **class** defines the abstract characteristics of a thing (object), including the thing's characteristics (its attributes, fields or properties) and the thing's behaviors or interactions (the things it can do, or methods, operations or features). One might say that a class is a blueprint or factory that describes the nature of something. For example, the class Option would consist of traits shared by all options, such as payoff and maturity (characteristics), and the "ability" to quote a price or to communicate whether it is expired or not (interactions). Classes provide modularity and structure in an object-oriented computer program. A class should typically be recognizable to a non-programmer familiar with the problem domain, meaning that the characteristics of the class should make sense in context. Also, the code for a class should be relatively self-contained (generally using encapsulation). Collectively, the properties and methods defined by a class are called members.

An **object** is a pattern (exemplar) of a class. The class of Option defines all possible options

by listing the characteristics and behaviors they can have; an over-the-counter European call option on the Goldman Sachs stock with a strike of $130 and expiring on May, 25th, 2009 is an object with its own particular characteristics.

One can have an **instance** of a class or a particular object. The instance is the actual object created at runtime. In programmer jargon, the object corresponding to the option described above is an instance of the Option class. The set of values of the attributes of a particular object is called its state. The object consists of state and the behaviour that's defined in the object's class.

Methods describe an object's abilities. In language, methods (sometimes referred to as "functions") are verbs. We want to make it possible for a particular option to quote its value, so we will implement an NPV() method. Within the program, using a method usually affects only one particular object; all Options will be able to quote their value, but at a given point we will only be interested in the value of a particular option.

Message passing is the process by which an object sends data to another object or asks the other object to invoke a method. Also known to some programming languages as interfacing. For example, a Portfolio object could ask every option in it to quote a value in order to retrieve the whole portfolio value.

Inheritance: "Subclasses" are more specialized versions of a class, which inherit attributes and behaviors from their parent classes, and can introduce their own. For example, the class Option might have sub-classes called Vanilla and Exotic. In this case, our European otpion would be an instance of the Vanilla subclass. Suppose the Option class defines a method called NPV() and a property called maturity. Each of its sub-classes (Vanilla, and Exotic) will inherit these members, meaning that the programmer only needs to write the code for them once. Each subclass can alter its inherited traits. For example, the Vanilla class might specify that the default expiration for a vanilla option is either European, American or Bermudan. Subclasses can also add new members. The Exotic subclass could add a method called sensitivity() to compute the sensitivity of hte option to additional specific parameters. Multiple inheritance is inheritance from more than one ancestor class, neither of these ancestors being an ancestor of the other.

Abstraction is simplifying complex reality by modelling classes appropriate to the problem, and working at the most appropriate level of inheritance for a given aspect of the problem. For example, the option that we were considering on Goldman Sachs stock may

be treated as an Option much of the time, or a Vanilla option when necessary to access Vanilla-specific attributes or behaviors, and as a Financial Instrument (perhaps the parent class of Option) when evaluating diversified portfolios. Abstraction is also achieved through Composition. For example, a class Option would be made up of a Maturity and a Payoff. To build the Option class, one does not need to know how the different components work internally, but only how to interface with them, i.e., send messages to them, receive messages from them, and perhaps make the different objects composing the class interact with each other.

Encapsulation conceals the functional details of a class from objects that send messages to it. For example, the Pricing Engine class has a calculate() method. The code for the calculate() method defines exactly how the value of a financial instrument is computed. When evaluating a portfolio, the portfolio does not need to know how every particular instrument is evaluated. Encapsulation is achieved by specifying which classes may use the members of an object. The result is that each object exposes to any class a certain interface - those members accessible to that class. The reason for encapsulation is to prevent clients of an interface from depending on those parts of the implementation that are likely to change in future, thereby allowing those changes to be made more easily, that is, without changes to clients. Members are often specified as public, protected or private, determining whether they are available to all classes, sub-classes or only the defining class.

Polymorphism allows the programmer to treat derived class members just like their parent class' members. More precisely, Polymorphism in object-oriented programming is the ability of objects belonging to different data types to respond to method calls of methods of the same name, each one according to an appropriate type-specific behavior. One method, or an operator such as +, -, or *, can be abstractly applied in many different situations. Overloading Polymorphism is the use of one method signature, or one operator such as "+", to perform several different functions depending on the implementation. The "+" operator, for example, may be used to perform integer addition, float addition, list concatenation, or string concatenation. Any two subclasses of Number, such as Integer and Double, are expected to add together properly in an Object Oriented Programing language. The language must therefore overload the addition operator, "+", to work this way. This helps improve code readability. How this is implemented varies from language to language, but most OOP languages support at least some level of overloading polymorphism.

Decoupling allows for the separation of object interactions from classes and inheritance into distinct layers of abstraction. A common use of decoupling is to polymorphically decouple the encapsulation, which is the practice of using reusable code to prevent discrete code modules from interacting with each other. However, in practice decoupling often involves trade-offs with regard to which patterns of change to favor.

Appendix B

Installing and using QuantLib

This appendix deals with the installation of QuantLib. Basically, the process can be divided in two parts: the installation of the Boost libraries and the installation of QuantLib itself. To have a better understanding of the process, we can distinguish between the raw code, that is, a set of instructions written in the programming language, and the compiled code, that would be the equivalent of our set of instructions, properly transalated in a language to be understood by our machine. When downloading both Boost and QuantLib, we will obtain the code. Then, some additional steps will be needed to compile and link the libraries so that they can be properly used by our programs.

We have opted for providing an explanation of every step rather than simply a list of commands that would soon become obsolete and highly dependent on the operating system used. However, we still provide an example based on an installation on an Ubuntu operating system. One can easily reproduce the base configuration of the example by installing a virtual machine on the current operating system and then running this new machine with an Ubuntu distribution.

Installing the Boost libraries

QuantLib uses some non-standard C++ features such as shared pointers or some types of error handling. To compile our QuantLib libraries we will need to have this features ready to be used. This means we need to install first the Boost libraries.

The Boost project consists in a highly reliable implementation of "existing practice". That is, some problems appear repeatedly in the develloping of programs. Every programmer can come to a solution of its own and use it to build its programs. However, some functions are so recurrent that they have given place to standard implementaitons so that they can be readily used by every programmer without wasting time in the re-implementaiton. Besides, standard functions are strictly reviewed and optimized so that the user can rely on the implementation as the most efficient for its own programs (except, of course, for some very specific problems). However, the scope of the standard library is limited and some widely used solutions lack a standard implementation. Here is where the Boost project comes in offering a collection of solid, peer-reviewed implementations of highly recurrent solutions. The project Boost is highly recognized in the industry and is somehow the antichamber for the standard library as a considerable number of Boost features are added in the standards as these are reviewed.

To install the Boost libraries we must first download the source files, ie the raw code, along with the compiling instructions. Generally, all this comes bundled in the distributions that can be obtained through the project's website: `www.boost.org`. For MS Windows users, the compiling instructions are a bit tricky and require previous configurations.

For Ubuntu users, it is advised to ensure that some components are already installed. This can be done with the following commands in a terminal:

```
1. apt-get install gcc
2. apt-get install g++
3. apt-get install python2.5 (or latest version)
4. apt-get install python2.5-dev (or latest version)
   (apt-get install libbz2-dev)
   (apt-get install zlib1g-dev)
   (apt-get install libicu36-dev)
```

Then, one should download the boost distribution from the project's website and unpack it:

```
1. sudo su (login as superuser to have manipulation privileges)
2. cd usr/local
3. tar --bzip2 -xf /home/nnn/Desktop/boost_x_xx_x.tar.bz2
```

```
    where nnn stands for the name of the user
    and x_xx_x the version of Boost
4. cd boost_x_xx_x
5. ./configure
6. make install
```

The execution of this last command may take a few minutes. After this, one should have in the `/usr/local/lib` directories the compiled libraries and in the `/usr/local/include` a folder named `boost-x_xx_x`. The last step to perform is to open this folder, copy the boot directory in it to the upper level (`/usr/local/include`) and erase the `boost-x_xx_x` folder.

After performing these steps, one should be able to use the Boost features.

Installing QuantLib

We then come to the installation of QuantLib itself. Again, we should download the distribution and unpack it in the `/usr/local` folder similarly to what we did with Boost. By doing so, we are downloading the source code plus the compiling instructions. For the rest, we will only need to execute the compiling instructions.

In the Ubuntu case, using again the privileged superuser mode in the terminal, we type the following comands:

```
1. cd /usr/local/QuantLib-x.x.x (where x stands for the version)
2. ./configure
3. make
4. make install
```

The exceution of the last two commands may take long (longer than 30mins in the testing configuration[1]). The installation will be complete if this process has generated a new `ql` folder in `/usr/local/include` and some new libraries in the `/usr/local/lib` directory.

[1]The configuration used to build and test QuantLib has been for this project a virtual machine running on MSWindows Vista. The virtual machine is powered by Sun xVM Virtual Box v.1.6.4 with 512MB of RAM memory. The OS on the virtual machine is Ubuntu 8.04

Using QuantLib

To be able to use QuantLib classes in our code, we will need to include the quantlib headers with the instruction: `#include<ql/quantlib.hpp>`. It is also necessary to link our program to the library we compiled in the previous step. This is done with the following compiler/linker command `g++ program.cpp -lQuantLib`.

Appendix C

QuantLib organization

The present appendix describes the contents of the library and how they all fit in the QuantLib framework. The presentation is structured according to QuantLib's directories structure to provide a more practical approach.

First, the main directory contains the base classes that implement basic concepts such as instrument, pricing engine, currency, cash-flow, stochastic process, term structure or error.

Cash flows: Cash flows have a key role in finance. We model them as a separate class that inherits from event. The information about amount and time is contained in this base class. The cash flow class provides the base for more elaborated magnitudes such as coupons that will have to bear information on accrual.

Currencies: contains currencies subclasses. For each currency, name, format, rounding and other conventions are specified. An algebra is defined over each currency allowing basic operations to be performed. One of the main weaknesses admited by the developers of the project is that a practitioners oriented library such as QuantLib should be working with this currency types rather than with the Real or Int actually coded. Although the need for a change is quite obvious, this would imply recoding practically all the library. Hence, a change is under way but it may take long to see the light, especially for a dynamically evolving project.

Experimental: This folder contains the new contributions that are being tested before finally allowing them into the library release.

Indexes: Similarly to the case of currencies, this directory contains the specifications for several indexes. The indexes coded range from equities to interest rates or inflation, all with the appropriate interface. The index base class is contained in the base directory.

Instruments: This directory contains all the specifications of financial instruments. Several instruments are found down a chain of inheritances that permits to work at different abstraction levels. In this way, a European option will inherit from a vanilla option, which in turn will inherit from one asset option, a subclass of option which is in turn a subclass of instrument. As new instruments are being added, there is added complexity in the library. It will then be natural to see the contents of this directory split in further subdirectories as the project carries on. One last important remark is the existence of the payoff class. As we have seen in the background chapter, what theoretically makes the object of a valuation is a particular payoff function rather than an instrument itself (the law of one price defines an equivalence relationship between the instruments that share the payoff function). Hence, in the specification of most instrument, what we really have is the determination of a payoff function along with an exercise (a concept also separately coded, although in the base directory).

Legacy: This folder contains files or structures that have been left behind. To keep the project useful for practitioners, a certain support must be given for legacy functions to make transitions smoother.

Math: This directory groups the mathematical tools implemented by the developers. The generality of mathematical tools makes them ideal candidates to be separately coded as independent classes to be reused in distinct contexts. We find from integrators to probability and statistics classes.

Methods: QuantLib implements basically three types of valuation methods that are reused in the distinct pricing engines. These three methodologies are Montecarlo, lattices and finite differences. If we go back to valuation background, we find that mostly financial instruments are priced by supposing some dynamics and then computing expectations under the risk-free measures induced by these dynamics. The straightforward way to proceed is to find the risk-free dynamics of the derivative and then compute its expectation by brute force, that is, by simulating several paths and averaging the final result over the samples. This is exactly what the Montecarlo framework is about. The Montecarlo subdirectory implements the classes that support the creation of multiple sample paths and the averaging of the obtained results. Another way to proceed is to generate some future

scenarios according to the dynamics and then perform a roll-back over time using the risk-free measures. Here lie the lattices methods, with variations according to the use of binomial, trinomial or other trees. Finally, often one can also view the valuation problem as the problem of solving a partial differential equation. PDEs can be solved either with closed form solutions, in which case the formula would be directly implemented in the pricing engine, or by using numerical methods such as finite differences, the third class of methods supported by QuantLib. Finite differences methods work by defining a grid and solving the discrete equation starting from a set of initial conditions and some boundary conditions. The process of finding the values for all the nodes of the grid can be stated in terms of an operator that will be dependent on the equation solved. In this way, we find a PDE class, along with operator and boundary conditions classes.

Models: This directory contains the implementation of pricing models that can be easily calibrated to market data. Instead of specifying the dynamics as a Brownian motion with a set of input term structures, here, the library allows users to choose one particular model and then let the model specify its parameters from input market data. For clarity, models are classified by asset classes.

Patterns: Most software applications rely on some design patterns that are helpful to solve in a studied way recurring problems. QuantLib makes use of a number of such design patterns to achieve a some of its features. The main design choice, that is, to separate the pricing engines from the instruments comes indeed from the application of a design pattern: the strategy pattern. Other design patterns are used through QuantLib and some of them have been generically coded in this directory. One noticeable case for QuantLib is the Observer pattern that is used to produce recalculations. We focus on this one as an example of a design pattern and as a relevant feature of the library. Developers wanted to give QuantLib a caching mechanism for the results so as to produce automatic recalculations whenever input data changes, a natural utility for a financial library. In fact, results are stored along with a variable that determines whether they are still valid or not. If the results are not valid and the user wants to retrieve them, they will be recomputed. The Observer pattern comes to determine this validity parameter. The pricing engine is declared as an observer and the magnitudes on which the price depends are declared as observables and linked to the observer pricing engine. Whenever the observables suffer modifications, this triggers an update method that informs the pricing engine that the stored result is no longer valid. Other patterns used by QuantLib are visitor, composite or singleton.

Pricing engines: This directory contains the implementation of pricing engines. Since pricing engines are in the end dependent on the instrument they price, we find a big variety of pricing engines according to the instrument they price. The proliferation of pricing engines has required this directory to be further structured in terms of the instruments priced by the engines. Although inheritance reduces the complexity of the actual code, the design of the library forces it to contain that many pricing engines.

Processes: The valuation of financial instruments is ultimately based on predictions of how things will evolve in the future. Since Black Scholes work, it has become a standard practice to model these future behaviours by using stochastic processes. Some asset classes have distinctive features that make some processes give better predictive results than others (mean reverting processes have proved to be more accurate for interest rate predictions than the standard Black Scholes Merton process). Thus, it makes sense to have the implementation of several processes in the library. Up to now, QuantLib supports multi-dimensional diffusion processes but the stochastic process base class only accounts for diffusion processes. Extensions will be surely made to jump and jump-diffusion some time soon as they become more widely used by market practitioners.

Quotes: Quotes may be regarded as the basic building block in financial markets so they deserve a class for their own right. However, one single class is not sufficient to capture the complexity of market quotes. Instruments may be quoted according to different related magnitudes following market conventions (eg. sometimes, options are quoted in terms of their implied volatility which is equivalent to a price through the Black Scholes model). This directory contains the subclasses specifications to capture some particular quotation systems widely used in financial markets.

Term structures: Time is critical in finance. The moment at which cash flows occur will be determinant to use one discount rate or another to obtain their present value. To model time dependent variables such as discount rates or volatilities, the authors have defined a term structure class whose mission is to handle time by considering a timeline from a specified reference date and relating the time dependent variables to this structure. The directory is further structured by asset classes.

Time: This directory mainly contains daycounters and calendars. To understand the role of such elements we come back to one of the functions of the termstructure: convert dates into points on a time line. To do so, we need a sort of distance between dates. This is provided by daycounters. However, daycounters may be dependent on factors such as work-

ing days that vary from one country to another. To take this into account, one can define several calendars and then make the daycounters run over the appropriate calendar. By doing so we can always compute a distance between two dates.

Utilities: This directory contains other technical code needed to support the rest of the library such as iterators or data parsers or formatters.

Appendix D

QuantLib program example: FD Valuation and Delta Hedging

In this appendix we present the code of an application that illustrates the structure of a QuantLib program. In particular, this application provides data on the accuracy of delta hedging for a European option.

We start by defining a number of context variables, then the product that we want to evaluate and finally we define a pricing engine for the option based on a finite difference scheme. The pricing engine provides a theoretical value for the option. In the Black Scholes framework, the value of the option should be the same as that of the self-financing portfolio built of stocks and cash that replicates the option when properly balanced through delta-hedging. This program provides the means to check whether the replication of the self-financed portfolio closely tracks the option value: the program first reads an initial market price for the underlying and sets the Black Scholes replicating portfolio. Then, as time passes, new market data comes for the price of the underlying and the option is re-evaluated and the portfolio properly balanced. At each period, the program outputs: the date, the price of the underlying, the theoretical value of the option, the value of the replicating portfolio and the composition of the portfolio.

The code is presented hereafter:

qlexample.cpp

```
#include<iostream>
#include<ql/quantlib.hpp>
using namespace QuantLib;

Month convertMonth(Integer mm);

int main ()
{
```

```
// 1-Context

Calendar calendar = UnitedStates(UnitedStates::NYSE);
Date todaysDate(5, June, 2008);
Natural settlementDays = 1;
Settings::instance().evaluationDate() = todaysDate;
DayCounter dayCounter = Actual365Fixed();

// 2-Instrument definition

Option::Type type(Option::Call);
Real strike = 130;
Date maturity(5, May, 2009);

boost::shared_ptr<StrikedTypePayoff> payoff
    (new PlainVanillaPayoff(type, strike));
boost::shared_ptr<Exercise> europeanExercise
    (new EuropeanExercise(maturity));

VanillaOption option (payoff,europeanExercise);

// 3-Engine definition (dynamics parameters)

    // 3.1-Declaration of parameters

boost::shared_ptr<SimpleQuote> underlying(new SimpleQuote);
boost::shared_ptr<SimpleQuote> dividendYield(new SimpleQuote);
boost::shared_ptr<SimpleQuote> riskFreeRate(new SimpleQuote);
boost::shared_ptr<SimpleQuote> volatility(new SimpleQuote);

    // 3.2-Declaration and initialization of handles

Handle<Quote> underlyingH(underlying);
Handle<Quote> dividendYieldH(dividendYield);
Handle<Quote> riskFreeRateH(riskFreeRate);
Handle<Quote> volatilityH(volatility);

Handle<YieldTermStructure> flatTermStructure
    (boost::shared_ptr<YieldTermStructure>(
    new FlatForward(settlementDays,calendar,riskFreeRateH,dayCounter)));
Handle<YieldTermStructure> flatDividendTS
    (boost::shared_ptr<YieldTermStructure>(
    new FlatForward(settlementDays,calendar,dividendYieldH,dayCounter)));
Handle<BlackVolTermStructure> flatVolTS
    (boost::shared_ptr<BlackVolTermStructure>(
    new BlackConstantVol(settlementDays,calendar,volatilityH,dayCounter)));

boost::shared_ptr<BlackScholesMertonProcess> bsmProcess(
    new BlackScholesMertonProcess
    (underlyingH, flatDividendTS, flatTermStructure, flatVolTS));

    // 3.3-Declaration of the pricing engine

boost::shared_ptr<PricingEngine> europeanEngine(
        new FDEuropeanEngine(bsmProcess));
```

```
    // 4-Associate instrument to pricing engine

    option.setPricingEngine(europeanEngine);

    // 5-Execution

        // 5.1-Variables declaration

    Integer dd, mm, yyyy; Date newDate;
    Real open, high, low, close, volume;
    Real stockHoldings, cashHoldings, portfolioValue, optionValue, cashGrowth;

    std::cin >> dd >> mm >> yyyy >> open >> high >> low >> close >> volume;

        // 5.2-Initial valuation

    Settings::instance().evaluationDate()= Date(dd,convertMonth(mm),yyyy);

    underlying->setValue(close);
    dividendYield->setValue(0.00);
    riskFreeRate->setValue(0.04);
    volatility->setValue(0.40);

    optionValue=option.NPV();
    stockHoldings=option.delta();
    cashHoldings=optionValue-stockHoldings*underlying->value();
    portfolioValue=cashHoldings+stockHoldings*underlying->value();
    std::cout << io::short_date(Settings::instance().evaluationDate()) << ' '
        << underlying->value() << ' ' << optionValue << ' ' << portfolioValue
        << ' ' << cashHoldings << ' ' << stockHoldings << std::endl;

        // 5.3-adjusting to price changes

    while(!std::cin.eof())
    {
        std::cin >> dd >> mm >> yyyy >> open >> high >> low >> close >> volume;
        if(std::cin.eof()) break;
        newDate = Date(dd,convertMonth(mm),yyyy);
        cashGrowth = cashHoldings * (1/flatTermStructure->discount(newDate)-1);

        Settings::instance().evaluationDate() = newDate;
        underlying->setValue(close);

        optionValue=option.NPV();
        cashHoldings=cashHoldings+cashGrowth-
                (option.delta()-stockHoldings)*underlying->value();
        stockHoldings=option.delta();
        portfolioValue=cashHoldings+stockHoldings*underlying->value();
        std::cout << io::short_date(Settings::instance().evaluationDate()) << ' '
            << underlying->value() << ' ' << optionValue << ' ' << portfolioValue
            << ' ' << cashHoldings << ' ' << stockHoldings << std::endl;
    }

}
```

```
Month convertMonth(Integer mm)
{
    switch(mm)
    {
        case 1: return January;
        case 2: return February;
        case 3: return March;
        case 4: return April;
        case 5: return May;
        case 6: return June;
        case 7: return July;
        case 8: return August;
        case 9: return September;
        case 10: return October;
        case 11: return November;
        case 12: return December;
        default: QL_FAIL ("Wrong argument: not a valid month");
    }
}
```

Appendix E

Uncertain volatility pricing code

In this appendix we prodide the code for the Uncertain Volatility Engine discussed in chapter 5.

The new code is split in seven different files.

One Asset Option Portfolio

```
#include <ql/instrument.hpp>
#include <ql/instruments/oneassetoption.hpp>
#include <list>
#include <utility>

#ifndef quantlib_oao_portfolio_hpp
#define quantlib_oao_portfolio_hpp


namespace QuantLib
{

  // This class implements a One Asset Portfolio Instrument
  // The implementation takes as reference the Composite Instrument
  // The choice has been to reimplement rather than to derive because
  // of several undesirable features observed in the Composite Instrument
  // implementation such as non-consolidation of the weights on the same asset

  class OAOPortfolio : public Instrument
  {

    public:

    typedef std::pair<boost::shared_ptr<OneAssetOption>,Real> holding;
    typedef std::list<holding>::iterator iterator;
    typedef std::list<holding>::const_iterator const_iterator;


    iterator find (boost::shared_ptr<OneAssetOption> opt)
    {
```

```
  for(iterator i=holdings_.begin(); i!=holdings_.end(); ++i)
    if (i->first == opt) return i;
  return holdings_.end();
}

void add (const boost::shared_ptr<OneAssetOption> &opt, Real r=1.0)
{
  if(r==0) return;

  iterator i = find(opt);
  if (i!=holdings_.end())
  {
    i->second+=r;
    if (i->second==0) holdings_.erase(i);
  }
  else
  {
    holdings_.push_back(std::make_pair(opt,r));
    // Although it would make sense to keep this registration,
    // it causes circular references when the portfolio is passed as
    // a parameter of the pricing engine
    //registerWith(opt);
  }
  update();

}

void substract (boost::shared_ptr<OneAssetOption> &opt, Real r=1.0)
{
  add(opt,-r);
}

bool isExpired() const
{
  for (const_iterator i=holdings_.begin(); i!=holdings_.end(); ++i)
    if (!i->first->isExpired()) return false;
  return true;
}

Real amount (boost::shared_ptr<OneAssetOption> opt)
{
  iterator i = find(opt);
  if (i!= holdings_.end()) return i->second;
  return 0;
}

std::list<holding> composition()
{
  return holdings_;
}

protected:

void performCalculations() const
{
  NPV_ = 0.0;
```

```
      for (const_iterator i=holdings_.begin(); i!=holdings_.end(); ++i)
      {
            NPV_ += i->second * i->first->NPV();
      }
    }

    private:

    std::list<holding> holdings_;

  };

}

#endif
```

Black Uncertain Vol

```
#ifndef quantlib_blackuncertainvol_hpp
#define quantlib_blackuncertainvol_hpp

#include <ql/termstructures/volatility/equityfx/blackvoltermstructure.hpp>
#include <ql/quotes/simplequote.hpp>
#include <ql/time/daycounters/actual365fixed.hpp>

/* Uncertain volatility class; constant values for the bounds */

namespace QuantLib
{

  enum MinMax
  // Auxiliar declaration to easily work with min/max
  {
    Max =  1,
    Min = -1
  };

  class BlackUncertainVol : public BlackVolatilityTermStructure
  {

    public:

    // Constructors taking several distinct sets of parameters
    BlackUncertainVol(const Date& referenceDate, const Calendar&,
      Volatility volmin, Volatility volmax, const DayCounter& dc);
    BlackUncertainVol(const Date& referenceDate, const Calendar&,
      const Handle<Quote>& volmin, const Handle<Quote>& volmax,
      const DayCounter& dc);
    BlackUncertainVol(Natural settlementDays, const Calendar&,
      Volatility volmin, Volatility volmax, const DayCounter& dc);
    BlackUncertainVol(Natural settlementDays, const Calendar&,
      const Handle<Quote>& volmin, const Handle<Quote>& volmax,
      const DayCounter& dc);

    // Term structure bounds
    Date maxDate() const;
    Real minStrike() const;
```

```
    Real maxStrike() const;

    // Acyclic visitor pattern adaptation
    virtual void accept(AcyclicVisitor&);

    // Public function to retrieve volatility bounds
    Volatility blackVol2 (Time t, Real r, MinMax m) const;

    protected:

    // Implementation of the derived function
    virtual Volatility blackVolImpl(Time t, Real) const;

    private:

    // Data structures to store constant vol bounds
    Handle<Quote> volmin_;
    Handle<Quote> volmax_;

  };

/*************************** METHODS IMPLEMENTATION ***************************/

  inline BlackUncertainVol::BlackUncertainVol(const Date& referenceDate,
    const Calendar& cal, Volatility volmin, Volatility volmax,
    const DayCounter& dc) :
        BlackVolatilityTermStructure(referenceDate, cal, Following, dc),
        volmin_(boost::shared_ptr<Quote>(new SimpleQuote(volmin))),
        volmax_(boost::shared_ptr<Quote>(new SimpleQuote(volmax))) {}

  inline BlackUncertainVol::BlackUncertainVol(const Date& referenceDate,
    const Calendar& cal, const Handle<Quote>& volmin,
    const Handle<Quote>& volmax, const DayCounter& dc):
        BlackVolatilityTermStructure(referenceDate, cal, Following, dc),
        volmin_(volmin), volmax_(volmax)
  {
    registerWith(volmin_);
    registerWith(volmax_);
  }

  inline BlackUncertainVol::BlackUncertainVol(Natural settlementDays,
    const Calendar& cal, Volatility volmin, Volatility volmax,
    const DayCounter& dc):
        BlackVolatilityTermStructure(settlementDays, cal, Following, dc),
        volmin_(boost::shared_ptr<Quote>(new SimpleQuote(volmin))),
        volmax_(boost::shared_ptr<Quote>(new SimpleQuote(volmax))) {}

  inline BlackUncertainVol::BlackUncertainVol(Natural settlementDays,
    const Calendar& cal, const Handle<Quote>& volmin,
    const Handle<Quote>& volmax, const DayCounter& dc):
        BlackVolatilityTermStructure(settlementDays, cal, Following, dc),
        volmin_(volmin), volmax_(volmax)
  {
    registerWith(volmin_);
    registerWith(volmax_);
  }
```

```
  inline Date BlackUncertainVol::maxDate() const
  {
    return Date::maxDate();
  }

  inline Real BlackUncertainVol::minStrike() const
  {
    return QL_MIN_REAL;
  }

  inline Real BlackUncertainVol::maxStrike() const
  {
    return QL_MAX_REAL;
  }

  inline void BlackUncertainVol::accept(AcyclicVisitor& v)
  {
    Visitor<BlackUncertainVol>* v1 =
      dynamic_cast<Visitor<BlackUncertainVol>*>(&v);

    if (v1 != 0) v1->visit(*this);
      else BlackVolatilityTermStructure::accept(v);
  }

  inline Volatility BlackUncertainVol::blackVol2(Time t,Real r,MinMax m) const
  {
    switch(m)
    {
      case  1: return volmax_->value(); break;
      case -1: return volmin_->value(); break;
      default: QL_FAIL("Wrong argument for uncertain vol: +1=max, -1=min");
    }
  }

  inline Volatility BlackUncertainVol::blackVolImpl(Time, Real) const
  // By disabling this function, we disable all the methods that are not
  // consistent with the uncertain volatility framework and that try to
  // retrieve a particular value for the volatility at a given time.
  {
    QL_FAIL("NOT AVAILABLE FOR UNCERTAIN VOL MODEL");
    return 0;
  }
}

#endif
```

BSB Process

```
#ifndef quantlib_black_scholes_barenblatt_process_hpp
#define quantlib_black_scholes_barenblatt_process_hpp

#include<ql/processes/blackscholesprocess.hpp>
#include <ql/termstructures/volatility/equityfx/blackuncertainvol.hpp>

namespace QuantLib
{
```

```
class BlackScholesBarenblattProcess : public GeneralizedBlackScholesProcess
{
  public:

    BlackScholesBarenblattProcess(
      const Handle<Quote>& x0,
      const Handle<YieldTermStructure>& dividendTS,
      const Handle<YieldTermStructure>& riskFreeTS,
      const Handle<BlackVolTermStructure>& blackVolTS,
      const boost::shared_ptr<discretization>& disc =
      boost::shared_ptr<discretization>(new EulerDiscretization));

  Volatility volMin (Time t, Real x);
  Volatility volMax (Time t, Real x);

  void update();

  private:

  Handle<BlackUncertainVol> blackUncertainVolatility_;

};


BlackScholesBarenblattProcess::BlackScholesBarenblattProcess(
    const Handle<Quote>& x0,
    const Handle<YieldTermStructure>& dividendTS,
    const Handle<YieldTermStructure>& riskFreeTS,
    const Handle<BlackVolTermStructure>& blackVolTS,
    const boost::shared_ptr<discretization>& d) :
  GeneralizedBlackScholesProcess(x0,dividendTS,riskFreeTS,blackVolTS,d)
{
  Handle<BlackVolTermStructure> aux1 = this->blackVolatility();
  boost::shared_ptr<BlackVolTermStructure> aux2 = *aux1;
      boost::shared_ptr<BlackUncertainVol> aux3 =
        boost::dynamic_pointer_cast<BlackUncertainVol>(aux2);

  blackUncertainVolatility_= Handle<BlackUncertainVol>(aux3);
}

void BlackScholesBarenblattProcess::update()
{
  Handle<BlackVolTermStructure> aux1 = this->blackVolatility();
  boost::shared_ptr<BlackVolTermStructure> aux2 = *aux1;
      boost::shared_ptr<BlackUncertainVol> aux3 =
        boost::dynamic_pointer_cast<BlackUncertainVol>(aux2);
  blackUncertainVolatility_= Handle<BlackUncertainVol>(aux3);

  GeneralizedBlackScholesProcess::update();
}

Volatility BlackScholesBarenblattProcess::volMin (Time t, Real x)
{
  return blackUncertainVolatility_->blackVol2(t,x,Min);
}
```

```
  Volatility BlackScholesBarenblattProcess::volMax (Time t, Real x)
  {
    return blackUncertainVolatility_->blackVol2(t,x,Max);
  }

}

#endif
```

BSB PDE

```
#ifndef quantlib_pdebsb_hpp
#define quantlib_pdebsb_hpp

#include <ql/methods/finitedifferences/pde.hpp>
#include <ql/processes/blackscholesprocess.hpp>

#include <boost/timer.hpp>

namespace QuantLib
{

  class PdeBSB : public PdeSecondOrderParabolic
  {

    public:

    typedef boost::shared_ptr<BlackScholesBarenblattProcess> argument_type;
    typedef LogGrid grid_type;

    PdeBSB(const argument_type & process) : process_(process) {};

    virtual Real diffusion(Time t, Real x) const
    {
      return process_->diffusion(t, x);
    }
    virtual Real drift(Time t, Real x) const
    {
      return process_->drift(t, x);
    }
    virtual Real discount(Time t, Real) const
    {
      if (std::fabs(t) < 1e-8) t = 0;
      return process_->riskFreeRate()->
        forwardRate(t,t,Continuous,NoFrequency,true);
    }

    void generateUncertainOperator(Time t, const TransformedGrid &tg,
      SampledCurve *intrinsicValues, TridiagonalOperator &L) const
    {

      boost::timer timer;

      if(t<0) t=0;

      Real gamma, sigma, sigma2, nu, r;
```

```
      // Auxiliary calculations for the drift nu which can no longer be
      // computed as drift(t,tg.grid(i))
      Time t1 = t + 0.0001;
      Real riskFree = process_->riskFreeRate()->forwardRate
            (t,t1,Continuous,NoFrequency,true);
      Real divYield = process_->dividendYield()->forwardRate
            (t,t1,Continuous,NoFrequency,true);

      for (Size i=1; i < tg.size() - 1; i++)
      {
        // The operator's dependence on the sign of gamma is introduced here

        gamma= intrinsicValues->value(i-1)-2*intrinsicValues->value(i)
          +intrinsicValues->value(i+1);

        if (gamma>=0) sigma = process_->volMax(t, tg.grid(i));
        else sigma = process_->volMin(t, tg.grid(i));

        sigma2 = sigma * sigma;
        nu= riskFree  - divYield - 0.5 * sigma2;

        Real pd = -(sigma2/tg.dxm(i)-nu)/ tg.dx(i);
        Real pu = -(sigma2/tg.dxp(i)+nu)/ tg.dx(i);
        Real pm = sigma2/(tg.dxm(i) * tg.dxp(i))+riskFree;

        L.setMidRow(i, pd,pm,pu);
      }

    }

    private:

    const argument_type process_;

  };

}

#endif
```

BSB Operator

```
#ifndef quantlib_bsb_operator_hpp
#define quantlib_bsb_operator_hpp

#include <ql/methods/finitedifferences/tridiagonaloperator.hpp>
#include <ql/methods/finitedifferences/pdebsb.hpp>
#include <ql/processes/blackscholesprocess.hpp>

namespace QuantLib
{

/****************************** Timesetter class ******************************/

  template <class PdeClass>
  class ModifiedTimeSetter:public TridiagonalOperator::TimeSetter
```

```
  // The aim of the modified time setter is to account for the evolution
  // of the finite difference operator according to the values appearing
  // during the roll-back process
  {
    public:

    template <class T>
    ModifiedTimeSetter(const typename PdeClass::grid_type &grid,
      SampledCurve *refprices, T process) :
      grid_(grid), refprices_(refprices), pde_(process) {}

    void setTime(Time t, TridiagonalOperator &L) const
    // Each time this function is called, the operator is recalculated.
    // Here, the interesting part for our aim is the dependence of the new
    // operator with respect to the initrinsic values that is coded through
    // a pointer.
    {
      pde_.generateUncertainOperator(t, grid_, refprices_, L);
    }

    private:

    typename PdeClass::grid_type grid_;
    SampledCurve *refprices_;
    PdeClass pde_;
  };


/******************************** Operator class ********************************/

  template <class PdeClass>

  class BSBOperator:public TridiagonalOperator
  {
    public:

      template <class T>
      BSBOperator(const Array &grid, SampledCurve *refprices, T process,
        Time residualTime = 0.0) : TridiagonalOperator(grid.size())
      {
        PdeBSB::grid_type logGrid(grid);
        timeSetter_ = boost::shared_ptr<ModifiedTimeSetter<PdeClass> >(
          new ModifiedTimeSetter<PdeClass>(logGrid, refprices, process));
        setTime(residualTime);
      }
  };
}

#endif
```

Modified FD Model

```
#ifndef quantlib_finite_difference_model_mod_hpp
#define quantlib_finite_difference_model_mod_hpp

#include <ql/methods/finitedifferences/stepcondition.hpp>
#include <ql/methods/finitedifferences/boundarycondition.hpp>
```

```
#include <ql/methods/finitedifferences/operatortraits.hpp>

namespace QuantLib {

    //! Generic finite difference model
    /*! \ingroup findiff */
    template<class Evolver>
    class FiniteDifferenceModelMod {
      public:
        typedef typename Evolver::traits traits;
        typedef typename traits::operator_type operator_type;
        typedef typename traits::array_type array_type;
        typedef typename traits::bc_set bc_set;
        typedef typename traits::condition_type condition_type;
        // constructors
        FiniteDifferenceModelMod(const operator_type& L,
                                 const bc_set& bcs,
                                 const std::vector<Time>& stoppingTimes =
                                    std::vector<Time>())
        : evolver_(L,bcs), stoppingTimes_(stoppingTimes) {
            std::sort(stoppingTimes_.begin(), stoppingTimes_.end());
            std::vector<Time>::iterator last =
                std::unique(stoppingTimes_.begin(), stoppingTimes_.end());
            stoppingTimes_.erase(last, stoppingTimes_.end());
        }
        FiniteDifferenceModelMod(const Evolver& evolver,
                                 const std::vector<Time>& stoppingTimes =
                                                        std::vector<Time>())
        : evolver_(evolver), stoppingTimes_(stoppingTimes) {
            std::sort(stoppingTimes_.begin(), stoppingTimes_.end());
            std::vector<Time>::iterator last =
                std::unique(stoppingTimes_.begin(), stoppingTimes_.end());
            stoppingTimes_.erase(last, stoppingTimes_.end());
        }
        // methods
        // array_type grid() const { return evolver.xGrid(); }
        const Evolver& evolver() const{ return evolver_; }
        /*! solves the problem between the given times.
            \warning being this a rollback, <tt>from</tt> must be a later
                     time than <tt>to</tt>.
        */

        // A dependency on a reference array is introduced
        void rollback(array_type& a, array_type& b,
                      Time from,
                      Time to,
                      Size steps)
        {
            // The dependence is carried forward to rollbackImpl
            rollbackImpl(a,b,from,to,steps, (const condition_type*) 0);
        }
        /*! solves the problem between the given times,
            applying a condition at every step.
            \warning being this a rollback, <tt>from</tt> must be a later
                     time than <tt>to</tt>.
        */
```

```
  // A dependency on a reference array is introduced
  void rollback(array_type& a, array_type& b,
                Time from,
                Time to,
                Size steps,
                const condition_type& condition) {
      rollbackImpl(a,b,from,to,steps,&condition);
  }
private:
  void rollbackImpl(array_type& a, array_type& b,
                    Time from,
                    Time to,
                    Size steps,
                    const condition_type* condition) {

      QL_REQUIRE(from >= to,
                 "trying to roll back from " << from << " to " << to);

      Time dt = (from-to)/steps, t = from;
      evolver_.setStep(dt);

      for (Size i=0; i<steps; ++i, t -= dt) {
          Time now = t, next = t-dt;
          bool hit = false;
          for (Integer j = stoppingTimes_.size()-1; j >= 0 ; --j) {
              if (next <= stoppingTimes_[j] && stoppingTimes_[j] < now) {
                  // a stopping time was hit
                  hit = true;

                  // perform a small step to stoppingTimes_[j]...
                  evolver_.setStep(now-stoppingTimes_[j]);
                  evolver_.step(a,now);
                  evolver_.step(b,now);
                  if (condition)
                  {
                      condition->applyTo(a,stoppingTimes_[j]);
                      condition->applyTo(b,stoppingTimes_[j]);
                  }
                  // ...and continue the cycle
                  now = stoppingTimes_[j];
              }
          }
          // if we did hit...
          if (hit)
          {
              // ...we might have to make a small step to
              // complete the big one...
              if (now > next)
              {
                  evolver_.setStep(now - next);
                  evolver_.step(a,now);
                  evolver_.step(b,now);
                  if (condition)
                  {
                      condition->applyTo(a,next);
                      condition->applyTo(b,next);
```

```
                        }
                    }
                    // ...and in any case, we have to reset the
                    // evolver to the default step.
                    evolver_.setStep(dt);
                }
                else
                {
                    // if we didn't, the evolver is already set to the
                    // default step, which is ok for us.
                    evolver_.step(a,now);
                    evolver_.step(b,now);
                    if (condition)
                    {
                        condition->applyTo(a, next);
                        condition->applyTo(b, next);
                    }
                }
            }
        }
        Evolver evolver_;
        std::vector<Time> stoppingTimes_;
    };


typedef FiniteDifferenceModelMod<CrankNicolson<TridiagonalOperator>  >
    StandardFiniteDifferenceModelMod;

}


#endif
```

FD Uncertain Engine

```
#ifndef quantlib_fd_uncertain_engine_hpp
#define quantlib_fd_uncertain_engine_hpp

#include <ql/pricingengine.hpp>
#include <ql/methods/finitedifferences/tridiagonaloperator.hpp>
#include <ql/methods/finitedifferences/boundarycondition.hpp>
#include <ql/processes/blackscholesprocess.hpp>
#include <ql/math/sampledcurve.hpp>
#include <ql/payoff.hpp>
#include <ql/processes/bsbprocess.hpp>
#include <ql/methods/finitedifferences/bsboperator.hpp>
#include <ql/methods/finitedifferences/pdebsb.hpp>

namespace QuantLib {

  class FDUncertainEngine
  // This class contains the methods for finite difference valuation;
  // It is built upon the FDVanillaEngine class making some changes to
  // adapt the original code to the requirements of the framework.
  {

    public:
```

```
  FDUncertainEngine(
    const boost::shared_ptr<BlackScholesBarenblattProcess>& process,
    Size timeSteps, Size gridPoints, bool timeDependent = false) :
        process_(process), timeSteps_(timeSteps),
        gridPoints_(gridPoints), timeDependent_(timeDependent),
        intrinsicValues_(gridPoints), BCs_(2) {}

  virtual ~FDUncertainEngine() {}

  // accessors
  const Array& grid() const { return intrinsicValues_.grid(); }

  protected:

  // methods
  virtual void setupArguments(const PricingEngine::arguments*) const;
  virtual void setGridLimits() const;
  virtual void setGridLimits(Real, Time) const;
  virtual void initializeInitialCondition() const;
  virtual void initializeBoundaryConditions() const;
  virtual void initializeOperator(SampledCurve *prices_ptr) const;
  virtual Time getResidualTime() const;

  // data
  boost::shared_ptr<BlackScholesBarenblattProcess> process_;
  Size timeSteps_, gridPoints_;
  bool timeDependent_;
  mutable Real requiredGridValue_;
  mutable Date exerciseDate_;
  mutable boost::shared_ptr<Payoff> payoff_;
  mutable TridiagonalOperator finiteDifferenceOperator_;
  mutable SampledCurve intrinsicValues_;
  typedef BoundaryCondition<TridiagonalOperator> bc_type;
  mutable std::vector<boost::shared_ptr<bc_type> > BCs_;

  // temporaries
  mutable Real sMin_, center_, sMax_;

  protected:

  void ensureStrikeInGrid() const;

  private:

  // temporaries
  mutable Real gridLogSpacing_;
  Size safeGridPoints(Size gridPoints, Time residualTime) const;
  static const Real safetyZoneFactor_;

};

template <typename base, typename engine>
class FDUncertainEngineAdapter : public base, public engine
{
```

```
    public:

    FDUncertainEngineAdapter (
      const boost::shared_ptr<BlackScholesBarenblattProcess>& process,
      Size timeSteps=100, Size gridPoints=100, bool timeDependent = false) :
          base(process, timeSteps, gridPoints,timeDependent)
    {
      this->registerWith(process);
    }

    private:

    void calculate() const
    {
      base::setupArguments(&(this->arguments_));
      base::calculate(&(this->results_));
    }

  };

/************************** Methods implementation ****************************/

  const Real FDUncertainEngine::safetyZoneFactor_ = 1.1;

  void FDUncertainEngine::setGridLimits() const
  {
    setGridLimits(process_->stateVariable()->value(), getResidualTime());
    ensureStrikeInGrid();
  }

  void FDUncertainEngine::setupArguments
    (const PricingEngine::arguments* a) const
  {
    const OneAssetOption::arguments * args =
      dynamic_cast<const OneAssetOption::arguments *>(a);
    QL_REQUIRE(args, "incorrect argument type");
    exerciseDate_ = args->exercise->lastDate();
    payoff_ = args->payoff;
    requiredGridValue_ =
    boost::dynamic_pointer_cast<StrikedTypePayoff>(payoff_)->strike();
  }

  void FDUncertainEngine::setGridLimits(Real center, Time t) const
  {
    QL_REQUIRE(center > 0.0, "negative or null underlying given");
    center_ = center;
    Size newGridPoints = safeGridPoints(gridPoints_, t);
    if (newGridPoints > intrinsicValues_.size())
    {
      intrinsicValues_ = SampledCurve(newGridPoints);
    }

    Real volSqrtTime;
    Volatility volmin = process_->volMin(t,center_);
    Volatility volmax = process_->volMax(t,center_);
    if(volmax>volmin)
```

```
    volSqrtTime = std::sqrt(t)*volmax;
  else
    volSqrtTime = std::sqrt(t)*volmin;

  // the prefactor fine tunes performance at small volatilities
  Real prefactor = 1.0 + 0.02/volSqrtTime;
  Real minMaxFactor = std::exp(4.0 * prefactor * volSqrtTime);
  sMin_ = center_/minMaxFactor;  // underlying grid min value
  sMax_ = center_*minMaxFactor;  // underlying grid max value
}

void FDUncertainEngine::ensureStrikeInGrid() const
// ensure strike is included in the grid
{
  boost::shared_ptr<StrikedTypePayoff> striked_payoff =
    boost::dynamic_pointer_cast<StrikedTypePayoff>(payoff_);
  if (!striked_payoff) return;
  Real requiredGridValue = striked_payoff->strike();

  if (sMin_ > requiredGridValue/safetyZoneFactor_)
  {
    sMin_ = requiredGridValue/safetyZoneFactor_;
    // enforce central placement of the underlying
    sMax_ = center_/(sMin_/center_);
  }

  if(sMax_ < requiredGridValue*safetyZoneFactor_)
  {
    sMax_ = requiredGridValue*safetyZoneFactor_;
    // enforce central placement of the underlying
    sMin_ = center_/(sMax_/center_);
  }
}

void FDUncertainEngine::initializeInitialCondition() const
{
  intrinsicValues_.setLogGrid(sMin_, sMax_);
  intrinsicValues_.sample(*payoff_);
}

void FDUncertainEngine::initializeOperator(SampledCurve *refprices) const
// This function has been redesigned to introduce the operator's dependence
// on the reference prices
{
  finiteDifferenceOperator_ = BSBOperator<PdeBSB>
    (intrinsicValues_.grid(),refprices,process_,getResidualTime());
}

void FDUncertainEngine::initializeBoundaryConditions() const
{
  BCs_[0] = boost::shared_ptr<bc_type>(new NeumannBC
    (intrinsicValues_.value(1) - intrinsicValues_.value(0),
    NeumannBC::Lower));
  BCs_[1] = boost::shared_ptr<bc_type>(new NeumannBC
    (intrinsicValues_.value(intrinsicValues_.size()-1) -
     intrinsicValues_.value(intrinsicValues_.size()-2),
```

```
      NeumannBC::Upper));
  }

  Time FDUncertainEngine::getResidualTime() const
  {
    return process_->time(exerciseDate_);
  }

  // safety check to be sure we have enough grid points.
  Size FDUncertainEngine::safeGridPoints(Size gridPoints,
    Time residualTime) const
  {
    static const Size minGridPoints = 10;
    static const Size minGridPointsPerYear = 2;
    return std::max(gridPoints,residualTime > 1.0 ?
      static_cast<Size>((minGridPoints + (residualTime-1.0) *
      minGridPointsPerYear)) : minGridPoints);
  }

}

#endif
```

FD Uncertain European Engine

```
#ifndef quantlib_fd_uncertain_european_engine_mod_hpp
#define quantlib_fd_uncertain_european_engine_mod_hpp

#include <ql/instruments/oneassetoption.hpp>
#include <ql/instruments/oaoportfolio.hpp>
#include <ql/pricingengines/vanilla/fduncertainengine.hpp>
#include <ql/math/sampledcurve.hpp>
#include <ql/methods/finitedifferences/finitedifferencemodelmod.hpp>
#include <ql/processes/bsbprocess.hpp>
#include <vector>

namespace QuantLib
{

  //! Pricing engine for European options using finite-differences
  //  in the uncertain volatility framework

  class FDUncertainRefEuropeanEngine : public OneAssetOption::engine,
    public FDUncertainEngine
  {

    public:

    FDUncertainRefEuropeanEngine(
      const boost::shared_ptr<BlackScholesBarenblattProcess>& process,
      const boost::shared_ptr<OAOPortfolio>& refportfolio,
      Size timeSteps=100, Size gridPoints=100, bool timeDependent = false):
        FDUncertainEngine(process, timeSteps, gridPoints, timeDependent),
        refportfolio_(refportfolio), prices_(gridPoints), refprices_(gridPoints)
    {
      registerWith(process);
      registerWith(refportfolio);
```

```
    }

    class PortfolioPayoffFunction
    {
      public:

        PortfolioPayoffFunction(OAOPortfolio portfolio):portfolio_(portfolio){}
        Real operator()(Real price) const;

      private:
        mutable OAOPortfolio portfolio_;
    };


    private:

    boost::shared_ptr<OAOPortfolio> refportfolio_;
    mutable SampledCurve refprices_;
    mutable SampledCurve prices_;
    void calculate() const;
    void initializeReference() const;

  };

/*************************** Methods implementation ****************************/


  void FDUncertainRefEuropeanEngine::calculate() const
  {
    SampledCurve *refprices_ptr = &refprices_;

    setupArguments(&arguments_);
    setGridLimits();
    initializeInitialCondition();
    initializeReference();
    initializeOperator(refprices_ptr);
    initializeBoundaryConditions();
    StandardFiniteDifferenceModelMod model(finiteDifferenceOperator_, BCs_);

    prices_ = intrinsicValues_;

    model.rollback(prices_.values(), refprices_.values(),
      getResidualTime(), 0, timeSteps_);

    results_.value = prices_.valueAtCenter();
    results_.delta = prices_.firstDerivativeAtCenter();
    results_.gamma = prices_.secondDerivativeAtCenter();
    results_.additionalResults["priceCurve"] = prices_;
  }

  void FDUncertainRefEuropeanEngine::initializeReference() const
  {
    PortfolioPayoffFunction samplingfunction(*refportfolio_);
    refprices_.setLogGrid(sMin_, sMax_);
    refprices_.sample(samplingfunction);
  }
```

```
  Real FDUncertainRefEuropeanEngine::PortfolioPayoffFunction::operator()
    (Real price) const
  {
    Real aux=0.;
    std::list<OAOPortfolio::holding> auxP=portfolio_.composition();

    for(OAOPortfolio::iterator iter=auxP.begin(); iter!=auxP.end(); iter++)
    {
      aux +=  (iter->first->payoff()->operator()(price))*(iter->second);
    }

    return aux;
  }

}


#endif
```

Appendix F

Uncertain volatility pricing example

uvexample.cpp

```
#include<iostream>
#include<ql/quantlib.hpp>

#include<ql/termstructures/volatility/equityfx/blackuncertainvol.hpp>
#include<ql/pricingengines/vanilla/fduncertainengine.hpp>
#include<ql/pricingengines/vanilla/fdrefeuropeanengine.hpp>
#include<ql/pricingengines/vanilla/fduncertaineuropeanengine.hpp>
#include<ql/methods/finitedifferences/bsboperator.hpp>
#include<ql/methods/finitedifferences/pdebsb.hpp>
#include<ql/instruments/oaoportfolio.hpp>


using namespace QuantLib;

int main()
{

    // 1-Context

    Calendar calendar = TARGET();
    Date todaysDate(5, November, 2008);
    Natural settlementDays = 1;
    Settings::instance().evaluationDate() = todaysDate;
    DayCounter dayCounter = Actual365Fixed();

    // 2-Instruments definition

    Option::Type type(Option::Call);
    Date maturity(5, May, 2009);
    Real strike1 = 90;
    Real strike2 = 100;

        // 2.1- Maturity

    boost::shared_ptr<Exercise> europeanExercise(
```

```
    new EuropeanExercise(maturity));

    // 2.2- Payoffs (Spread and becnhmark instruments)

boost::shared_ptr<StrikedTypePayoff> payoffcall1(
    new PlainVanillaPayoff(type, strike1));

boost::shared_ptr<StrikedTypePayoff> payoffcall2(
    new PlainVanillaPayoff(type, strike2));

    // 2.3- Instruments

boost::shared_ptr<OneAssetOption> call1 (
    new OneAssetOption(payoffcall1, europeanExercise));
boost::shared_ptr<OneAssetOption> call2 (
    new OneAssetOption(payoffcall2, europeanExercise));

OAOPortfolio portfolio;

portfolio.add(call1,1);
portfolio.add(call2,-1);

// 3-Engine definition (dynamics parameters)

    // 3.1-Declaration of parameters

boost::shared_ptr<SimpleQuote> underlying (new SimpleQuote);
boost::shared_ptr<SimpleQuote> dividendYield (new SimpleQuote(0.00));
boost::shared_ptr<SimpleQuote> riskFreeRate (new SimpleQuote(0.05));
boost::shared_ptr<SimpleQuote> volmin (new SimpleQuote(0.10));
boost::shared_ptr<SimpleQuote> volmax (new SimpleQuote(0.40));

    // 3.2-Declaration and initialization of handles

Handle<Quote> underlyingH(underlying);
Handle<Quote> dividendYieldH(dividendYield);
Handle<Quote> riskFreeRateH(riskFreeRate);
Handle<Quote> volminH(volmin);
Handle<Quote> volmaxH(volmax);

Handle<YieldTermStructure> flatTermStructure(
    boost::shared_ptr<YieldTermStructure>(
    new FlatForward(settlementDays, calendar, riskFreeRateH, dayCounter)));
Handle<YieldTermStructure> flatDividendTS(
    boost::shared_ptr<YieldTermStructure>(
    new FlatForward(settlementDays, calendar, dividendYieldH, dayCounter)));
Handle<BlackVolTermStructure> uncertainVolTS(
    boost::shared_ptr<BlackVolTermStructure>(
    new BlackUncertainVol(settlementDays,calendar,
            volminH, volmaxH, dayCounter)));
Handle<BlackVolTermStructure> blackVolTSMin(
    boost::shared_ptr<BlackVolTermStructure>(
    new BlackConstantVol(settlementDays, calendar, volminH, dayCounter)));
Handle<BlackVolTermStructure> blackVolTSMax(
    boost::shared_ptr<BlackVolTermStructure>(
    new BlackConstantVol(settlementDays, calendar, volmaxH, dayCounter)));
```

```
    boost::shared_ptr<BlackScholesMertonProcess> bsbProcess(
        new BlackScholesMertonProcess(
        underlyingH, flatDividendTS,flatTermStructure, uncertainVolTS));
    boost::shared_ptr<BlackScholesMertonProcess> bsmMinProcess(
         new BlackScholesMertonProcess(
        underlyingH, flatDividendTS, flatTermStructure, blackVolTSMin));
    boost::shared_ptr<BlackScholesMertonProcess> bsmMaxProcess(
        new BlackScholesMertonProcess(
        underlyingH, flatDividendTS, flatTermStructure, blackVolTSMax));

        // 3.3-Declaration of the pricing engine

    Size Nsteps =100;

    boost::shared_ptr<PricingEngine> minVolEngine(
        new FDEuropeanEngine(bsmMinProcess,Nsteps,Nsteps));
    boost::shared_ptr<PricingEngine> maxVolEngine(
        new FDEuropeanEngine(bsmMaxProcess,Nsteps,Nsteps));
    boost::shared_ptr<PricingEngine> refVolEngine(
        new FDUncertainRefEuropeanEngine(bsbProcess,portfolio,Nsteps,Nsteps));

    // 4-Application

    for (Real u=5; u<=200; u+=5)
    {
        (*underlying).setValue(u);
        Real spreadvalue;
        Real call1maxvalue, call2minvalue;

        call1->setPricingEngine(refVolEngine);
        call2->setPricingEngine(refVolEngine);

        spreadvalue = portfolio.NPV();

        call1->setPricingEngine(maxVolEngine);
        call2->setPricingEngine(minVolEngine);

        call1maxvalue = call1->NPV();
        call2minvalue = call2->NPV();

        std::cout << u << " " << spreadvalue << " " <<
            call1maxvalue - call2minvalue << std::endl;
    }
}
```

Appendix G

Uncertain volatility pricing example

luvexample.cpp

```
#include<iostream>
#include<ql/quantlib.hpp>

#include<ql/termstructures/volatility/equityfx/blackuncertainvol.hpp>
#include<ql/pricingengines/vanilla/fduncertainengine.hpp>
#include<ql/pricingengines/vanilla/fdrefeuropeanengine.hpp>
#include<ql/methods/finitedifferences/bsboperator.hpp>
#include<ql/methods/finitedifferences/pdebsb.hpp>
#include<ql/instruments/oaoportfolio.hpp>
#include<ql/processes/bsbprocess.hpp>


using namespace QuantLib;

int main()
{

    // 1-Context

    Calendar calendar = TARGET();
    Date todaysDate(5, November, 2008);
    Natural settlementDays = 1;
    Settings::instance().evaluationDate() = todaysDate;
    DayCounter dayCounter = Actual365Fixed();

    // 2-Instruments definition

    Option::Type type(Option::Call);
    Date maturity(5, May, 2009);
    Real strike1 = 100;
    Real strike2 = 110;

        // 2.1- Maturity

    boost::shared_ptr<Exercise> europeanExercise(
```

```
    new EuropeanExercise(maturity));

    // 2.2- Payoffs (Spread and becnhmark instruments)

boost::shared_ptr<StrikedTypePayoff> payoffcall1(
    new PlainVanillaPayoff(type, strike1));

boost::shared_ptr<StrikedTypePayoff> payoffcall2(
    new PlainVanillaPayoff(type, strike2));

    // 2.3- Instruments

boost::shared_ptr<OneAssetOption> call1 (
    new OneAssetOption(payoffcall1, europeanExercise));
boost::shared_ptr<OneAssetOption> call2 (
    new OneAssetOption(payoffcall2, europeanExercise));

boost::shared_ptr<OAOPortfolio> residual (new OAOPortfolio);


// 3-Engine definition (dynamics parameters)

    // 3.1-Declaration of parameters

boost::shared_ptr<SimpleQuote> underlying (new SimpleQuote(100));
boost::shared_ptr<SimpleQuote> dividendYield (new SimpleQuote(0.00));
boost::shared_ptr<SimpleQuote> riskFreeRate (new SimpleQuote(0.07));
boost::shared_ptr<SimpleQuote> volmin (new SimpleQuote(0.32));
boost::shared_ptr<SimpleQuote> volmax (new SimpleQuote(0.02));

    // 3.2-Declaration and initialization of handles

Handle<Quote> underlyingH(underlying);
Handle<Quote> dividendYieldH(dividendYield);
Handle<Quote> riskFreeRateH(riskFreeRate);
Handle<Quote> volminH(volmin);
Handle<Quote> volmaxH(volmax);

Handle<YieldTermStructure> flatTermStructure(
    boost::shared_ptr<YieldTermStructure>(
    new FlatForward(settlementDays, calendar, riskFreeRateH, dayCounter)));
Handle<YieldTermStructure> flatDividendTS(
    boost::shared_ptr<YieldTermStructure>(
    new FlatForward(settlementDays, calendar, dividendYieldH, dayCounter)));
Handle<BlackVolTermStructure> uncertainVolTS(
    boost::shared_ptr<BlackVolTermStructure>(
    new BlackUncertainVol(settlementDays, calendar,
            volminH, volmaxH, dayCounter)));
Handle<BlackVolTermStructure> blackVolTSMin(
    boost::shared_ptr<BlackVolTermStructure>(
    new BlackConstantVol(settlementDays, calendar, volminH, dayCounter)));
Handle<BlackVolTermStructure> blackVolTSMax(
    boost::shared_ptr<BlackVolTermStructure>(
    new BlackConstantVol(settlementDays, calendar, volmaxH, dayCounter)));

boost::shared_ptr<BlackScholesBarenblattProcess> bsbProcess(
```

```
    new BlackScholesBarenblattProcess(
    underlyingH, flatDividendTS,flatTermStructure, uncertainVolTS));
boost::shared_ptr<BlackScholesMertonProcess> bsmMinProcess(
     new BlackScholesMertonProcess(
    underlyingH, flatDividendTS, flatTermStructure, blackVolTSMin));
boost::shared_ptr<BlackScholesMertonProcess> bsmMaxProcess(
    new BlackScholesMertonProcess(
    underlyingH, flatDividendTS, flatTermStructure, blackVolTSMax));

    // 3.3-Declaration of the pricing engine

Size Nsteps =500;

boost::shared_ptr<PricingEngine> minVolEngine(
        new FDEuropeanEngine(bsmMinProcess,Nsteps,Nsteps));
boost::shared_ptr<PricingEngine> maxVolEngine(
        new FDEuropeanEngine(bsmMaxProcess,Nsteps,Nsteps));
boost::shared_ptr<PricingEngine> refVolEngine(
        new FDUncertainRefEuropeanEngine(bsbProcess,residual,Nsteps,Nsteps));

// 4-Pricing Engine Specification

call1->setPricingEngine(refVolEngine);
call2->setPricingEngine(refVolEngine);

// 5-Application

residual->add(call1,-1);
residual->add(call2,+0);

Real call2market = 5.;

/*
for (int i=0; i<25; i++)
{

    Real cost = residual->amount(call2) * call2market;

    std::cout << residual->NPV() << ' ' << residual->amount(call1) << ' '
                        << residual->amount(call2) << ' ' << call1->NPV()
                        << ' ' << call2->NPV() << ' ' << - cost << ' '
                        << residual->NPV() - cost << std::endl;

    residual->add(call2,+0.05);

}

*/
Real step = 0.01;
Real precision = 0.0001;
Real lambdaOld=0;
Real lambdaNew=+0.5;

residual->add(call2,lambdaNew);
Real worstcasehedgecost=residual->NPV()-residual->amount(call2)*call2market;
```

```
    std::cout << residual->NPV() << ' ' << residual->amount(call2) << ' '
                        << call2->NPV() << ' ' << worstcasehedgecost
                        << std::endl;

    while (std::fabs(lambdaNew-lambdaOld)>precision)
    {
        lambdaOld = lambdaNew;
        lambdaNew = lambdaNew + step * (call2->NPV()-call2market);
        residual->add(call2,lambdaNew-lambdaOld);

        worstcasehedgecost=residual->NPV()-residual->amount(call2)*call2market;
        std::cout << residual->NPV() << ' ' << residual->amount(call2) << ' '
                            << call2->NPV() << ' ' << worstcasehedgecost
                            << std::endl;

    }


}
```

Appendix H

Financial vocabulary

The aim of this section is to provide a short glossary of the main financial terms.

Financial market: A financial market is a risk market in where speculators buy risk by giving an amount of money to hedgers in return of a financial security that gives them rights over a share of future cash-flows. We can distinguish primary markets where the securities are issued (the most typical would be an Initial Public Offering of shares) and secondary markets where those securities are traded.

Underlying: Magnitude on which a derivative security is dependent. Examples of magnitudes used as underlyings range from stock prices or commodities prices to indexes such as the consumer price index for inflation derivatives or physical indices for weather or catastrophe derivatives.

Derivative: A financial security whose value depends on the value of a pre-specified underlying. The market for derivatives has experienced a dramatic growth during the past years, reaching a USD600,000bn of outstanding deals by 2008 according to the Bank for International Settlements, which is 16 times the global equity market capitalisation. The technicalities of derivatives valuation have generated a lot of controversy during the past years, especially with the unfolding of the late financial crisis.

Option: Options are one of the most commonly used derivatives; an option gives the right to the holder to acquire the underlying at a given price, the strike and under certain maturity conditions. At the money options are options whose strike is equal to the price of the underlying while in the money options will be those for which the price of the underlying

is above the strike and out of the money options those for which the underlying price is under the strike price. We can also distinguish European options, which have a single maturity, American options, that can be exercised over their lifetime or Bermudan options that can only be exercised at some given dates. Options with additional features are called exotic. We can name for instance barrier options, options that are activated or deactivated when the underlying reaches a certain threshold.

Fair value: The value of financial securities should reflect the time value of money as well as the uncertainty of the outcomes related to that security. The appreciation of this uncertainty is often subjective and so is the fair value concept. When the price differs from the fair value, there is an arbitrage opportunity. In liquid markets with absence of arbitrage opportunities, the price is considered to reflect the fair value; this is the base of fair value accounting by which a financial security should be accounted for at its market price. The problem arises when the lack of liquidity makes prices uninformative.

Arbitrage: Arbitrage is the circumstance in where an agent is able to obtain a risk free profit by taking positions on traded securities. Pure arbitrage exploits price differences while pseudo-arbitrage bets on the convergence of a price to a value obtained by means of a valuation model.

Valuation models: To compute the fair value of a security, one can also proceed by elaborating a set of assumptions and translating these assumptions into a value. The tool to do this translation is a financial model.

Diffusion model: A diffusion model is a valuation model that uses as main building block a diffusion process, that is a process governed by a Brownian motion.

Volatility: Volatility is the scaling factor applied to the Brownian motions that convey the uncertainty of the underlyings in diffusion models. The volatility can be understood as a second order moment of a probability distribution. Volatility will be relevant in pricing convex instruments as linear payoffs can be easily replicated by combining basic securities.

Balance: The balance sheet is one of the three mandatory statements that companies have to submit to investors. In the balance sheet one can find the list of assets and liabilities of the company. The assets are the goods and rights owned by the company while the liabilities are its obligations towards investors or other counterparties. The methodology by which the different operations of the company are accounted for can be controversial,

especially in presence of model dependent valuations like in the case of derivatives.

P&L: The Profit & Loss account is one of the three mandatory statements that companies have to submit to investors. In this account, companies keep track of the revenues and costs over a specified period of time. The key point here is of timing, that is when an operation can be considered to originate a profit or a loss. More generally, the concept can be extrapolated to business units such as trading desks.

Cash-Flow statement: The cash flow statement is one of the three mandatory statements that companies have to submit to investors. This document keeps the record of all cash inflows and outflows and explains the transition from the beginning of the period cash to the end of period cash.

Bibliography

[ALP95] Marco Avellaneda, Arnon Levy, and Antonio Paras, *Pricing and hedging derivative securities in markets with uncertain volatilities*, Applied Mathematical Finance **2** (1995), 73–88.

[APEK+96] Marco Avellaneda, Antonio Paras, Nicole El-Karoui, Arnon Levy, Howard Savery, Nassim Taleb, and Paul Wilmott, *Managing the volatility risk of portfolios of derivative securities: The lagrangian uncertain volatility model*, Applied Mathematical Finance **3** (1996), 21–52.

[Arm06] Deborah J. Armstrong, *The quarks of object-oriented development*, Commun. ACM **49** (2006), no. 2, 123–128.

[BR96] Martin Baxter and Andrew Rennie, *Financial calculus: An introduction to derivative pricing*, Cambridge University Press, 1996.

[CGM01] Peter Carr, Helyette Geman, and Dilip Madan, *Pricing and Hedging in Incomplete Markets*, Journal of Financial Economics **62** (2001), no. 1.

[Der96] Emmanuel Derman, *Model risk*, Risk **9** (1996), no. 5.

[DW] Emmanuel Derman and Paul Wilmott, *The Financial Modeler's Manifesto*.

[Hau05] Martin Haugh, *Term structure models: Stochastic calculus*, 2005, `http://www.columbia.edu/~mh2078/TS05.html`.

[Hes93] Steven L. Heston, *A closed-form solution for options with stochastic volatility with applications to bond and currency options*, The Review of Financial Studies **6** (1993), no. 2, 327–343.

[Hul05] John Hull, *Options, futures and other derivatives*, Prentice Hall, 2005.

[HW87] John C Hull and Alan D White, *The pricing of options on assets with stochastic volatilities*, Journal of Finance **42** (1987), no. 2, 281–300.

[Met] *MetaUML Project*, `http://metauml.sourceforge.net`.

[OOP] *Object Oriented Programming*, `http://en.wikipedia.org/wiki/Object-oriented`.

[Reb98] Riccardo Rebonato, *Interest-rate option models*, John Wiley & Sons, 1998.

[Reb03] ———, *Theory and practice of model risk management*, ch. 15, Risk Books, 2003.

[UML] *UML Resource page*, `http://www.uml.org/#UML2.0`.

[Wil06] Paul Wilmott, *Paul Wilmott on quantitative finance*, Wiley, 2006.