C++ and Applications to Financial Mathematics

QUANTITATIVE FINANCE TOOLBOX

March 10, 2021

Benoit Vandevelde

Universite Gustave Eiffel
Ecole Nationale des Ponts et Chaussees
Master 2 Mathematiques et Applications

vandevelde.benoit@yahoo.com

Chapter 1

Volatility Smile and Financial Models that capture it

1.1 Black-Scholes Model and Implied Volatility Surface

1.1.1 Black-Scholes Analytical formula

Let's recall the Black-Scholes model [under a probability measure $\mathbb{P}$]:

$$\frac{dS_t}{S_t} = \mu(t)dt + \sigma dW_t^{\mathbb{P}} \tag{1.1}$$

We assume a deterministic time-dependent risk-free rate $(r(s))_{s\geq 0}$.
Under the risk-neutral measure $\mathbb{Q}$, the Black-Scholes model is written:

$$\frac{dS_t}{S_t} = r(t)dt + \sigma dW_t^{\mathbb{Q}} \tag{1.2}$$

The reason being that the numeraire associated with $\mathbb{Q}$ is the savings account $(B_t = e^{\int_0^t r(s)ds})_{t\geq 0}$, and that any discounted asset by this numeraire $(\frac{S_t}{B_t})_{t\geq 0}$ is therefore a martingale under this measure. The price of a vanilla call option on the underlying asset $(S_t)_{t\geq 0}$ for maturity T, strike K, is given by the following analytical formula under the BS model:

$$\boxed{C_{T,K}(\sigma) = C_{BS}(S_0, T, K, \sigma) = S_0 N(d_1) - K e^{-\int_0^T r(s)ds} N(d_2)} \tag{1.3}$$

where

$$d_1 = d_1(T, K, \sigma) = \frac{\ln(\frac{S_0}{K}) + (\int_0^T r(s)ds + \frac{\sigma^2 T}{2})}{\sigma\sqrt{T}}$$

$$d_2 = d_2(T, K, \sigma) = \frac{\ln(\frac{S_0}{K}) + (\int_0^T r(s)ds - \frac{\sigma^2 T}{2})}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T}$$

For information, we call the forward value F_T at maturity T the expectation under the risk-neutral measure of the asset at that maturity:

$$F_T = \mathbb{E}^{\mathbb{Q}}[S_T] = S_0 e^{\int_0^T r(s)ds}$$

1.1.2 Implied Volatility Surface

In the market, for each asset, we observe a matrix of call prices for a given set of strikes and set of maturities $\{C^{Mkt}(T_i, K_j)\}_{i\in[[1,M]],j\in[[1,N]]}$. Therefore we can reverse the implied volatility matrix on the same set of strikes and maturities. To do so, we just need to inverse the formula:

$$\boxed{\forall (i,j) \in [[1,M]] \times [[1,N]], \quad \sigma^*(T_i, K_j) = C^{-1}_{T_i,K_j}(C^{Mkt}(T_i, K_j))} \tag{1.4}$$

Then we need to interpolate/extrapolate this volatility matrix to be able to request the implied volatility at any maturity and strike.

For each maturity, the volatility smile is often interpolated/extrapolated using natural cubic splines.

The interpolation/extrapolation along maturities has different conventions, but the one we will use is linear interpolation in total variance $(\sigma^*)^2(T,K)T$ along a constant forward moneyness line $k_{F_T} = \frac{K}{F_T} = \frac{K}{S_0 e^{\int_0^T r(s)ds}}$.

1.1.3 Interpolation/Extrapolation along the strikes

For each $i \in [[1,M]]$, we call the slice $(\sigma * (T_i, K))_{K\geq 0}$ the volatility smile at maturity T_i.
As we only get a few points $\{\sigma * (T_i, K_1), \dots, \sigma * (T_i, K_N)\}$ from the market option prices, a first intuition would be to perform linear interpolation in between points.

As we will see in the next chapter, we will need the implied volatility surface to be $C^{1,2}$, meaning once-differentiable towards the maturity variable T, and twice-differentiable towards the strike variable K.

A better idea is then to perform polynomial interpolation in between points: we will use the **Natural Cubic Spline** approach.

Let's call $(x_j = K_j, y_j = \sigma^*(T_i, K_j))_{j\in[[1,N]]}$ the set of points of the smile at maturity T_i.

$\forall j \in [[1, N-1]]$, we consider the cubic polynomial function S_j defined on $[x_j, x_{j+1}]$ by:

$$S_j(x) = \alpha_j(x - x_j)^3 + \beta_j(x - x_j)^2 + \gamma_j(x - x_j) + \delta_j \tag{1.5}$$

The spline interpolation function is therefore the piecewise combination of those cubic polynomials:

$$\forall K \in [x_1, x_N], \quad \sigma * (T_i, K) = S_j(K) \quad \text{if } K \in [x_j, x_{j+1}] \tag{1.6}$$

We need to find the $4 \times (N-1)$ coefficients $\{\alpha_j, \beta_j, \gamma_j, \delta_j\}$.

Conditions at points x_j

Firstly, let's use the fact that the spline function **contains all the points** given by the market:

$$\forall j \in [[1, N-1]], \quad S_j(x_j) = y_j \text{ and } S_j(x_{j+1}) = y_{j+1} \quad \text{[2(N-1) conditions]} \tag{1.7}$$

Secondly, let's use the C^2 **property** of the spline function:

$$\forall j \in [[1, N-2]], \quad S'_j(x_{j+1}) = S'_{j+1}(x_{j+1}) \text{ and } S''_j(x_{j+1}) = S''_{j+1}(x_{j+1}) \quad \text{[2(N-2) conditions]} \tag{1.8}$$

Finally, there are 2 conditions left to make the system solvable. The most natural choice is to have **zero second order derivative** at extremities:

$$S''_1(x_1) = S''_{N-1}(x_N) = 0 \tag{1.9}$$

Solving the conditions

Denoting $\Delta x_j = x_{j+1} - x_j$, the set of equations 1.7 brings to:

$$\delta_j = y_j \tag{1.10}$$

$$\alpha_j \Delta x_j^3 + \beta_j \Delta x_j^2 + \gamma_j \Delta x_j = y_{j+1} - y_j \tag{1.11}$$

The set of equations 1.8 brings to:

$$3\alpha_j \Delta x_j^2 + 2\beta_j \Delta x_j = \gamma_{j+1} - \gamma_j \tag{1.12}$$

$$3\alpha_j \Delta x_j = \beta_{j+1} - \beta_j \tag{1.13}$$

The last set of equations 1.9 brings to:

$$\beta_1 = 0 \tag{1.14}$$

$$3\alpha_{N-1} \Delta x_{N-1} + \beta_{N-1} = 0 \tag{1.15}$$

Let's multiply equation 1.13 by Δx_j:

$$3\alpha_j \Delta x_j^2 = (\beta_{j+1} - \beta_j) \Delta x_j \tag{1.16}$$

Let's incorporate this equation in 1.12:

$$\boxed{\gamma_{j+1} - \gamma_j = (\beta_{j+1} + \beta_j) \Delta x_j} \tag{1.17}$$

We can then incorporate $\alpha_j = \frac{\beta_{j+1} - \beta_j}{3\Delta x_j}$ in 1.11 and multiply this equation by $\frac{3}{\Delta x_j}$:

$$(\beta_{j+1} + 2\beta_j) \Delta x_j + 3\gamma_j = 3\frac{y_{j+1} - y_j}{\Delta x_j} \tag{1.18}$$

Let's display the same equation for $j+1$:

$$(\beta_{j+2} + 2\beta_{j+1}) \Delta x_{j+1} + 3\gamma_{j+1} = 3\frac{y_{j+2} - y_{j+1}}{\Delta x_{j+1}} \tag{1.19}$$

Let's subtract equation 1.19 and 1.18, incorporating 1.17into it:

$$\forall j \in [[1, N-3]], \quad \boxed{\beta_{j+2}\Delta x_{j+1} + 2\beta_{j+1}(\Delta x_{j+1} + \Delta x_j) + \beta_j \Delta x_j = 3\left(\frac{y_{j+2} - y_{j+1}}{\Delta x_{j+1}} - \frac{y_{j+1} - y_j}{\Delta x_j}\right)} \tag{1.20}$$

Noticing that $\beta_1 = 0$ and incorporating $\alpha_{N-1} = -\frac{\beta_{N-1}}{3\Delta x_{N-1}}$ into equation 1.11, we deduce the following expression:

$$\boxed{\Delta x_{N-2}\beta_{N-2} + 2(\Delta x_{N-2} + \Delta x_{N-1})\beta_{N-1} = 3\left(\frac{y_N - y_{N-1}}{\Delta x_{N-1}} - \frac{y_{N-1} - y_{N-2}}{\Delta x_{N-2}}\right)} \tag{1.21}$$

Matricial expression for the β_j

We can express the (N-3) equations from 1.20 and the last equation 1.21 in an elegant matricial way:

$$\begin{pmatrix} 2(\Delta x_1 + \Delta x_2) & \Delta x_2 & 0 & 0 & 0 & 0 & 0 \\ \Delta x_2 & 2(\Delta x_2 + \Delta x_3) & \Delta x_3 & 0 & 0 & 0 & 0 \\ 0 & \dots & \dots & \dots & 0 & 0 & 0 \\ 0 & 0 & \dots & \dots & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & \dots & \dots & 0 \\ 0 & 0 & 0 & 0 & \Delta x_{N-3} & 2(\Delta x_{N-3} + \Delta x_{N-2}) & \Delta x_{N-2} \\ 0 & 0 & 0 & 0 & 0 & \Delta x_{N-2} & 2(\Delta x_{N-2} + \Delta x_{N-1}) \end{pmatrix} \begin{pmatrix} \beta_2 \\ \beta_3 \\ \dots \\ \dots \\ \dots \\ \beta_{N-2} \\ \beta_{N-1} \end{pmatrix}$$

$$= 3 \times \begin{pmatrix} \frac{y_3 - y_2}{\Delta x_2} - \frac{y_2 - y_1}{\Delta x_1} \\ \frac{y_4 - y_3}{\Delta x_3} - \frac{y_3 - y_2}{\Delta x_2} \\ \dots \\ \dots \\ \dots \\ \frac{y_{N-1} - y_{N-2}}{\Delta x_{N-2}} - \frac{y_{N-2} - y_{N-3}}{\Delta x_{N-3}} \\ \frac{y_N - y_{N-1}}{\Delta x_{N-1}} - \frac{y_{N-1} - y_{N-2}}{\Delta x_{N-2}} \end{pmatrix} \tag{1.22}$$

We denote A the $(N-2)\times(N-2)$ **tridiagonal symmetrical matrix**, Z the unknown vector $\{Z_1=\beta_2,\ldots,Z_{N-2}=\beta_{N-1}\}$ of size $N-2$ and R $=\{R_1,\ldots,R_{N-2}\}$ the right-hand side vector of the equation above.

This system is a typical application of the Thomas decomposition, given A is tridiagonal.
The algorithm will be stable due to the fact that A is strictly diagonally dominant with real positive diagonal entries, therefore A is positive definite.

Thomas algorithm: finding the β_j's

The Thomas algorithm is used to solve the system

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 & 0 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 & 0 & 0 & 0 \\ 0 & \ldots & \ldots & \ldots & 0 & 0 & 0 \\ 0 & 0 & \ldots & \ldots & \ldots & 0 & 0 \\ 0 & 0 & 0 & \ldots & \ldots & \ldots & 0 \\ 0 & 0 & 0 & 0 & a_{N-3} & b_{N-3} & c_{N-3} \\ 0 & 0 & 0 & 0 & 0 & a_{N-2} & b_{N-2} \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ \ldots \\ \ldots \\ \ldots \\ X_{N-3} \\ X_{N-2} \end{pmatrix} = \begin{pmatrix} R_1 \\ R_2 \\ \ldots \\ \ldots \\ \ldots \\ R_{N-3} \\ R_{N-2} \end{pmatrix}$$

with

$$\begin{cases} a_j = \Delta x_j & \text{for } j \in [[2, N-2]] \\ b_j = 2(\Delta x_j + \Delta x_{j+1}) & \text{for } j \in [[1, N-2]] \\ c_j = \Delta x_j & \text{for } j \in [[1, N-3]] \\ R_j = 3\left(\frac{y_{j+2}-y_{j+1}}{\Delta x_{j+1}} - \frac{y_{j+1}-y_j}{\Delta x_j}\right) & \text{for } j \in [[1, N-2]] \end{cases}$$

Here how it is done:

1. Compute some new coefficients [sweep to eliminate the a_j's]:

$$c'_j = \begin{cases} \frac{c_j}{b_j} & \text{if } j = 1 \\ \frac{c_j}{b_j - a_j c'_{j-1}} & \text{if } j \in [[2, N-3]] \end{cases}$$

and

$$R'_j = \begin{cases} \frac{R_j}{b_j} & \text{if } j = 1 \\ \frac{R_j - a_j R'_{j-1}}{b_j - a_j c'_{j-1}} & \text{if } j \in [[2, N-2]] \end{cases}$$

2. The solution is then obtained by back substitution:

$$X_{N-2} = R'_{N-2}$$

$$X_j = R'_j - c'_j X_{j+1}$$

Deducing other coefficients: α_j's and γ_j's

Now that all β_j's have been computed from the Thomas algorithm above, the rest is easier to deduce:

$$\alpha_j = \begin{cases} \alpha_j = \frac{\beta_{j+1}-\beta_j}{3\Delta x_j} & \text{if } j \in [[1, N-2]] \\ \\ \alpha_{N-1} = -\frac{\beta_{N-1}}{3\Delta x_{N-1}} & \text{if } j = N-1 \end{cases} \tag{1.23}$$

Finally:

$$\gamma_j = \frac{y_{j+1} - y_j}{\Delta x_j} - \alpha_j \Delta x_j^2 - \beta_j \Delta x_j \tag{1.24}$$

Extrapolation along the strike tails

In the regions $K < K_1$ and $K > K_N$, we make the assumption of linear extrapolation (we prolongate the zero second-order derivative from the extreme points).
We first compute the Left and Right Derivatives:

$$\begin{cases} D_L = S_1'(x_1) = \gamma_1 \\ D_R = S_{N-1}'(x_N) = 3\alpha_{N-1}\Delta x_{N-1}^2 + 2\beta_{N-1}\Delta x_{N-1} + \gamma_{N-1} \end{cases} \tag{1.25}$$

The extrapolation formula in the tail regions become:

$$\sigma^*(T_i, K) = \begin{cases} \sigma^*(T_i, K_1) + D_L \times (K - K_1) & \text{if } K < K_1 \\ \sigma^*(T_i, K_N) + D_R \times (K - K_N) & \text{if } K > K_N \end{cases} \tag{1.26}$$

1.1.4 Interpolation/Extrapolation along the maturities

The algorithm below assumes that all M smile functions $(\sigma^*(T_i, K))_{K\geq 0}$ have been computed by interpolation/extrapolation for all maturities $\{T_1, \ldots, T_M\}$ as done in the previous section.

Interpolation along maturities

The choice we are making for the interpolation along the maturities is a **linear interpolation in variance** following **constant forward moneyness** levels.

For a given maturity $T \in [T_i, T_{i+1}]$ and any strike K

1. We compute the forward moneyness level: $k_{F_T} = \frac{K}{F_T} = \frac{K}{S_0} e^{-\int_0^T r(s)ds}$
2. We extract the strikes $K^{(i)}$ and $K^{(i+1)}$ corresponding to that forward moneyness for maturities T_i and T_{i+1}:
$$K^{(i)} = k_{F_T} \times S_0 e^{\int_0^{T_i} r(s)ds}$$
$$K^{(i+1)} = k_{F_T} \times S_0 e^{\int_0^{T_{i+1}} r(s)ds}$$
3. We denote the variance quantity $v(T, k) = (\sigma^*)^2(T, k \times S_0 e^{\int_0^T r(s)ds}) \times T$.
4. We get the value $v(T, k_{F_T})$ by linear interpolation of $v(T_i, k_{F_T})$ and $v(T_{i+1}, k_{F_T})$:
$$v(T, k_{F_T}) = v(T_i, k_{F_T}) + \frac{v(T_{i+1}, k_{F_T}) - v(T_i, k_{F_T})}{T_{i+1} - T_i} \times (T - T_i)$$
5. As a summary, the quantity $\sigma^*(T, K)$ is therefore computed by the following formulae:
$$\boxed{\sigma^*(T, K) = \sqrt{\frac{1}{T} \times \left((\sigma^*)^2(T_i, K^{(i)})T_i + \frac{(\sigma^*)^2(T_{i+1}, K^{(i+1)})T_{i+1} - (\sigma^*)^2(T_i, K^{(i)})T_i}{T_{i+1} - T_i}(T - T_i) \right)}} \tag{1.27}$$
where
$$K^{(i)} = K e^{\int_0^{T_i - T} r(s)ds}$$
$$K^{(i+1)} = K e^{\int_0^{T_{i+1} - T} r(s)ds}$$
6. All the quantities above are obtained thanks to interpolation/extrapolation of all the smile functions at all maturities $\{T_i\}_{i\in[[1,M]]}$

Extrapolation along maturities

The extrapolation of the implied volatility surface outside the range of the market input maturities can be assumed to be constant, still following a same level of forward moneyness from the extreme maturities:
$$\sigma^*(T, K) = \begin{cases} \sigma^*(T_1, K^(1)) & \text{if } T < T_1 \\ \sigma^*(T_M, K^(M)) & \text{if } T > T_M \end{cases} \tag{1.28}$$
where
$$K^{(1)} = K e^{\int_0^{T_1 - T} r(s)ds}$$
$$K^{(M)} = K e^{\int_0^{T_M - T} r(s)ds}$$

1.2 Local Volatility Model

The stochastic volatility(LV) model is driven by the following SDEs [under a probability measure $\mathbb{P}$]:

$$\frac{dS_t}{S_t} = \mu(t)dt + \sigma_D(t, S_t)dW_t^{\mathbb{P}} \tag{1.29}$$

Again, it is better to reasonate from now on in the risk-neutral measure $\mathbb{Q}$:

$$\frac{dS_t}{S_t} = r(t)dt + \sigma_D(t, S_t)dW_t^{\mathbb{Q}} \tag{1.30}$$

σ_D stands for **Dupire** local volatility.

The probability density π of the stochastic process $(S_t)_{t\geq 0}$ is following the Fokker-Planck equation (see A.3):

$$\frac{\partial \pi}{\partial t} = -\frac{\partial}{\partial s}(r(t)s\pi) + \frac{1}{2}\frac{\partial^2}{\partial s^2}(\sigma_D(t,s)^2 s^2 \pi) \tag{1.31}$$

Recalling the link between the Call option price $C(T,K)$ with respect to the underlying S and the probability density π:

$$\begin{aligned} C(T,K) &= e^{-\int_0^T r(s)ds}\mathbb{E}^{\mathbb{Q}}[(S_T - K)^+] \\ &= e^{-\int_0^T r(s)ds}\int_{S=-\infty}^{+\infty}(S-K)^+\pi(T,S)dS \\ &= e^{-\int_0^T r(s)ds}\int_{S=K}^{+\infty}(S-K)\pi(T,S)dS \end{aligned} \tag{1.32}$$

Let's notice the following reformulation:

$$C(T,K) = e^{-\int_0^T r(s)ds} \times \left(\int_{S=K}^{+\infty} S\pi(T,S)dS - K\int_{S=K}^{+\infty}\pi(T,S)dS\right) \tag{1.33}$$

1.2.1 First order derivative of call option price towards strike

We can compute the derivative of the call price towards the strike K:

$$\begin{aligned} \frac{\partial C}{\partial K}(T,K) &= \frac{\partial}{\partial K}(e^{-\int_0^T r(s)ds}\int_{S=K}^{+\infty}(S-K)\pi(T,S)dS) \\ &= e^{-\int_0^T r(s)ds} \times (-(K-K)\pi(T,K) - \int_{S=K}^{+\infty}\pi(T,S)dS) \\ &= -e^{-\int_0^T r(s)ds}\int_{S=K}^{+\infty}\pi(T,S)dS \end{aligned} \tag{1.34}$$

1.2.2 Second order derivative of call option price towards strike

We can compute the second order derivative of the call price towards the strike K:

$$\begin{aligned} \frac{\partial^2 C}{\partial K^2}(T,K) &= \frac{\partial}{\partial K}(-e^{-\int_0^T r(s)ds}\int_{S=K}^{+\infty}\pi(T,S)dS) \\ &= e^{-\int_0^T r(s)ds} \times \pi(T,K) \end{aligned} \tag{1.35}$$

1.2.3 First order derivative of call option price towards maturity

We can compute the derivative of the call price towards the maturity T:

$$\begin{aligned}
\frac{\partial C}{\partial T}(T,K) &= \frac{\partial}{\partial T}(e^{-\int_0^T r(s)ds}\int_{S=K}^{+\infty}(S-K)\pi(T,S)dS) \\
&= -r(T)e^{-\int_0^T r(s)ds}\int_{S=K}^{+\infty}(S-K)\pi(T,S)dS + e^{-\int_0^T r(s)ds}\int_{S=K}^{+\infty}(S-K)\frac{\partial \pi}{\partial T}(T,S)dS \\
&= -r(T)C(T,K) + e^{-\int_0^T r(s)ds}\int_{S=K}^{+\infty}(S-K)(-\frac{\partial}{\partial S}(r(T)S\pi(T,S)) + \frac{1}{2}\frac{\partial^2}{\partial S^2}(\sigma_D^2(T,S)S^2\pi(T,S))dS \\
&= -r(T)C(T,K) + e^{-\int_0^T r(s)ds}\times \\
&(-\underbrace{\int_{S=K}^{+\infty}(S-K)\frac{\partial}{\partial S}(r(T)S\pi(T,S))dS}_{F_A} + \frac{1}{2}\underbrace{\int_{S=K}^{+\infty}(S-K)\frac{\partial^2}{\partial S^2}(\sigma_D^2(T,S)S^2\pi(T,S))dS}_{F_B})
\end{aligned} \tag{1.36}$$

Let's now focus on both terms F_A and F_B:

$$\begin{aligned}
F_A &= r(T)\int_{S=K}^{+\infty}(S-K)\frac{\partial}{\partial S}(S\pi(T,S)dS \\
&= r(T)\times(\underbrace{[(S-K)S\pi(T,S)]_K^{+\infty}}_{=0} - \int_{S=K}^{+\infty}S\pi(T,S)dS) \\
&= -r(T)\int_{S=K}^{+\infty}S\pi(T,S))dS \\
&= -r(T)\times(e^{\int_0^T r(s)ds}C(T,K) - K\int_{S=K}^{+\infty}\pi(T,S)dS) \text{ [cf } \textbf{??} \text{]} \\
&= -r(T)\times(e^{\int_0^T r(s)ds}C(T,K) - Ke^{\int_0^T r(s)ds}\frac{\partial C}{\partial K}(T,K)) \quad \text{[cf 1.34]} \\
&= -r(T)e^{\int_0^T r(s)ds}\times(C(T,K) - K\frac{\partial C}{\partial K}(T,K))
\end{aligned} \tag{1.37}$$

$$\begin{aligned}
F_B &= \int_{S=K}^{+\infty}(S-K)\frac{\partial^2}{\partial S^2}(\sigma_D^2(T,S)S^2\pi(T,S))dS \\
&= \underbrace{[(S-K)\frac{\partial}{\partial S}(\sigma_D^2(T,S)S^2\pi(T,S))]_K^{+\infty}}_{=0} - \int_{S=K}^{+\infty}\frac{\partial}{\partial S}(\sigma_D^2(T,S)S^2\pi(T,S))dS \\
&= -[\sigma_D^2(T,S)S^2\pi(T,S)]_K^{+\infty} \\
&= \sigma_D^2(T,K)K^2\pi(T,K) \\
&= \sigma_D^2(T,K)\ K^2\ e^{\int_0^T r(s)ds}\ \frac{\partial^2 C}{\partial K^2}(T,K) \text{[cf 1.35]}
\end{aligned} \tag{1.38}$$

1.2.4 Dupire formulae in function of call option prices and derivatives

Finally, let's have a final expression for the derivative of the call price towards the maturity:

$$\frac{\partial C}{\partial T}(T,K) = -r(T)C(T,K) + r(T)(C(T,K) - K\frac{\partial C}{\partial K}(T,K)) + \frac{1}{2}\sigma_D^2(T,K)K^2\frac{\partial^2 C}{\partial K^2}(T,K) \tag{1.39}$$

We get then the **Dupire calibration formulae** linking the local volatility of our LV model and the Call option prices and derivatives towards maturity and strike:

$$\boxed{\sigma_D^2(T,K) = \frac{\frac{\partial C}{\partial T}(T,K) + r(T)K\frac{\partial C}{\partial K}(T,K)}{\frac{1}{2}K^2\frac{\partial^2 C}{\partial K^2}(T,K)}} \tag{1.40}$$

1.2.5 Dupire formulae in function of BS implied volatility

Given the Black Scholes implied volatility surface $(\sigma^*(t,k))_{t\geq 0, k\geq 0}$, obtained from interpolation/extrapolation of a finite number of implied volatilities obtained from inversing market vanilla options prices, we have the following:

$$C(T,K) = C_{T,K}(\sigma^*(T,K)) = C_{BS}(S_0,T,K,\sigma^*(T,K)) \tag{1.41}$$

Hence:

$$\frac{\partial C}{\partial T}(T,K) = \frac{\partial C_{BS}}{\partial T} + \frac{\partial \sigma^*}{\partial T}\ \frac{\partial C_{BS}}{\partial \sigma^*} \tag{1.42}$$

$$\frac{\partial C}{\partial K}(T,K) = \frac{\partial C_{BS}}{\partial K} + \frac{\partial \sigma^*}{\partial K}\ \frac{\partial C_{BS}}{\partial \sigma^*} \tag{1.43}$$

$$\frac{\partial^2 C}{\partial K^2}(T,K) = \frac{\partial^2 C_{BS}}{\partial K^2} + \frac{\partial \sigma^*}{\partial K}\ \frac{\partial^2 C_{BS}}{\partial K \partial \sigma^*} + \frac{\partial^2 \sigma^*}{\partial K^2}\ \frac{\partial C_{BS}}{\partial \sigma^*} + \frac{\partial \sigma^*}{\partial K}\left(\frac{\partial^2 C_{BS}}{\partial \sigma^* \partial K} + \frac{\partial \sigma^*}{\partial K}\ \frac{\partial^2 C_{BS}}{\partial (\sigma^*)^2}\right) \tag{1.44}$$

Now we want to express $\frac{\partial C_{BS}}{\partial T}$, $\frac{\partial C_{BS}}{\partial K}$ and $\frac{\partial^2 C_{BS}}{\partial K^2}$ as a function of σ^*, and therefore express the Dupire volatility surface as a function of the BS implied volatility surface and its derivatives towards maturity and strikes.

First, let's compute:

$$\frac{\partial d_1}{\partial T}(T,K,\sigma^*) = \frac{r(T) + \frac{1}{2}(\sigma^*)^2 - \frac{d_1\sigma^*}{2\sqrt{T}}}{\sigma^*\sqrt{T}} \tag{1.45}$$

$$\frac{\partial d_2}{\partial T}(T,K,\sigma^*) = \frac{r(T) - \frac{1}{2}(\sigma^*)^2 - \frac{d_2\sigma^*}{2\sqrt{T}}}{\sigma^*\sqrt{T}} = \frac{\partial d_1}{\partial T}(T,K,\sigma^*) - \frac{\sigma^*}{2\sqrt{T}} \tag{1.46}$$

$$\frac{\partial d_1}{\partial K}(T,K,\sigma^*) = \frac{\partial d_2}{\partial K}(T,K,\sigma^*) = -\frac{1}{K\sigma^*\sqrt{T}} \tag{1.47}$$

$$\frac{\partial d_1}{\partial \sigma^*}(T,K,\sigma^*) = -\frac{d_1}{\sigma^*} + \sqrt{T} \tag{1.48}$$

$$\frac{\partial d_2}{\partial \sigma^*}(T,K,\sigma^*) = -\frac{d_2}{\sigma^*} - \sqrt{T} \tag{1.49}$$

Therefore, we get the following:

$$\frac{\partial C_{BS}}{\partial T} = Kr(T)e^{-\int_0^T r(s)ds}N(d_2) + \frac{\sigma^*}{2\sqrt{T}}Ke^{-\int_0^T r(s)ds}n(d_2) \tag{1.50}$$

$$\frac{\partial C_{BS}}{\partial K} = -e^{-\int_0^T r(s)ds}N(d_2) \tag{1.51}$$

$$\frac{\partial^2 C_{BS}}{\partial K^2} = e^{-\int_0^T r(s)ds}\frac{n(d_2)}{K\sigma^*\sqrt{T}} \tag{1.52}$$

$$\frac{\partial C_{BS}}{\partial \sigma^*} = Ke^{-\int_0^T r(s)ds}\sqrt{T}n(d_2) \tag{1.53}$$

$$\frac{\partial^2 C_{BS}}{\partial K \partial \sigma^*} = e^{-\int_0^T r(s)ds}\sqrt{T}n(d_2) + e^{-\int_0^T r(s)ds}\frac{d_2 n(d_2)}{\sigma^*} = e^{-\int_0^T r(s)ds}\frac{d_1 n(d_2)}{\sigma^*} \tag{1.54}$$

$$\frac{\partial^2 C_{BS}}{\partial (\sigma^*)^2} = Ke^{-\int_0^T r(s)ds}\sqrt{T}\ \frac{d_1 d_2}{(\sigma^*)}n(d_2) \tag{1.55}$$

So now we can express the following derivatives:

$$\frac{\partial C}{\partial T}(T,K) = Ke^{-\int_0^T r(s)ds}\left(r(T)N(d_2) + \frac{\sigma^*}{2\sqrt{T}}n(d_2) + \frac{\partial \sigma^*}{\partial T}\ \sqrt{T}n(d_2)\right) \tag{1.56}$$

$$\frac{\partial C}{\partial K}(T,K) = e^{-\int_0^T r(s)ds}\left(-N(d_2) + \frac{\partial \sigma^*}{\partial K}\ K\sqrt{T}n(d_2)\right) \tag{1.57}$$

$$\begin{aligned}\frac{\partial^2 C}{\partial K^2}(T,K) = e^{-\int_0^T r(s)ds}\ \times \\ \left(\frac{n(d_2)}{K\sigma^*\sqrt{T}} + \frac{\partial \sigma^*}{\partial K}\ \frac{d_1 n(d_2)}{\sigma^*} + \frac{\partial^2 \sigma^*}{\partial K^2}\ K\sqrt{T}n(d_2) + \frac{\partial \sigma^*}{\partial K}\left(\frac{d_1 n(d_2)}{\sigma^*} + \frac{\partial \sigma^*}{\partial K}\ K\sqrt{T}\ \frac{d_1 d_2}{(\sigma^*)}n(d_2)\right)\right)\end{aligned} \tag{1.58}$$

By factorizing by the term $e^{-\int_0^T r(s)ds}\frac{Kn(d_2)}{2\sigma^*\sqrt{T}}$, the final expression for linking the Dupire local volatility to the implied BS volatility at any maturity T and strike K becomes:

$$\boxed{\sigma_D(T,K)^2 = \sigma^*(T,K)^2\ \frac{1 + \frac{2T}{\sigma^*(T,K)}(\frac{\partial \sigma^*}{\partial T}(T,K) + r(T)\frac{\partial \sigma^*}{\partial K}(T,K))}{1 + 2d_1(K\frac{\partial \sigma^*}{\partial K}(T,K)\sqrt{T}) + d_1 d_2(K\frac{\partial \sigma^*}{\partial K}(T,K)\sqrt{T})^2 + (K\frac{\partial^2 \sigma^*}{\partial K^2}(T,K)\sqrt{T})(K\sigma^*(T,K)\sqrt{T})}} \tag{1.59}$$

1.3 Stochastic Volatility Model

The stochastic volatility(SV) model is driven by the following SDEs under the risk-neutral measure:

$$\begin{cases} \frac{dS_t}{S_t} = r(t)dt + \Psi(V_t)dW_t^S \\ dV_t = a_v(t,V_t)dt + b_v(t,V_t)dW_t^V \\ d < W_t^S, W_t^V >= \rho dt \end{cases} \tag{1.60}$$

1.3.1 Heston model

The Heston model is determined by the following:

- $\Psi(V) = \sqrt{V}$
- $a_v(t,V) = \kappa(\theta - V)$
- $b_v(t,V) = \sigma_v\sqrt{V}$

We can then re-express the Heston model SDE under the risk-neutral measure:

$$\begin{cases} \frac{dS_t}{S_t} = r(t)dt + \sqrt{V_t}dW_t^S \\ dV_t = \kappa(\theta - V_t)dt + \sigma_v\sqrt{V_t}dW_t^V \\ d < W_t^S, W_t^V >= \rho dt \end{cases} \tag{1.61}$$

There is an analytical formula to price European vanilla options [calls and puts], as described in the following sections.
This will allow us to implement a calibration procedure based on Least Squares.

1.3.2 Analytical formula for vanilla call options under Heston model

Decomposition of the call option price

The price of a vanilla call option with maturity T and strike K is decomposed by:

$$\begin{aligned} C(S_0,V_0) &= e^{-\int_0^T r(s)ds}\mathbb{E}^{\mathbb{Q}}[(S_T - K)\mathbb{I}_{S_T \geq K}] \\ &= S_0\ \mathbb{E}^{\mathbb{Q}}[e^{-\int_0^T r(s)ds}\frac{S_T}{S_0}\mathbb{I}_{S_T \geq K}] - Ke^{-\int_0^T r(s)ds}\mathbb{E}^{\mathbb{Q}}[\mathbb{I}_{S_T \geq K}] \end{aligned} \tag{1.62}$$

Change of numeraire

We define $\mathbb{Q}^S$ is the measure associated with the numeraire $(S_t)_{t\in[0,T]}$.
The Radon-Nikodym derivative of $\mathbb{Q}^S$ with regards to $\mathbb{Q}$ is expressed by:

$$\frac{d\mathbb{Q}^S}{d\mathbb{Q}}|_{\mathcal{F}_T} = e^{-\int_0^T r(s)ds}\frac{S_T}{S_0} \tag{1.63}$$

The interesting equation follows:

$$\begin{aligned} C(S_0,V_0) &= S_0\ \mathbb{E}^{\mathbb{Q}^S}[\frac{d\mathbb{Q}}{d\mathbb{Q}^S}|_{\mathcal{F}_T}e^{-\int_0^T r(s)ds}\frac{S_T}{S_0}\mathbb{I}_{S_T \geq K}] - Ke^{-\int_0^T r(s)ds}\mathbb{E}^{\mathbb{Q}}[\mathbb{I}_{S_T \geq K}] \\ &= S_0\ \underbrace{\mathbb{E}^{\mathbb{Q}^S}[\mathbb{I}_{S_T \geq K}]}_{P_1} - Ke^{-\int_0^T r(s)ds}\underbrace{\mathbb{E}^{\mathbb{Q}}[\mathbb{I}_{S_T \geq K}]}_{P_2} \end{aligned} \tag{1.64}$$

We are now interested in computing the terms P_1 and P_2.

Dynamics of $(S_t, V_t)_{t\in[0,T]}$ under$\mathbb{Q}^S$

According to Girsanov theorem, since we have:

$$\frac{d\mathbb{Q}^S}{d\mathbb{Q}}|_{\mathcal{F}_T} = L_T = e^{\int_0^T \sqrt{V_s} dW_s^S - \frac{1}{2}\int_0^T V_s ds} \tag{1.65}$$

The process $(L_t)_{t\in[0,T]}$ is a $\mathbb{Q}$-martingale and therefore under the measure $\mathbb{Q}^S$, the following process defined by:

$$(\tilde{W}_t^S = W_t^S - \int_0^t \sqrt{V_s} ds)_{t\in[0,T]} \tag{1.66}$$

is a $\mathbb{Q}^S$-martingale.
The correlation property between brownians $d < W_t^S, W_t^V >= \rho dt$ can be translated into:

$$dW_t^V = \rho\ dW_t^S + \sqrt{1-\rho^2} dW_t^\perp = \rho\ d\tilde{W}_t^S + \rho\sqrt{V_t} dt + \sqrt{1-\rho^2} dW_t^\perp \tag{1.67}$$

where $(W_t^\perp)_{t\in[0,T]}$ is a $\mathbb{Q}$ - Brownian motion, independent from $(W_t^S)_{t\in[0,T]}$.
We define then the $\mathbb{Q}^S$ - Brownian motion $(\tilde{W}_t^V)_{t\in[0,T]}$ by:

$$d\tilde{W}_t^V = \rho\ d\tilde{W}_t^S + \sqrt{1-\rho^2} dW_t^\perp \tag{1.68}$$

Under the measure $\mathbb{Q}^S$, the dynamics of $(S_t, V_t)_{t\in[0,T]}$ reads:

$$\begin{cases} \frac{dS_t}{S_t} = (r(t) + V_t)dt + \sqrt{V_t} d\tilde{W}_t^S \\ dV_t = (\kappa\theta - \kappa V_t - \rho\sigma_v V_t)dt + \sigma_v\sqrt{V_t} d\tilde{W}_t^V \\ d < \tilde{W}_t^S, \tilde{W}_t^V >= \rho dt \end{cases} \tag{1.69}$$

Feynman-Kac PDEs to solve P_1 and P_2

According to Feynman-Kac theorem [see **??**] we can express each price $P_i (i \in \{1, 2\})$ as the solution of the following PDE:

$$\begin{cases} \frac{\partial \hat{U}^{(i)}}{\partial t} + \mu_S^{(i)} S \frac{\partial \hat{U}^{(i)}}{\partial S} + \mu_V^{(i)} \frac{\partial \hat{U}^{(i)}}{\partial V} + \frac{1}{2} V^2 S^2 \frac{\partial^2 \hat{U}^{(i)}}{\partial S^2} + \frac{1}{2}\sigma_v^2 V \frac{\partial^2 \hat{U}^{(i)}}{\partial V^2} + \rho\sigma_v SV \frac{\partial^2 \hat{U}^{(i)}}{\partial S \partial V} = 0 \\ \\ \hat{U}^{(i)}(T, S, V) = \mathbb{I}_{S\geq K} \end{cases} \tag{1.70}$$

with

$$\begin{cases} \mu_S^{(1)} = r(t) \\ \mu_S^{(2)} = r(t) + V \\ \mu_V^{(1)} = \kappa\theta - \kappa V \\ \mu_V^{(2)} = \kappa\theta - \kappa V - \rho\sigma_v V \end{cases}$$

Solutions of PDE with contingent payoff $\mathbb{I}_{S_T \geq K}$ for both measures $\mathbb{Q}$ and $\mathbb{Q}^S$

Using the following transformations:

$$\begin{cases} \tau = T - t \\ x = \ln(S) \\ r = r(t) \end{cases}$$

The PDE is converted into the following form:

$$\begin{cases} \frac{\partial \hat{U}^{(i)}}{\partial \tau} = \mu_x^{(i)} \frac{\partial \hat{U}^{(i)}}{\partial x} + \mu_V^{(i)} \frac{\partial \hat{U}^{(i)}}{\partial V} + \frac{1}{2} V^2 \frac{\partial^2 \hat{U}^{(i)}}{\partial x^2} + \frac{1}{2} \sigma_v^2 V \frac{\partial^2 \hat{U}^{(i)}}{\partial V^2} + \rho \sigma_v V \frac{\partial^2 \hat{U}^{(i)}}{\partial x \partial V} \\ \\ \hat{U}^{(i)}(0, x, V) = \mathbb{I}_{e^x \geq K} \end{cases} \tag{1.71}$$

with

$$\begin{cases} \mu_x^{(1)} = r - \frac{1}{2} V \\ \mu_x^{(2)} = r + \frac{1}{2} V \\ \mu_V^{(1)} = \kappa\theta - \kappa V \\ \mu_V^{(2)} = \kappa\theta - \kappa V - \rho \sigma_v V \end{cases}$$

Applying the Fourier transform on this equation with respect to the variable x, we obtain the following problem:

$$\begin{cases} \frac{\partial \tilde{U}^{(i)}}{\partial \tau} = \frac{1}{2} \sigma_v^2 V \frac{\partial^2 \tilde{U}^{(i)}}{\partial V^2} + (\kappa\theta + (\rho \sigma_v y_i - \kappa) V) \frac{\partial \tilde{U}^{(i)}}{\partial V} + (r j\omega - \frac{1}{2}(u_i j\omega + \omega^2) V) \tilde{U}^{(i)} \\ \\ \tilde{U}^{(i)}(0, \omega, V) = \mathcal{F}[H(x - \ln(K))] \end{cases} \tag{1.72}$$

with

$$H(u) = \mathbb{I}_{u \geq 0}$$

$$\begin{cases} u_1 = -1 \\ u_2 = 1 \\ y_1 = j\omega - 1 \\ y_2 = j\omega \end{cases}$$

According to Heston's paper, the solution of the above PDE can be assumed to be an affine exponential function of v of the following form:

$$\tilde{U}^{(i)}(\tau, \omega, V) = e^{C^{(i)}(\tau, \omega) + D^{(i)}(\tau, \omega) V} \; \tilde{U}(0, \omega, V) \tag{1.73}$$

Substituting this form into the PDE, since the variable V can take any value in $\mathbb{R}_+$, it leads to 2 ordinary differential equations (ODE):

$$\begin{cases} \frac{\partial D^{(i)}}{\partial \tau} = \frac{1}{2} \sigma_v^2 (D^{(i)})^2 + (\rho \sigma_v y_i - \kappa) D^{(i)} - \frac{1}{2}(u_i j\omega + \omega^2) \\ \\ \frac{\partial C^{(i)}}{\partial \tau} = \kappa\theta D^{(i)} + r j\omega \end{cases} \tag{1.74}$$

With initial conditions:

$$\begin{cases} C^{(i)}(0, \omega) = 0 \\ \\ D^{(i)}(0, \omega) = 0 \end{cases}$$

First ODE

The first ODE is a so-called Riccati equation.
To solve it, we first look for a constant solution D_0 for D:

$$\frac{1}{2}\sigma_v^2 D_0^2 + (\rho\sigma_v y_i - \kappa)D_0 - \frac{1}{2}(u_i j\omega + \omega^2) = 0 \tag{1.75}$$

We define a, Δ and b such as:

$$\begin{cases} a = \kappa - \rho\sigma_v y_i \\ \\ \Delta = a^2 + \sigma_v^2\ (u_i j\omega + \omega^2) \\ \\ b = \sqrt{\Delta} \end{cases}$$

In that case, solving this famous quadratic equation gives the following solution for D_0:

$$\boxed{D_0 = \frac{a-b}{\sigma_v^2}} \tag{1.76}$$

Then denoting $\tilde{D} = D - D_0$, we get the following equation:

$$\frac{\partial\tilde{D}}{\partial\tau} = \frac{1}{2}\sigma_v^2\tilde{D}^2 - b\tilde{D} \tag{1.77}$$

Dividing by $\tilde{D}^2$, we get the following equation:

$$\frac{\partial(\frac{1}{\tilde{D}})}{\partial\tau} + \frac{1}{2}\sigma_v^2 - b\frac{1}{\tilde{D}} = 0 \tag{1.78}$$

This equation can be rewritten the following way:

$$\frac{\partial(\frac{1}{\tilde{D}} - \frac{1}{2b}\sigma_v^2)}{\partial\tau} - b(\frac{1}{\tilde{D}} - \frac{1}{2b}\sigma_v^2) = 0 \tag{1.79}$$

The solution is of the form:

$$\frac{1}{\tilde{D}} - \frac{1}{2b}\sigma_v^2 = Ke^{b\tau} \tag{1.80}$$

Taking $\tau = 0$, it gives:

$$K = -(\frac{1}{2b}\sigma_v^2 + \frac{1}{D_0}) \tag{1.81}$$

So that the general expression for D is therefore:

$$\begin{aligned} D(\tau,\omega) &= D_0 + \frac{1}{-(\frac{1}{2b}\sigma_v^2 + \frac{1}{D_0})e^{b\tau} + \frac{1}{2b}\sigma_v^2} \\ &= D_0\ \left(1 - \frac{2b}{(\sigma_v^2 D_0 + 2b)e^{b\tau} - \sigma_v^2 D_0}\right) \\ &= D_0\ (a+b)\frac{e^{b\tau}-1}{(a+b)e^{b\tau} - (a-b)} \end{aligned} \tag{1.82}$$

$$\boxed{D^{(i)}(\tau,\omega) = \frac{a-b}{\sigma_v^2}\frac{1-e^{-b\tau}}{1-ge^{-b\tau}}} \tag{1.83}$$

Where:

$$g = \frac{a-b}{a+b}$$

Second ODE

$$C(\tau,\omega) = j\omega \int_0^\tau r(T-s)ds + \kappa\theta \frac{a-b}{\sigma_v^2} \int_0^\tau \frac{1-e^{-bu}}{1-ge^{-bu}} du \tag{1.84}$$

Noticing the following:

$$\begin{aligned} \frac{1-e^{-bu}}{1-ge^{-bu}} &= 1 - \frac{g-1}{b} \frac{-gbe^{bu}}{1-ge^{bu}} \\ &= 1 - \frac{2}{a-b} \frac{gbe^{-bu}}{1-ge^{-bu}} \end{aligned} \tag{1.85}$$

We finally have:

$$\boxed{C^{(i)}(\tau,\omega) = j\omega \int_0^\tau r(T-s)ds + \frac{\kappa\theta}{\sigma_v^2} \left[(a-b)\tau - 2\ln(\frac{1-ge^{-b\tau}}{1-g}) \right]} \tag{1.86}$$

Solution U

The solution $\hat{U}^{(i)}$ we are looking for in our problem verifies:

$$\hat{U}^{(i)}(\tau,x,V) = \mathcal{F}^{-1}[e^{C^{(i)}(\tau,\omega)+D^{(i)}(\tau,\omega)V} \mathcal{F}[H(x-\ln(K))]] \tag{1.87}$$

With the function H being:

$$H(u) = \mathbb{I}_{u>=0} \tag{1.88}$$

The Fourier transform of the unit step function H is known and is given by:

$$\mathcal{F}[H(u)](\omega) = \int_{x=-\infty}^{+\infty} e^{-j\omega u} \mathbb{I}_{u\geq 0} du = \frac{1}{j\omega} + \pi\delta(\omega) \tag{1.89}$$

By translation property of the Fourier transform, we get the following:

$$\mathcal{F}[H(x-\ln(K))](\omega) = e^{-j\omega\ln(K)} \mathcal{F}[H(x)](\omega) = \frac{e^{-j\omega\ln(K)}}{j\omega} + e^{-j\omega\ln(K)}\pi\delta(\omega) \tag{1.90}$$

The inverse Fourier transform for any function $g(\omega)$ is expressed by:

$$\mathcal{F}^{-1}[g](x) = \frac{1}{2\pi} \int_{\omega=-\infty}^{+\infty} e^{j\omega x} g(\omega) d\omega \tag{1.91}$$

Therefore we apply the formula on 1.87:

$$\hat{U}(\tau,x,V) = \frac{1}{2\pi} \int_{\omega=-\infty}^{+\infty} \frac{\phi_i(\tau,x,V,\omega)e^{-j\omega\ln(K)}}{j\omega} d\omega + \frac{\pi}{2\pi} \int_{\omega=-\infty}^{+\infty} \phi_i(\tau,x,V,\omega)e^{-j\omega\ln(K)}\delta(\omega)d\omega \tag{1.92}$$

with

$$\phi_i(\tau,x,V,\omega) = e^{C^{(i)}(\tau,\omega)+D^{(i)}(\tau,\omega)V+j\omega x}$$

We can check that :

$$\begin{cases} C^{(i)}(\tau,0) = 0 \\ \\ D^{(i)}(\tau,0) = 0 \end{cases}$$

Therefore the second term of the above equation is reduced to:

$$\frac{\pi}{2\pi}\int_{\omega=-\infty}^{+\infty}\phi_i(\tau,x,V,\omega)e^{-j\omega\ln(K)}\delta(\omega)d\omega=\frac{1}{2}\phi_i(\tau,x,V,0)e^{-j\times 0\times\ln(K)}=\frac{1}{2} \tag{1.93}$$

Since the first term has to return a real number, we can then take the real part of this complex integral and compute it as the integral of the real part:

$$\hat{U}(\tau,x,V)=\frac{1}{2}+\frac{1}{2\pi}\int_{\omega=-\infty}^{+\infty}Re\Big[\frac{\phi_i(\tau,x,V,\omega)e^{-j\omega\ln(K)}}{j\omega}\Big]d\omega \tag{1.94}$$

Finally, the two quantities P_1 and P_2 are given by:

$$P_i=\hat{U}(T,\ln(S_0),V_0)=\frac{1}{2}+\frac{1}{2\pi}\int_{\omega=-\infty}^{+\infty}Re\Big[\frac{\phi_i(T,\ln(S_0),V_0,\omega)e^{-j\omega\ln(K)}}{j\omega}\Big]d\omega \tag{1.95}$$

1.3.3 Calibration of the Heston model

This section is up to your research.
As stochastic volatility models do not fit perfectly the option prices [calibration instruments], the next section will introduce an extension of those models that allow perfect calibration.

1.3.4 Efficient Monte-Carlo simulation of the Heston model

Law of the variance V_T conditionally to V_t

Let's denote $m(t,T)$ and $s^2(t,T)$ $[t<T]$ the respective expectation and variance of the variance process at time T V_T conditionally on its value at a previous time V_t:

$$\begin{cases} \mathbb{E}[V_T|V_t] = m(t,T) = \theta + (V_t - \theta)e^{-\kappa(T-t)} \\ \\ \mathbb{V}ar[V_T|V_t] = s^2(t,T) = \frac{V_t\sigma_v^2 e^{-\kappa(T-t)}}{\kappa}(1-e^{-\kappa(T-t)}) + \frac{\theta\sigma_v^2}{2\kappa}(1-e^{-\kappa(T-t)})^2 \end{cases} \tag{1.96}$$

Broadie-Kaya scheme for log-spot process

Let's denote $(X_t)_{t\in[0,T]} = (\ln(S_t))_{t\in[0,T]}$ the log-spot process.

The SDE followed by the process $(X_t, V_t)_{t\in[0,T]}$ is then:

$$\begin{cases} dX_t = (r(t) - \frac{1}{2}V_t)dt + \sqrt{V_t}dW_t^S \\ dV_t = \kappa(\theta - V_t)dt + \sigma_v\sqrt{V_t}dW_t^V \\ d < W_t^S, W_t^V >= \rho dt \end{cases} \tag{1.97}$$

To simulate the log-spot process from $X_t = \hat{X}_t$ to $\hat{X}_{t+\Delta_t}$, let's notice the following on the Variance process:

$$V_{t+\Delta_t} = V_t + \kappa\int_{u=t}^{t+\Delta_t}(\theta - V_u)du + \sigma_v\int_{u=t}^{t+\Delta_t}\sqrt{V_u}dW_u^V \tag{1.98}$$

Therefore:

$$\boxed{\int_{u=t}^{t+\Delta_t}\sqrt{V_u}dW_u^V = \frac{1}{\sigma_v}\Big[V_{t+\Delta_t} - V_t - \kappa\theta\Delta_t + \kappa\int_{u=t}^{t+\Delta_t}V_u du\Big]} \tag{1.99}$$

Let's come back to the log-spot process:

$$X_{t+\Delta_t} = X_t + \int_{u=t}^{t+\Delta_t}(r(u) - \frac{1}{2}V_u)du + \int_{u=t}^{t+\Delta_t}\sqrt{V_u}dW_u^S \tag{1.100}$$

Now we need to express the brownian increment dW_u^S as a function of dW_u^V, using their instant correlation ρ:

$$dW_u^S = \rho\, dW_u^V + \sqrt{1-\rho^2}\, dW_u^\perp \tag{1.101}$$

where $(W_u^\perp)$ is a Brownian-motion independent from (W_u^V).

Therefore, we decompose the following term:

$$\begin{aligned} \int_{u=t}^{t+\Delta_t}\sqrt{V_u}dW_u^S &= \rho\int_{u=t}^{t+\Delta_t}\sqrt{V_u}dW_u^V + \sqrt{1-\rho^2}\int_{u=t}^{t+\Delta_t}\sqrt{V_u}dW_u^\perp \\ &= \frac{\rho}{\sigma_v}\Big[V_{t+\Delta_t} - V_t - \kappa\theta\Delta_t + \kappa\int_{u=t}^{t+\Delta_t}V_u du\Big] + \sqrt{1-\rho^2}\int_{u=t}^{t+\Delta_t}\sqrt{V_u}dW_u^\perp \end{aligned} \tag{1.102}$$

The Broadie-Kaya scheme reads then:

$$X_{t+\Delta_t} = X_t + \int_{u=t}^{t+\Delta_t} r(u)du + \frac{\rho}{\sigma_v}\left[V_{t+\Delta_t} - V_t - \kappa\theta\Delta_t\right] + \left(\frac{\kappa\rho}{\sigma_v} - \frac{1}{2}\right)\int_{u=t}^{t+\Delta_t} V_u du + \sqrt{1-\rho^2}\int_{u=t}^{t+\Delta_t}\sqrt{V_u}dW_u^{\perp} \tag{1.103}$$

The integral term $\int_{u=t}^{t+\Delta_t} V_u du$ can be approximated by the following form:

$$\begin{cases} \int_{u=t}^{t+\Delta_t} V_u du \approx \Delta_t(\gamma_1 V_t + \gamma_2 V_{t+\Delta_t}) \\ \gamma_1 + \gamma_2 = 1 \end{cases} \tag{1.104}$$

In the following, we will always choose $\gamma_1 = \gamma_2 = \frac{1}{2}$ [arithmetic average or trapezes method].

Given simulated values for the log-spot process $X_t = \hat{X}_t$ and for the Variance process $V_t = \hat{V}_t$ and $V_{t+\Delta_t} = \hat{V}_{t+\Delta_t}$, and approximating the integral term as we did above, we can notice that $X_{t+\Delta_t}$ follows the Normal Law:

$$X_{t+\Delta_t} \sim N\left(\mu_X^{(t+\Delta_t)}; (\sigma_X^{(t+\Delta_t)})^2\right) \tag{1.105}$$

where

$$\begin{cases} \mu_X^{(t+\Delta_t)} = \hat{X}_t + \int_{u=t}^{t+\Delta_t} r(u)du + \frac{\rho}{\sigma_v}\left[\hat{V}_{t+\Delta_t} - \hat{V}_t - \kappa\theta\Delta_t\right] + \left(\frac{\kappa\rho}{\sigma_v} - \frac{1}{2}\right)\Delta_t(\gamma_1\hat{V}_t + \gamma_2\hat{V}_{t+\Delta_t}) \\ (\sigma_X^{(t+\Delta_t)})^2 = (1-\rho^2)\Delta_t(\gamma_1\hat{V}_t + \gamma_2\hat{V}_{t+\Delta_t}) \end{cases} \tag{1.106}$$

So the simulation value $\hat{X}_{t+\Delta_t}$ is obtained by:

$$\hat{X}_{t+\Delta_t} = \mu_X^{(t+\Delta_t)} + \sigma_X^{(t+\Delta_t)} \times Z_X \tag{1.107}$$

with

$$Z_X \sim N(0;1)$$

In the next two sections, we have different schemes for simulating $V_{t+\Delta_t}$ from V_t, insuring that its simulated value keeps positive.

Truncature Gaussian scheme for Variance process

The Truncature Gaussian [TG] scheme assumes that the simulated value $\hat{V}_{t+\Delta_t}$ is given by a normal variable with parameters μ and σ, and floored by zero (to enforce the positivity of the Variance process).

$$\boxed{\hat{V}_{t+\Delta_t} = (\mu + \sigma Z_V)^+} \tag{1.108}$$

where μ and σ are parameters that depend on $\hat{V}_t$, κ, θ, σ_v, Δ_t etc... and

$$Z_V \sim N(0,1)$$

Finding the parameters μ and σ consist in performing a calibration procedure that will satisfy the following moment matching :

$$\begin{cases} \mathbb{E}[\hat{V}_{t+\Delta_t}] = \mathbb{E}[V_{t+\Delta_t}|V_t = \hat{V}_t] = m(t, t+\Delta_t) \\ \mathbb{E}[\hat{V}^2_{t+\Delta_t}] = \mathbb{E}[V^2_{t+\Delta_t}|V_t = \hat{V}_t] = s^2(t, t+\Delta_t) \end{cases} \tag{1.109}$$

Quadratic Exponential scheme for Variance process

The Quadratic Exponential [QE] scheme assumes that the simulated value $\hat{V}_{t+\Delta_t}$ can be given by either:

- A **Quadratic** transformation of a normal variable.

$$\boxed{\hat{V}_{t+\Delta_t} = a \times (b + Z_V)^2} \tag{1.110}$$

  where a and b are parameters that depend on $\hat{V}_t$, κ, θ, σ_v, Δ_t etc... and

$$Z_V \sim N(0, 1)$$

- A random variable whose law is a Dirac mass of strength p supplemented with an **Exponential** tail with parameter β.

$$\boxed{\hat{V}_{t+\Delta_t} = \Psi^{-1}(U_V; p, \beta)} \tag{1.111}$$

  where

$$\Psi^{-1}(u; p, \beta) = \begin{cases} 0 & \text{if } 0 \leq u \leq p \\ \frac{1}{\beta} \ln(\frac{1-p}{1-u}) & \text{if } p < u \leq 1 \end{cases} \tag{1.112}$$

Martingale correction

1.4 Stochastic-Local Volatility Model

The stochastic-local volatility(SLV) model is driven by the following SDEs under the risk-neutral measure:

$$\begin{cases} \frac{dS_t}{S_t} = r(t)dt + \sigma(t, S_t)\Psi(V_t)dW_t^S \\ dV_t = a_v(t, V_t)dt + b_v(t, V_t)dW_t^V \\ d < W_t^S, W_t^V >= \rho dt \end{cases} \tag{1.113}$$

Chapter 2

C++ and Application to Financial Mathematics

In this chapter, we will use a financial mathematics example to understand how the programming language C++ works and we will code a financial derivatives pricer.

Keywords

- Classes and Objects
- Data members, Function members
- Encapsulation
- Access modifiers *public*, *private*
- *const* usages
- Pointers
- *this* pointer
- Constructors [Default, With arguments, Copy]
- Initialization list
- Passing by reference, passing by value
- Assignment operator
- Destructor
- Inheritance
- Access modifier *protected*
- Virtual methods and Pure virtual methods
- Abstact classes

2.1 C language features

In the following, I assume the reader to be already familiar with the basics of C, such as:

- Basic data types [char, int, double, void]
- Variables
- Operators [+,-,*,/,%,++,–,==,!=,≥,≤,&&,!]
- Loops [while, for, do ... while]
- Functions [return type, function name, parameters, function body]
- Passing function arguments/parameters by reference or by value
- Arrays
- Pointers
- String type
- Structures [struct]

2.2 Class

A class is a way to define a new data type.

It provides two essential features:

- Data members [what identifies any object or instance of that class]
- Function members or methods [behaviors or functionalities that any object can provide]

We start with the class **BlackScholesModel** that represents the simplest Black Scholes model. Such a model would be defined or identified by two elements:

$$\begin{cases} \text{Drift } \mu \\ \text{Volatility } \sigma \end{cases}$$

Those two real number [**double** type in C++] are then the data members of that class.

In the case we want to simulate this model in a context of Monte Carlo method with Euler scheme, we would like the model to be able to return:

$$\begin{cases} \text{The drift term as a function of time and asset price: } \mu(t,s) = \mu \times s \\ \text{The volatility term as a function of time and asset price: } \sigma(t,s) = \sigma \times s \end{cases}$$

Those two functions are then methods of that class.

2.2.1 Declaration [.h]

```
class BlackScholesModel
{
public:
     double drift_term(double time, double asset_price);
     double diffusion_term(double time, double asset_price);
private:
     double _drift;
     double _volatility;
};
```

By convention in many workplaces, the private data members are starting with _ or m_.

You can notice the use of keywords *private* and *public*, *const*.
We are discussing them in the next sections.

2.2.2 Definition/Implementation [.cpp]

```
BlackScholesModel::drift_term(double time, double asset_price)
{
     return _drift * asset_price;
}
```

```
BlackScholesModel::diffusion_term(double time, double asset_price)
{
     return _volatility * asset_price;
}
```

2.3 Object or Instance of a class

Let's call *bs_model* an object of type *BlackScholesModel*.
It is also called an instance of the *BlackScholesModel* class.
It is also seen as a **variable** of type *BlackScholesModel*.

To access the values of its data members, *_drift* and *_volatility* from outside the class scope, we are tempted to write:

```
bs_model._drift        // returns the _drift data from the object bs_model
bs_model._volatility   // returns the _volatility data from the object bs_model
```

However, in C++, class instances don't work like struct objects, where access to data members is public. To be allowed to do that, the data members *_drift* and *_volatility* should be accessible publicly, which is not the desired feature of C++ programming, as you can see in the next section.

2.4 Encapsulation

In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulates them.
Encapsulation also lead to data abstraction or hiding.
We want to allow the data members to be protected from being altered outside the class they are defined.

There are three access modifiers that allow encapsulation:

- public
- protected
- private

From the example above, we can see that in our *BlackScholesModel* class, the two data members are declared private.
This means that no function outside the class can modify the state of any instance of that class directly by using the accessor operator "." as it is done for a struct object.

The methods *drift_term(...)* and *diffusion_term()* are defined public.
This means that any function outside the class can access those methods from any BlackScholesModel instance.

2.5 Pointers

In C++, a pointer refers to a variable whose value is the address of another variable.
For example, a pointer of type *BlackScholesModel* will be declared and defined by the following ways:

```
 // Declaration of two variables of type BlackScholesModel
BlackScholesModel bsmodel1;
BlackScholesModel bsmodel2;
...........
// Definition of bsmodel1 and bsmodel2 [using constructors or assignment operator]
..........

 // Declaration of a pointer of type BlackScholesModel
BlackScholesModel* bsmodel_ptr;

// Assigment of the pointer to the address of the bsmodel1 variable
bsmodel_ptr = &bsmodel1;

// Access and modify the value pointed by the pointer
*bsmodel_ptr = bsmodel2;

// Modifying the pointer : pointing to the another variable bsmodel2
bsmodel_ptr = &bsmodel2;
```

In the example above, the following occurred:

- Two variables *bsmodel1* and *bsmodel2* of type *BlackScholesModel* are declared and defined.
- A pointer *bsmodel_ptr* of type *BlackScholesModel* is declared and assigned to the address of *bsmodel1*.
- The variable pointed by *bsmodel_ptr*, which is *bsmodel1*, is assigned to *bsmodel2*.
- The pointer is then modified: it holds the address of *bsmodel2* variable.
- Conclusion: the entire sequence of actions can be summarized by *bsmodel1* = *bsmodel2* and the pointer pointing from *bsmodel1* to *bsmodel2*.

2.6 this pointer

Any method of a class that needs to access the address of the object/instance that is calling that method can do so by using the keyword **this**.
The keyword "this" is a pointer to the current object of the class, so the reference object is obtained by dereferencing the pointer [*this].

2.7 const usages

2.7.1 Passing argument by reference vs. by value

For any function or class method that have arguments, those arguments can be either:

1. Passed by Reference

```
void modify_variable_byReference(double& var, double new_value)
{
     var = new_value;
     // var is here a reference to the parameter passed in the function.
}
// At the exit of the scope, the parameter var has been modified.
```

2. Passed by Value

```
void modify_variable_byValue(double var, double new_value)
{
     var = new_value;
     // var is here a copy of the parameter passed in the function.
}
// At the exit of the scope, the parameter var has not been modified.
```

The fundamental difference is that passing by value consists in copying the argument and using that copy in the scope of the function, whereas passing by reference directly uses the argument in the scope of the function, allowing for modification of the variable passed by the body of the function.

Passing by value in our example is equivalent to the following code:

```
void modify_variable_byValue(double var, double new_value)
{
     // Calling Copy Constructor of the double class.
     double var_copy(var);
     var_copy = new_value;
}
// At the exit of the scope, var_copy,
// which is a local variable, is destroyed, by calling its Destructor.
```

2.7.2 const parameter

As seen above, parameters of a function can be passed by reference or by value.
For memory purposes, we want to avoid duplicating the memory of existing objects by calling parameters by value.
But passing by reference presents a risk ... that the parameter can be modified by the method.
Therefore, we are using the keyword *const* that obliges the method to NOT MODIFY the parameter.

```
// This method won't work
void modify_variable_byReference(const double& var, double new_value)
```

```
{
     var = new_value;
     // var is declared const reference, therefore cannot be changed
     // Assignment operator cannot be called.
}
```

2.7.3 const methods

Apart from insuring that the arguments of a methods cannot be modified, we want to insure that a method does not change the instance of the class itself, meaning that it doesn't change the data members of the object referred by *this.
To do this, we need to declare the method *const*:

```
class BlackScholesModel
{
public:
     double drift_term(double time, double asset_price) const;
     double diffusion_term(double time, double asset_price) const;
private:
     double _drift;
     double _volatility;
};

BlackScholesModel::drift_term(const double& time, const double& asset_price) const
{
     return _drift * asset_price;
}

BlackScholesModel::diffusion_term(const double& time, const double& asset_price) const
{
     return _volatility * asset_price;
}
```

In our example, the bodies of the methods *drift_term(...)* and *diffusion_term(...)* don't modify any of the data members *_drift* or *_volatility*, so they are valid const methods.

2.7.4 const pointer

Let's come back to the example used when we introduced the concept of pointers.

- **Non-const pointer to a const data**

```
 // Declaration of two variables of type BlackScholesModel
BlackScholesModel bsmodel1;
BlackScholesModel bsmodel2;
...........
// Definition of bsmodel1 and bsmodel2 [using constructors or assignment operator]
..........
```

```
 // Declaration of a pointer of type CONST BlackScholesModel
// Assigment of the pointer to the address of the bsmodel1 variable
const BlackScholesModel*  bsmodel_ptr = &bsmodel1;

// Access and modify the value pointed by the pointer
// FORBIDDEN : the code won't compile !!
*bsmodel_ptr = bsmodel2;

// Modifying the pointer : pointing to the another variable bsmodel2 [OK]
bsmodel_ptr = &bsmodel2;
```

- **Const pointer to a non-const data**

```
 // Declaration of two variables of type BlackScholesModel
BlackScholesModel bsmodel1;
BlackScholesModel bsmodel2;
...........
// Definition of bsmodel1 and bsmodel2 [using constructors or assignment operator]
..........

 // Declaration of a CONST pointer of type BlackScholesModel
// Assigment of the pointer to the address of the bsmodel1 variable
BlackScholesModel* const  bsmodel_ptr = &bsmodel1;

// Access and modify the value pointed by the pointer [OK]
*bsmodel_ptr = bsmodel2;

// Modifying the pointer : pointing to the another variable bsmodel2
// FORBIDDEN : the code won't compile !!
bsmodel_ptr = &bsmodel2;
```

- **Const pointer to a const data**

```
 // Declaration of two variables of type BlackScholesModel
BlackScholesModel bsmodel1;
BlackScholesModel bsmodel2;
...........
// Definition of bsmodel1 and bsmodel2 [using constructors or assignment operator]
..........

 // Declaration of a CONST pointer of type CONST BlackScholesModel
// Assigment of the pointer to the address of the bsmodel1 variable
const BlackScholesModel* const  bsmodel_ptr = &bsmodel1;

// Access and modify the value pointed by the pointer
// FORBIDDEN : the code won't compile !!
*bsmodel_ptr = bsmodel2;

// Modifying the pointer : pointing to the another variable bsmodel2
```

```
    // FORBIDDEN : the code won't compile !!
    bsmodel_ptr = &bsmodel2;
```

2.7.5 Try to use const as much as possible

The main rule of thumb here is:
Use const whenever possible and relevant!
This will ensure the safest level of code in terms of manipulation of variables.
The *BlackScholesModel* class declaration is then improved to the following:

```
class BlackScholesModel
{
public:
    double drift_term(const double& time, const double& asset_price) const;
    double diffusion_term(const double& time, const double& asset_price) const;
private:
    double _drift;
    double _volatility;
};
```

2.8 Constructors

A constructor is a method that has no return type, and has to be called with the same name as the name of the class. An object of a class that is declared but not "defined" yet has to be initialized starting somewhere.

2.8.1 Default constructor

The method that takes care of this default initialization is the **default constructor**.
The default constructor has no arguments/parameters.

```
class BlackScholesModel
{
public:
     // Default constructor
     BlackScholesModel();

     ......................

private:
     double _drift;
     double _volatility;
};
```

For the implementation of this constructor, we could assign the data members with default values (for instance, 0. and 0.) inside the body of the function.
But there is a more efficient way that is called **initialization list**:

```
BlackScholesModel::BlackScholesModel()
     :_drift(0.), _volatility(0.)
{
}
```

In the below line of code, the default constructor is automatically called

```
// Declaration of a variable without explicit constructor
// calls automatically default constructor
BlackScholesModel bsmodel;
```

2.8.2 Constructor with parameters/arguments

Obviously, when instantiating an object, we want to initialize the data members by giving them values.
To do that we can use a constructor with parameters:

```
class BlackScholesModel
{
public:
     ..............
```

```
     // Constructor with parameters
     BlackScholesModel(const double& drift, const double& volatility);

    ..............

private:
     double _drift;
     double _volatility;
};
```

For the implementation of this constructor, we could assign the data members inside the body of the function.
But there is a more efficient way that is called **initialization list**:

```
BlackScholesModel::BlackScholesModel(const double& drift, const double& volatility)
     :_drift(drift), _volatility(volatility)
{
}
```

We can call the constructor with parameters directly to instantiate the class:

```
// Constructing the object with drift 0.03 and volatility 0.2
BlackScholesModel bsmodel(0.03, 0.2);
```

We can use the memory allocation keyword **new** to create a pointer to a constructed instance of the class:

```
// Constructing the pointer to an object with drift 0.03 and volatility 0.2
BlackScholesModel* bsmodel_ptr = new BlackScholesModel(0.03, 0.2);
```

2.8.3 Copy constructor

We also want to be able to instantiate an object as a copy of another one.
To do so, there is a type of constructor called copy constructor that is defined below.

```
class BlackScholesModel
{
public:
  ...........

     // Copy constructor
     BlackScholesModel(const BlackScholesModel& model);

   ...........

private:
     double _drift;
     double _volatility;
};
```

The reason why the model parameter is passed by reference is that if it were passed by value, it would be copied using the copy constructor ... which is the method we are implementing!

So that's why the copy constructor takes a parameter that is passed by const reference, also avoiding the risk of modifying the object to be copied.
The use of **initialization list** is also possible here:

```
BlackScholesModel::BlackScholesModel(const BlackScholesModel& model)
     :_drift(model._drift), _volatility(model._volatility)
{
}
```

We can call the copy constructor directly to instantiate the class:

```
// Constructing the object with drift 0.03 and volatility 0.2
BlackScholesModel bsmodel1(0.03, 0.2);

// Calling copy constructor with argument bsmodel1
BlackScholesModel bsmodel2(bsmodel1);
```

We can use the memory allocation keyword **new** to create a pointer to a constructed instance of the class:

```
// Constructing the object with drift 0.03 and volatility 0.2
BlackScholesModel bsmodel(0.03, 0.2);

// Constructing pointer to a copy of  bsmodel
BlackScholesModel* bsmodel_ptr = new BlackScholesModel(bsmodel);
```

2.9 Assignment operator

A similar operation that copies an object from another one is the usage the assignment operator "=".
The difference between copying and assigning is that **copying** actually **constructs** the object, while **assigning** can only be done to an object that is **already constructed**.
Therefore, the return type of the assignment operator should be the object itself (***this**).

```
class BlackScholesModel
{
public:
  ..........

     // Assignment operator
     BlackScholesModel& operator=(const BlackScholesModel& model);

   ..........

private:
     double _drift;
     double _volatility;
};
```

The implementation works as follow:

```
// "this" is the pointer of the current object
BlackScholesModel& BlackScholesModel::operator=(const BlackScholesModel& model)
{
     if (this != &model)
     {
          _drift = model._drift;
          _volatility = model._volatility;
     }
     return *this;
}
```

What the code above does is the following:

- If the address of the current object [this] is the same as the address of the variable *model* [&model] then no need to assign anything, as those variables are the same.
- Otherwise, we operate the assignment operation data member by data member [_drift and _volatility].
- Finally, return the reference to the current object [*this].

2.10 Destructor

The destructor is the method in charge of freeing the memory occupied by the data members of the instance of the class.
There can be **only one destructor** and it **does not take any parameter**. There is no need to re-implement the destructor method unless the class contains **at least one pointer as a data member**.
The reason being that the default behavior of the destructor would delete the variable that contains the address of the object pointed by the pointer, but not the object itself! Therefore this would lead to memory leakage, meaning that some objects can be persisting in memory but their addresses are lost.

```
class BlackScholesModel
{
public:
     ~BlackScholesModel() = default;
     ....................................................
};
```

In the example above, since we don't need to re-implement the destructor method, we declare its default behavior using the keyword **default**.

2.11 Inheritance

Inheritance is a very powerful way of designing C++ code.
It allows specifications to be coded in a minimal amount of time and code.

For example, designing a Black Scholes model with jumps as a class could reuse the data members of the BlackScholes class.
Below is the example of *BlackScholesJumpModel* which is a class inherited or derived from *BlackScholesModel*.
We can also say that *BlackScholesModel* is a base class and *BlackScholesJumpModel* is a subclass of *BlackScholesModel*.

```
class BlackScholesModel
{
public:
     ....................................................

protected: // [Protected keyword]
double _drift;         // mu [or r under risk-neutral mwasure]
double _volatility;    // sigma
};

class BlackScholesJumpModel : public BlackScholesModel  // [Public Inheritance]
{
public:
     ....................................................

private:
     double _intensity; //lambda
     double _jump_mean; // m
     double _jump_variance; // s^2
};
```

You can notice two things in the example above:

- **Protected keyword**: all data and function members that need to be accessed by a class and its subclasses only, not outside of those, can be under the **protected** scope.
  In our example, *_drift* and *_volatility* can be accessed by any method of *BlackScholesModel* and *BlackScholesJumpModel*, but not outside of those two classes.
- **Public Inheritance**: the *BlackScholesJumpModel* class is publicly derived from the *BlackScholesModel*.
  Public inheritance means that public data and function members from the base class will remain public in the derived class, protected data and function members from the base class will remain protected in the derived class, and private data and function members from the base class won't be accessible in the derived class.
  There is also protected and private inheritance, but we won't be using them.

2.11.1 Constructors of a subclass/derived class

When constructing an object of a subclass type, the order of construction is the following:

1. Constructing the base class part of the object.
2. Constructing the derived class part of the object.

Default constructor

```
class BlackScholesJumpModel : public BlackScholesModel
{
public:
     // Default constructor
     BlackScholesJumpModel();

private:
     double _intensity; //lambda
     double _jump_mean; // m
     double _jump_variance; // s^2
};
```

The implementation can use the initialization list to call the base class default constructor:

```
BlackScholesJumpModel::BlackScholesJumpModel()
     : BlackScholesModel(),
      _intensity(0.),
     _jump_mean(0.),
     _jump_variance(0.)
{}
```

Constructor with parameters

```
class BlackScholesJumpModel : public BlackScholesModel
{
public:
     // The subclass constructor needs the base class parameters
     // [drift and volatility] to be constructed
     BlackScholesJumpModel(const double& drift,
                           const double& volatility,
                           const double& intensity,
                           const double& jump_mean,
                           const double& jump_variance);

private:
     double _intensity; //lambda
     double _jump_mean; // m
     double _jump_variance; // s^2
};
```

The implementation can use the initialization list to call the base class constructor with parameters:

```
BlackScholesJumpModel::BlackScholesJumpModel(const double& drift,
                          const double& volatility,
```

```
                        const double& intensity,
                        const double& jump_mean,
                        const double& jump_variance):
     BlackScholesModel(drift, volatility),
     _intensity(intensity),
     _jump_mean(jump_mean),
     _jump_variance(jump_variance)
{}
```

Copy constructor

```
class BlackScholesJumpModel : public BlackScholesModel
{
public:
     BlackScholesJumpModel(const BlackScholesJumpModel& model);

     ..............................
};
```

The implementation can use the initialization list to call the base class copy constructor.
The call *BlackScholesModel(model)* is possible because a reference of type *BlackScholesJumpModel* is also a reference of type *BlackScholesModel*.
The opposite is not true!

```
BlackScholesJumpModel:: BlackScholesJumpModel(const BlackScholesJumpModel& model):
     BlackScholesModel(model),
     _intensity(model._intensity),
     _jump_mean(model._jump_mean),
     _jump_variance(model._jump_variance)
{}
```

Assignement operator

```
class BlackScholesJumpModel : public BlackScholesModel
{
public:
     BlackScholesJumpModel& operator=(const BlackScholesJumpModel& model);

     ..............................
};
```

The implementation is similar to the one of the base class and can re-use the call to the assignment operator of the base class:

- If the address of the current object [this] is the same as the address of the variable *model* [&model] then no need to assign anything, as those variables are the same.
- Otherwise, we operate the assignment operation call from the base class [_drift and _volatility], then on the specialized data member by data member[_intensity, _jump_mean and _jump_variance].
- Finally, return the reference to the current object [*this].

```
BlackScholesJumpModel& BlackScholesJumpModel::operator=(const BlackScholesJumpModel& model)
{
     if (this != &model)
     {
          BlackScholesModel::operator=(model);
          _intensity = model._intensity;
          _jump_mean = model._jump_mean;
          _jump_variance = model._jump_variance;
     }
     return *this;
}
```

2.12 Virtuality

The keyword "virtual" is essential to make sure the correct method is called by a subclass instance. The typical example is when there is a pointer to a base class, but constructed as a specific subclass instance. If the subclass re-implements a method from the base class, how can we ensure which method is called?

```
class BlackScholesModel
{
public:
     double drift_term(const double& time, const double& asset_price) const;

     ..............................

};

class BlackScholesJumpModel : public BlackScholesModel  // [Public Inheritance]
{
public:
     // re-implementing the base class method
     double drift_term(const double& time, const double& asset_price) const;

     ..............................
};
```

We are in the case where the two methods have same signature but not same body [or behavior]:

```
double BlackScholesModel::drift_term(const double& time, const double& asset_price) const
{
     return _drift * asset_price;
}

double BlackScholesJumpModel::drift_term(const double& time, const double& asset_price) const
{
     return (_drift - _intensity) * asset_price;
}
```

Next, let's look at the following piece of code:

```
// Pointer of base class type BlackScholesModel, calling BlackScholesJumpModel constructor with pa
BlackScholesModel bsModel_ptr = new BlackScholesJumpModel(0.03, 0.2, 0.5, 0., 1.);

// Calling drift_term(...) method
double time = 1.;
double asset_price = 100.;

double drift = bsModel_ptr->drift_term(time, asset_price);
```

Which method is this pointer calling ???
The answer is: it depends if the base class method has been declared **virtual**.

1. Base class method declared virtual:

```
class BlackScholesModel
{
public:
     virtual double drift_term(const double& time, const double& asset_price) const;

     ..............................
};
```

In this case, the method from the Derived class *BlackScholesJumpModel* will be called.

2. Base class method NOT declared virtual:

```
class BlackScholesModel
{
public:
     double drift_term(const double& time, const double& asset_price) const;

     ..............................
};
```

In this case, the method from the Base class *BlackScholesModel* will be called.

Therefore is it safer to declare virtual all base class methods that need to be re-implemented by subclasses.
Once this is done, the derived class methods that re-implement those methods can have an extra keyword **override** that ensures to the compiler that the base class method must have been declared virtual [the compiler will return an exception if this is not the case].

```
class BlackScholesModel
{
public:
     virtual double drift_term(const double& time, const double& asset_price) const;

     ..............................
};

class BlackScholesJumpModel : public BlackScholesModel  // [Public Inheritance]
{
public:
     // re-implementing the base class method
     double drift_term(const double& time, const double& asset_price) const override;

     ..............................
};
```

2.12.1 Virtual destructor

A very important use of the **virtual** keyword is in the case of destruction and inheritance.
When destructing an object of a subclass type, the order of destruction is the following:

1. Destructing the derived class part of the object.

2. Destructing the base class part of the object.

The rule of thumb is to always declare a destructor virtual for a base class.
From the previous section, after declaring a pointer of type *BlackScholesModel* but pointing to an object of type *BlackScholesJumpModel*, once the **delete** operation is performed on the pointer, if the Base class destructor is not declared virtual, then only the base class element of the object will be destroyed, not the specialized part. This will lead to some memory leaks and unstable behaviors.

2.13 Abstract class

Imagine a class that will represent all financial models in general.
Such a model is represented by a stochastic process $(S_t)_{t\geq 0}$ and is then defined by its 2 SDE components:

- Drift : $\mu(t, s)$
- Diffusion : $\sigma(t, s)$

The two methods of such a class don't have any default behavior.
Only the sub-classes of that class will implement the behaviors of *drift_term* and *diffusion_term*.
They should therefore be virtual pure in the scope of Model class, which makes the latter an **abstract** class.

```
class Model
{
public:
     virtual double drift_term(const double& time, const double& asset_price) const = 0;
     virtual double diffusion_term(const double& time, const double& asset_price) const = 0;

private:
};
```

2.14 clone() method

When a class contains a data member which is a pointer to an abstract class, how can the constructor be implemented properly?
Let's take the example of the class **PathSimulator** that has a data member which a pointer to the abstract class **Model**:

```
class PathSimulator
{
public:
     ......................

     PathSimulator(const double& initial_value,
                   const std::vector<double>& time_points,
                   const Model& model);

protected:
     ......................

     double _initial_value;
     std::vector<double> _time_points; // dynamic array that represents [t_0, t_1, .... t_M]
     const Model* _model;
};
```

The problem here is that the object pointed by *_model* needs to be copied from the constructor parameter *model*?
As **Model** is an abstract class, we cannot instantiate it, we cannot create a variable of type

Model to be pointed by *_model*.
The solution is then to create a pure virtual method inside the abstract base class **Model** that will be responsible for creating a new pointer to a copy of any instance of a subclass of **Model**.

```
class Model
{
public:
     virtual Model* clone() const = 0;

     ..............................

};

class BlackScholesModel : public Model
{
public:
     BlackScholesModel* clone() const override;

     ......................
};

class DupireLocalVolatilityModel : public Model
{
public:
     DupireLocalVolatilityModel* clone() const override;

     ........................
};
```

To achieve the correct behavior in each subclass, the implementation consists in creating a new pointer by using the keyword **new**, and calling the copy constructor of the current subclass, taking as parameter the reference to the current object ***this**;

```
BlackScholesModel * BlackScholesModel::clone() const
{
     return new BlackScholesModel(*this);
}

DupireLocalVolatilityModel * DupireLocalVolatilityModel::clone() const
{
     return new DupireLocalVolatilityModel(*this);
}
```

Once this is done, the constructor in *PathSimulator* works as follows:

```
PathSimulator::PathSimulator(const double& initial_value,
                             const std::vector<double>& time_points,
                             const Model& model)
     : _initial_value(initial_value),
       _time_points(time_points),
       _model(model.clone())
```

```
{
}
```

The data member pointer *_model* is assured to be correctly constructed and to point to a copy of the variable **model* with the correct corresponding type [subclass].

2.14.1 Example of re-implementing the destructor

PathSimulator class has one of its data member which is a pointer.
Therefore we need to re-implement the destructor method. As PathSimulator is a base class and gets inherited, we also need to declare the destructor virtual.

```
class PathSimulator
{
public:
     ..................

     virtual ~PathSimulator();

protected:
     ..................

     double _initial_value;
     std::vector<double> _time_points; // dynamic array that represents [t_0, t_1, .... t_M]
     const Model* _model;
};
```

The implementation consists in calling the keyword **delete** on the pointer, that will implicitly call the destructor of the variable type pointed by the pointer.

```
PathSimulator::~PathSimulator()
{
     delete _model;

     // Calls Destructor method of Model to destroy *_model.
     // Model's destructor method should then be declared virtual
     // to make sure the right Model Derived class destructor is called.
}
```

Appendix A

Fokker-Planck equation for one-dimensional SDE

Let $(X_t)_{t\geq 0}$ be a stochastic process of the form:

$$dX_t = a(t, X_t)dt + b(t, X_t)dW_t \tag{A.1}$$

and π be its probability density:

$$\pi(t, x) = f_{X_t}(x) \tag{A.2}$$

Then π satisfies the forward Kolmogorov equation:

$$\frac{\partial \pi}{\partial t} = -\frac{\partial}{\partial x}(a\pi) + \frac{1}{2}\frac{\partial^2}{\partial x^2}(b^2\pi) \tag{A.3}$$

Appendix B

Feynman-Kac PDE for 2D models

Let's consider an asset $(S_t)_{t\in[0,T]}$ whose SDE under a measure $\mathbb{Q}$ is driven by:

$$\begin{cases} \frac{dS_t}{S_t} = r(t)dt + \Psi(V_t)dW_t^S \\ dV_t = a_v(t,V_t)dt + b_v(t,V_t)dW_t^V \\ d < W_t^S, W_t^V >= \rho dt \end{cases} \tag{B.1}$$

Any contingent claim $U = U(t, S_t, V_t)$ which has an European payoff function ϕ at maturity T of the form $\phi(S_T, V_T)$, its PDE is given by:

$$\begin{cases} \frac{\partial U}{\partial t} + r(t)S\frac{\partial U}{\partial S} + a_v(t,V)\frac{\partial U}{\partial V} + \frac{1}{2}\Psi(V)^2S^2\frac{\partial^2 U}{\partial S^2} + \frac{1}{2}b_v(t,V)^2\frac{\partial^2 U}{\partial V^2} + \rho\Psi(V)Sb_v(t,V)\frac{\partial^2 U}{\partial S\partial V} - r(t)U = 0 \\ \\ U(T,S,V) = \phi(S,V) \end{cases} \tag{B.2}$$

The Feynman-Kac theorem states then that the solution of this PDE verifies:

$$U(0,S_0,V_0) = e^{-\int_0^T r(s)ds}\ \mathbb{E}^{\mathbb{Q}}[\phi(S_T,V_T)] \tag{B.3}$$

The undiscounted price $\hat{U} = \hat{U}(t,S_t,V_t) = e^{\int_t^T r(s)ds}U(t,S_t,V_t)$ then follows the Feynman-Kac PDE:

$$\begin{cases} \frac{\partial \hat{U}}{\partial t} + r(t)S\frac{\partial \hat{U}}{\partial S} + a_v(t,V)\frac{\partial \hat{U}}{\partial V} + \frac{1}{2}\Psi(V)^2S^2\frac{\partial^2 \hat{U}}{\partial S^2} + \frac{1}{2}b_v(t,V)^2\frac{\partial^2 \hat{U}}{\partial V^2} + \rho\Psi(V)Sb_v(t,V)\frac{\partial^2 \hat{U}}{\partial S\partial V} = 0 \\ \\ \hat{U}(T,S,V) = \phi(S,V) \end{cases} \tag{B.4}$$

The solution of this PDE is obviously:

$$\hat{U}(0,S_0,V_0) = \mathbb{E}^{\mathbb{Q}}[\phi(S_T,V_T)] \tag{B.5}$$

Appendix C

C++ Need-To-Know

C.1 const parameter/argument inside a method's signature

An argument of a method's signature is declared *const* if the method doesn't intend to modify the state of that argument.
It is similar to a read-only kind of access.

C.2 const method

A method of a class is *const* if it doesn't modify the state of the object that will call it.
In other words, a *const* method cannot modify the object *this.
As long as the data members are not modified inside the scope of a method, this method should be declared const.
A const method cannot call non-const methods inside its body.

C.3 Virtual

Let's have a base class A and one of its subclasses B which have a same method prototype, say *func()*.
A pointer *ptr* on A, that is constructed using a constructor of B, calling *func()*, will:

- Call the A's *func()* method if that method has not been declared virtual in A.
- Call the B's *func()* method if that method has been declared virtual in A.

C.4 Abstract class

An abstract class is a class that has at least one pure virtual method.