Libor Market Model and CVA

Jose Melo

Santiago, Chile
January 2020

Contents

1 Introduction

In this report we explain all the parts needed to be taken into consideration in order to deploy the Libor Market Model successfully and be able to price interest rate derivatives, and in particular CVA. Market models is a family of interest rate models that follow the HJM framework and attempt to simulate multi factor curves, obtaining complete yield curve realizations, being this one of the main differences between this types of models and short-rate models.

The popularity of this model, proposed by Brace, Gatarek and Musiela, arises from the fact that it is able to prices products consistently with Black's formula. It's popularity has made this model one of the main tools to calculate xVA components in financial institutions.

Calculation and management of xVA components has been focus of attention because with the financial crisis, financial institutions realized that this impacts were significant in the derivatives dealing business. After the crisis, CVA started to be incorporated in prices as a market practice and CSA agreements began to flourish across institutions, affirming that the way derivatives were seen was changed.

In this report, we attempt to build a pricing framework to calculate derivatives prices and CVA. This task requires diving into concepts like volatility, correlation and rate models. Each section in this report can, by it self, make a complete study subject, so here we try to emphasize in the main detail that are needed for this particular model. As a simplification, all formulae and code in this project assumes constant year fractions, in particular, 0.5 for 6 month periods, because working with proper year fractions requires developing day counter classes and proper calendar structures that are beyond the scope of this work.

2 Interpolation

Interpolation is an important subject in finance, specially because we work with discrete data altogether with continuous models. For this project, we implement the following methods:

- Linear Interpolation
- Cubic Spline Interpolation

2.1 Linear Interpolation

Linear interpolation is widely used in the financial industry for its simplicity, even tough is the least preferred option in more complex problems, because it assumes that the value interpolated lies over the straight line between two colliding points, but many financial application exhibit non-linear relationships. Its formulation is as follows:

$$y = y_0 + (x - x_0)\frac{y_1 - y_0}{x_1 - x_0}$$

Where x is a value inside the interval (x_0, x_1). Linear interpolation is used usually to interpolate rates, but it produces non-smooth forward rate curves that lead to local arbitrage opportunities.

2.2 Cubic Splines

Cubic spline interpolation is achieved by fitting a piece-wise polynomial -twice continuously differentiable- to the data points. At one hand, this type of interpolation produces smooth functions with continuously changing first and second order derivatives across the data set, but at the other might produce spurious oscillations or unrealistic shapes in the interpolated functions.

This last problem can be mitigated by using monotonic cubic interpolation, which aims to produce an increasing/decreasing curve at each data point. Following [1], for each x, inside an interval (x_i, x_{i+1}), this is achieved by:

$$f(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i$$

Where a_i,b_i,c_i,d_i are determined by:

$$a_i = (y_i^{'} + y_{i+1}^{'} - 2s_i)/h_i^2$$

$$b_i = (3s_i - 2y_i^{'} - y_{i+1}^{'})/h_i$$

$$c_i = y_i^{'}$$

$$d_i = y_i$$

with $h_i = (x_{i+1} - x_i)$ and $s_i = (y_{i+1} - y_i)/h_i$. For $y_i^{'}$ we set:

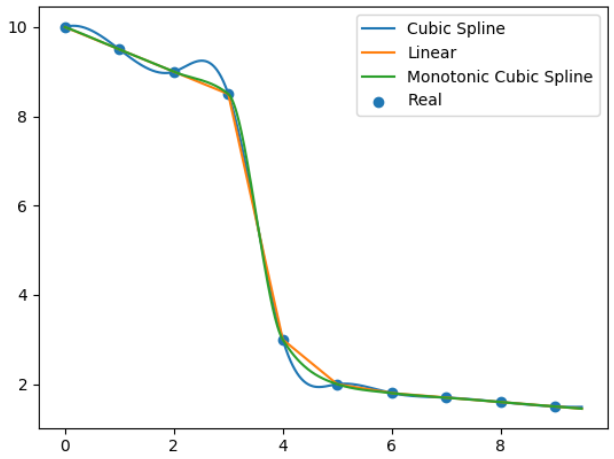
$$y_i^{'} = \begin{cases} 0 & s_{i-1}s_i \leq 0 \\ 2a \min(|s_{i-1}|, |s_i|), a = sign(s_{i-1}) = sign(s_i) & |p_i| > 2|s_{i-1}|, |p_i| > 2|s_i| \\ p_i & otherwise \end{cases}$$

and $p_i = (s_{i-1}h_i + s_i h_{i-1})/(h_{i-1} + h_i)$.

2.3 Code - *Interpolator* Class

Figure [1] presents the interpolation methods mentioned above. We notice that linear interpolation looses information related to the curvature of the function, but at the same time the cubic spline, obtained from *scipy.interpolate.CubicSpline* class, produces unwanted oscillations. The monotonic spline interpolation produces the best result for finance related applications, since most of the functions we need to approximate are monotonic, like forward rate and discount factor curves.

Figure 1: Interpolation types



The interpolation function we implement works as follows:

```
options = {'kernel':'CS','extrapolate':True}
interpolator = Interpolator(x, y, options)
value = interpolator(x_hat)
```

Where *options* sets the interpolation method with the *kernel* key. Interpolation options available are *linear*, *CS* (cubic spline) and *MCS* (monotonic cubic spline). The variable *extrapolation* is self explanatory; if it is set to false, a *ValueError* exception is raised.

3 Curves

The main input for our model are interest rates so we need to calculate proper discount factors that reflect the prices observed in the market and are consistent with the yield curve we observe. For this matter we use two approaches:

- For interest rate curves: we use QuantLib to obtain forward rates, since developing a proper multi-curve bootstrapping framework requires setting up multiple products that are beyond of the scope of this project. A curve handler class is define to be able to use *numpy* vectors altogether with QuantLib curves.
- For CDS curves: once the rates needed to price CDS are obtained, a simple probability of survival (P) bootstrapping algorithm is provided.

3.0.1 Code - Bootstrapping in QuantLib

Most of the production-grade algorithms for bootstrapping rates or discount factors use recurrent procedures where different products are priced according to a fixed rate or known parameter and that resulting value is compared to the one obtained with the curve we are looking for. In particular, QuantLib uses helper classes to define products that will be priced with the curve:

```
swap = SwapRateHelper(r,period,calendar,frequency,convention,index)
ois = OISRateHelper(r,period,calendar,frequency,convention,index)
deposit = DepositRateHelper(r,period,calendar,convention)
```

Since we are providing information for the fixed leg of each instrument, we know it's value today. Then, our bootstrapping algorithm needs to find the floating rates required to match those prices.

In order to use QuantLib yield curves with *numpy* and our code, we create a handler class:

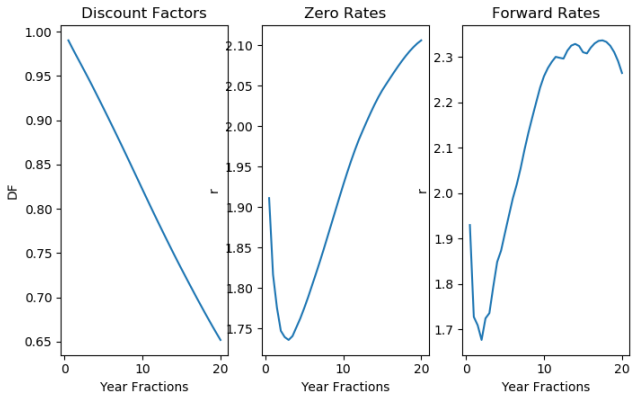
```
from abc import ABC
import QuantLib as ql

class Curve(ABC):
    def forward_rate(self,t, T):
        df_T = self.discount(T)
        df_t = self.discount(t)
        return (np.log(df_t) - np.log(df_T))/(T-t)

class QLCurveHandle(Curve):
    def __init__(self, ql_curve):
        self.ql_curve = ql_curve
        self.f = np.vectorize(self.ql_curve.discount)
    def discount(self, t):
        if isinstance(t,(np.ndarray, np.generic)):
            return self.f(t)
        else:
            return self.ql_curve.discount(t)
```

With the help of *np.vectorize()*, we can transform any function into one that is able to handle *numpy* vectors.

Figure 2: Libor 6M Bootstraped Rates



3.1 CDS Curve

Once interest rates are bootstrapped, survival probabilities $P(T_n)$ can be obtained following that:

$$P(T_0) = \frac{L}{L + \pi\tau_n}$$

$$P(T_N) = \frac{\sum_{n=1}^{N-1} D(0,T_n)[LP(T_{n-1}) - (L + \tau\pi_N)P(T_n)]}{D(0,T_N)(L + \tau_n\pi_N l))} + \frac{P(T_{N-1})L}{L + \tau_N\pi_N}$$

where $P(T_n)$ is the probability of survival, $D(0,T_i)$ represents the discount factor up to time T_i, L is the loss given default -1 $-$ *recovery*-, π_N is the CDS premium for maturity N and τ_n is the year fraction between iteration.

3.1.1 Code - *CDSCurve* Class

The *CDSCurve* class provides all the methods required for obtaining $P(T_n)$, default probabilities (dP) and hazard rates (λ):

```
class CDSCurve:
    def __init__(self, ois_curve, options)
    def bootstrap(self, t, r, coupon_freq=0.5, recovery=0.4)
    def dP(self, t, T)
    def P(self, t, T)
    def hazard_rate(self, t, T)
```

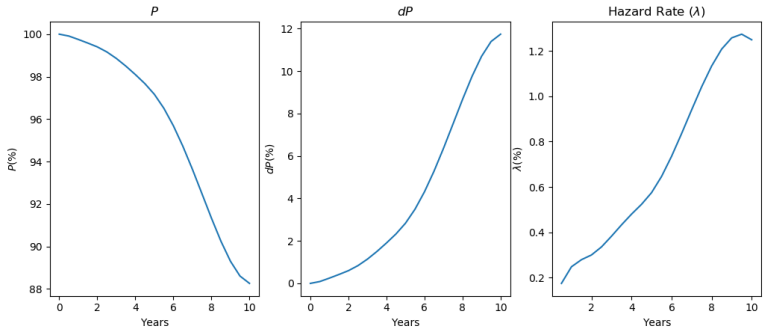
The logic behind the main method of this class, *bootstrap()*, can be best described in the following lines:

```
for yf in yfs:
            if yf <= 0.5:
                pi = par_rates(yf)
                p = L/(L+pi*coupon_freq)
            else:
```

```
                yf_ = np.arange(coupon_freq,yf+coupon_freq,
    coupon_freq)
                df = self.ois_curve.discount(yf_)
                pi = par_rates(yf)
                tmp = [df[i]*(L*ps[i-1]-(L+coupon_freq*pi)*ps[i])
    for i in range(1,ps.shape[0])]
                p = np.sum(tmp)/(df[-1]*(L+coupon_freq*pi))+ps[-1]*
    L/(L+coupon_freq*pi)
            ps = np.append(ps,p)
```

Here we iterate through all CDS maturities to get, at each time step, P. For the short-term end, we assume one payment is done by the CDS contract.

Figure 3: CDS Bootstrapping Results



4 Volatility

When talking about volatility in the LM model, two parts are neededq1qz to be taken into account. First, we need to strip caplet volatility from cap volatility and second, we need to define the way we will compute instantaneous volatility to be used in our model. Both topics are detailed below.

4.1 Volatility Stripping

The issue that arises with volatility is that we can not observe directly caplet volatility in the market, and therefore we need to rely on a procedure to transform cap prices into caplet prices.

Caps can be broken down into smaller pieces of forward volatility (caplets) with an iterative procedure. First, we define the price of a caplet at time t with strike K, expiring at T_{k-1} and paying at time T_k:

$$Caplet(t, T_k, T_{k-1}, K, \sigma_{ATM}) = D(0, T_K)\tau Black(F_k, K, \sigma_{ATM}\sqrt{T_{k-1} - t})$$

Where $Black(F_k, K, \sigma_{ATM}\sqrt{T_{k-1} - t})$ is the Black's classic formula, F_k the forward rate equivalent for the period and σ_{ATM} is the implied volatility. With this and given the fact that Caps are equivalent to a sum of caplets, we can write:

$$Cap(t, T_k, T_{k-1}, K, \sigma_{ATM}) = \sum_{k=1}^{n} D(0, T_K)\tau Black(F_k, K, \sigma_{ATM}\sqrt{T_{k-1} - t})$$

Is important to notice that for ATM Caps, K is the ATM swap rate for the cap equivalent maturity and also is relevant to mention that is market practice to set $\sigma_{Cap(t=0,T=1)} = \sigma_{Caplet(t=0.5,T=1)}$, so we have our lower boundary condition for the stripping algorithm.

With an interpolation procedure, we can obtain $\sigma_{Cap(t=0,T=1.5)}$ from the market prices, and therefore obtain with a optimization tool the value of σ_{Caplet} for $(t = 1, T_k = 1.5)$ since:

$$Cap(t = 0, T_k = 1.5) = Cap(t = 0, T_k = 1) + Caplet(t = 1, T_k = 1.5)$$

4.1.1 Code - *CapletVolStrip* Class

This procedure is implemented in the code in the *CapletVolStrip* class. It's usage is presented below:

```
spot_data = {'ois_curve': ois_curve, 'libor_curve':libor_curve}
options = {'kernel':'MCS','extrapolate':True}
vol_boost =  CapletVolStrip(spot_data, options)
t, vol = vol_boost.strip(maturities, cap_vol)
```

The idea behind this class is to set the basic information needed -curves for discounting, projecting the forward rates and interpolation parameters- and then provide the cap volatilities and maturities to the *fit()* method. This last method will run the bootstrapping algorithm, returning the caplet volatilities for each τ.

4.2 Parametric Volatility

Volatility in the market usually exhibit two common shapes across time. First, it presents a humped shape, where volatility increases in the short term and then declines as maturity advances. The other form, is a monotonically decreasing function of time [3]. Because of this, has become market practice to use parametric functions to describe this behavior. One function that has gained popularity is the one proposed by Rebonato [4]:

$$\sigma_k(t) = g(T_{k-1} - t) = (a + b(T_{k-1} - t))e^{-c(T_{k-1}-t)} + d$$

When this parametric form is used to compute volatility in the LM model, produces the volatility table presented in table [1], where σ remains constant depending on the time to expiry.

FWD\Mat	$T_0 - T_1$	$T_1 - T_2$	...	$T_{N-1} - T_N$
F_0	σ_0	σ_1	...	σ_N
F_1	Matured	σ_0	...	σ_{N-1}
...	...	...	...	...
F_N	Matured	Matured	...	σ_0

Table 1: Parametric Instantaneous Volatility

Under this parametric function, the volatility of a ATM caplet maturing at time T_{k-1} can be described as:

$$\sigma_k(ATM) = \sqrt{\frac{1}{T_{k-1} - t} \int_t^{T_{k-1}} g(u)^2 du}$$

We can use use optimization libraries in order to get a, b, c and d, using as objective funciton the minimization of the squared sum of differences between the market and parametric volatilities:

$$arg \min_{a,b,c,d} \sum_{k=1}^{N} \left[\sigma_k(t)^{MKT} - \sqrt{1/(T_{k-1} - t) \int_t^{T_{k-1}} g(u)^2 du} \right]^2$$

4.2.1 Code - *ParametricVolatility* Class

The fitting procedure and instantaneous volatility function is implemented in the *ParametricVolatility* class and it's usage is as follows:

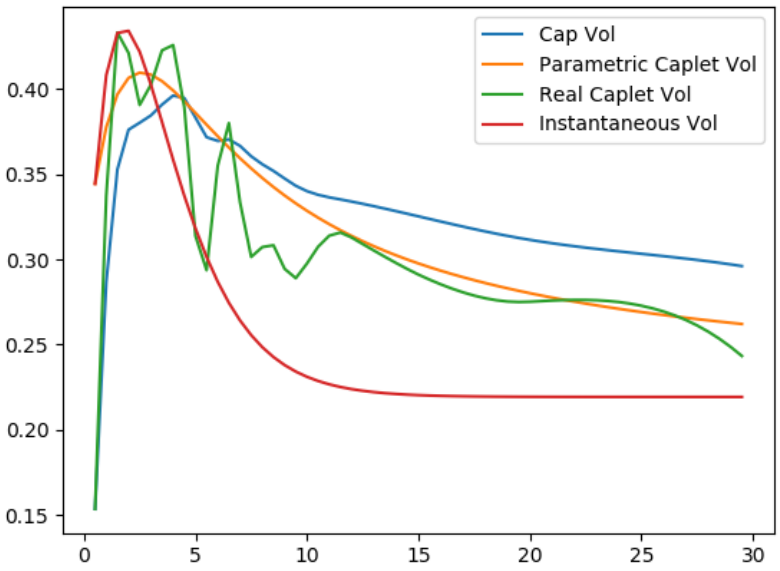
```
vol_f = ParametricVolatility()
vol_f.fit(t,caplet_vols)
instant_vols = vol_f(t0,t1)
```

This class implements a *__call__* method, similar to the *Interpolator* class, that let us call the variable containing the class to get the needed instantaneous volatility. The most relevant lines of code in this class are inside the *caplet_vol()* method, that following [3], calculate $\sqrt{1/(T_{k-1} - t) \int_t^{T_{k-1}} g(u)^2 du}$:

```python
def caplet_vol(self, t, T):
    T = np.arange(0.5,T+0.5,0.5)
    a = self.instant_vol(t,T,self.params)
    q = [0.5*v**2 for v in a]
    for i, x in enumerate(T):
        tmp = np.sqrt(np.sum(q[:i+1])/x)
    return tmp
```

One important point to mention is that parameters a,b,c and d have to be constrained over $(0, +\infty)$, otherwise we might end up with undesired volatility shapes when optimization is done. Scipy's minimize provides two solvers that allows us to work with bound; in particular we use *'L-BFGS-B'* method, but the user is able to change easily by changing the *solver* argument in the *fit()* method.

Figure 4: Cap, Caplet and Instantaneous Volatility



5 Correlation

How we choose to estimate correlation for the Libor Market Model depends on our pricing needs. For example, if the product we are pricing has low dependency on rate correlation, like Swaps or Swaptions, this last can be estimated using historical forward rate time series. At the other hand, if we are pricing products that have strong dependencies on correlation, as the case of spread products, modeling correlation is a more sensible approach. In this project we use historical rates, but we implement a factor reduction scheme that let us work with the most significant eigenvalues when producing our random numbers and correlation matrix, and therefore reduce the computational load required in Monte Carlo simulations.

5.1 Historical Forward Rates

First we need to obtain historical forward rate time series. Here we take advantage that Bloomberg provides already bootstrapped forward rates in Excel, but it also possible to use the classes provided in this report to achieve this task.

After getting the historical rates, since the LM model works with log-normal differences, makes sense to calculate our historical correlation matrix using log returns:

$$\Delta F_k(t) = ln(\frac{F_k(t)}{F_k(t-1)})$$

5.2 Factor Reduction

Simulation of random numbers can become a major bottleneck in the LMM framework, since it needs to draw random numbers accordingly with the number of forward rates that we are simulating. For example, if we are simulating semi-annual rates, and choose a spanning period of 30 years, means we will be requiring 60 correlated random numbers at each time step.

This problem can be overcome with factor reduction. This type of reduction does not have major impacts in the information we are using because forward rates are highly correlated -specially adjacent ones- and the dynamics of the yield curve can by best explained by parallel shifts, bumps and inclination of the curve, without loose much explanatory information.

First, we define a matrix ρ and decompose it by:

$$\rho = Q\lambda Q$$

where Q contains the eigenvectors of ρ and λ is a diagonal matrix with eigenvalues. According to [5], we set to 0 the $N - N_{factors}$ eigenvalues of the ρ matrix, where N is the number of columns, and then take the square root of the remaining values. This results in a matrix $\lambda^{reduced}$. We can define:

$$B = Q\lambda^{reduced}$$

$$\rho^{reduced} = BB^T$$

where B is the Cholesky decomposition of $\rho^{reduced}$. Because this procedure doesn't provide necessarily instant auto correlations equal to one in $\rho^{reduced}$, we re-scale the matrix with:

$$\rho_{k,l}^{*reduced} = \frac{\rho_{k,l}^{reduced}}{\sqrt{\rho_{k,k}^{reduced}\rho_{l,l}^{reduced}}}$$

5.2.1 Code - *CorrelationReduction* Class

This class is intended to produce $\rho^{reduced}$ altogether with the correlated random numbers, so this can be run before we start our Monte Carlo simulation:

```
reduced_correlation = CorrelationReduction(historical_cor)
reduced_correlation.factor_reduction(n_factors=3)
dZ = reduced_correlation.correlated_dZ(n_sim=1000)
```

Again, the most relevant piece of code can be compressed in the following lines:

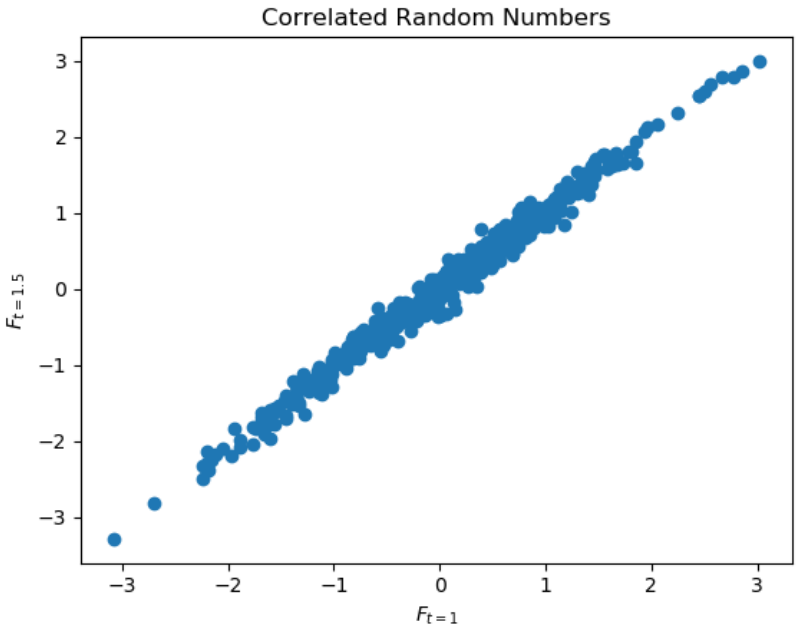
```
def factor_reduction(self,n_factors=3):
        rho = self.corr(return_as='array')
        v, A = np.linalg.eig(rho)

        if n_factors=='all':
            self.n_factors = rho.shape[0]
            B = A @ np.diag(np.sqrt(v))
        else:
            self.n_factors = n_factors
            v[n_factors:] = 0
            B = A @ np.diag(np.sqrt(v))
            B = B[:,0:n_factors]

        reduced = B @ B.T
        for i in range(reduced.shape[0]):
            for j in range(reduced.shape[1]):
                reduced[i,j] = reduced[i,j]/(np.sqrt(reduced[i,i]*
    reduced[j,j]))
        self.reduced = reduced
        self.B = B
        return reduced
```

Here with *np.linalg.eig()* we decompose the *rho* matrix into eigenvalues and eigenvectors. Then we proceed to set to zero the non-relevant eigenvalues and create the B matrix. The output is shown in figure [5].

Figure 5: Correlated dZ



6 Libor Market Model

The main part of this project is the Libor Market Model. As mentioned in our introduction, the popularity of *market models* arises from their capability of pricing caps and swaptions consistently with Black's formula, something that was not possible by short-rate models.

The idea of this model is to produce Libor forward rates which are observable in the market. Lets define each forward rate tenor $0 \leq T_0 \leq T_1 \leq ... \leq T_N$. We are interested in modeling the forward Libor rate $L_n(t) = L(t, T_{n-1}, T_n)$. The log-normal process of L under the risk-neutral measure Q^{T_N} is given by:

$$dL_N(t) = \sigma_N(t) L_N(t) dW_t^{Q^{T_N}}$$

where $\sigma(t)$ is the instantaneous volatility and $dW^{Q^{T_N}}$ is a brownian motion under the measure Q^{T_N}. Notice that $dW_i^{Q^{T_i}}$ and $dW_j^{Q^{T_i}}$ are related via:

$$dW_i^{Q^{T_i}} dW_j^{Q^{T_i}} = \rho_{ij} dt$$

The forward rate dynamics can be written in term of different measures Q^{T_i}

by setting the appropriate drift. In particular:

$$dL_n(t) = \begin{cases} \mu_n(t)dt + \sigma_n(t)L_n(t)dW_n^{Q^{T_i}}(t) & n < i, t \leq T_n \\ \sigma_n(t)L_n(t)dW_n^{Q^{T_i}}(t) & n = i, t \leq T_{i-1} \\ -\mu_n(t)dt + \sigma_n(t)L_n(t)dW_n^{Q^{T_i}}(t) & n > i, t \leq T_{i-1} \end{cases}$$

$$\mu_n(t) = \sigma_n(t)L_n(t) \sum_{j=i+1}^{n} \frac{\rho_{k,j}\sigma_j(t)L_j(t)\tau}{1 + \tau L_j(t)}$$

6.1 Predictor-Correction Scheme

Following [6], since we need are planning to model log-normal forward rates, a sensible scheme for the discrete SDE would be:

$$dL_n(t + \Delta t) = dL_n(t)e_n^{(\mu_n(t) - 0.5\sigma_n(t)^2)\Delta t - \sigma_n(t)^2\sqrt{(}\Delta t)d\tilde{Z}}(t)$$

where $d\tilde{Z}_n(t)$ comes from the Cholesky decomposition achieve with $\rho^{reduced}$. This scheme works well as long as the time steps Δt are small since $\mu_n(t)$ is state-dependent. To calculate the drift properly, we can adjust it with the following procedure:

- Generate one simulation across Δt of $L_n(t)$, lets call it $\tilde{L_n}(t)$, using the unadjusted drift, $\mu_n(t)$.
- Use $\tilde{L_n}(t)$ calculated before and compute $\tilde{\mu_n}(t)$ with it.
- Average $\tilde{\mu_n}(t)$ and $\mu_n(t)$ and use this new drift to make the final realization $L_n(t)$ using the same random variables.

6.2 OIS Discounting in the Libor Market Model

OIS discounting is a requirement in most of the products traded in today's market, because CSA agreements are widely adopted. In this project, OIS discounted is implemented via subtracting the LOIS spread to the Libor Market Model simulation. This approach has it's pros and cons:

- Pros: It's simplicity enables to use the available forward rates to construct OIS discount factors.
- Cons: Basis risk should be modeled because it is a stochastic variable.

While having a stochastic basis model in conjunction with the LM model might sound appealing, it adds an extra layer of complexity to the model, beyond the scope of this project.

6.2.1 Code - *LiborMarketModel* Class

The *LiborMarketModel* class encapsulates all the classes exposed in the previous sections. It's design was thought to be simple for the end use, while retaining the possibility of some options to be adjusted:

```
spot_data = {'ois_curve': ois_curve,
             'libor_curve':libor_curve,
             'cap_vol': (t, vol),
             'historical_rates': historical_rates}
options = {'r_interp':  {'kernel':'MCS','extrapolate':True},
           'df_interp': {'kernel':'MCS','extrapolate':True},
           'vol_interp':{'kernel':'MCS','extrapolate':True},
           'T': 5,
           'corr_factors': 3}

model = LiborMarketModel(options)
model.calibrate(spot_data)
f, P = model.simulate()
```

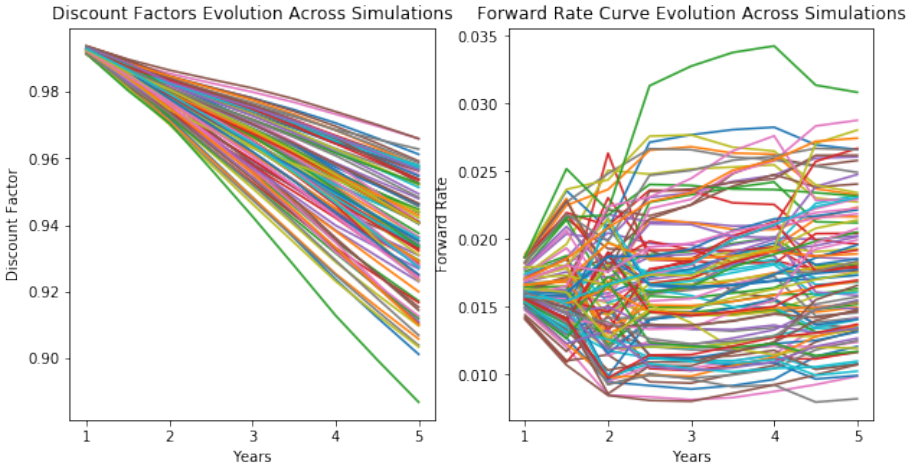
Basically, first we set the model constructor with the options to set the interpolation configuration, last maturity to simulate T and the number of significant factors we want to use. Then, the *calibrate()* method is executed, passing along the spot data, which contains today's curves, volatility and historical forward rate matrix. This method defines and executes all the classes discussed in the previous sections.

After the initialization of the model, we proceed to simulate via Monte Carlo using the *simulate()* method. Here we implement the logic explained in section [6.0.1]. The main lines are:

```
#first drift calculation
tmp = (tau*vol[1:]*f_k[1:]*p[1:])/(1+tau*f_k[1:])
#terminal measure has 0 drift
tmp = np.append(tmp,0)
u_k_1 = np.flipud(np.flipud(tmp).cumsum())
df = (u_k_1 - 0.5*vol**2)*self.dt+vol*np.sqrt(self.dt)*dZ[i,:,k][
    live_T]
f_k_ = f_k*np.exp(df)
#second drift
tmp = (tau*vol[1:]*f_k_[1:]*p[1:])/(1+tau*f_k_[1:])
#terminal measure has 0 drift
tmp = np.append(tmp,0)
u_k_2 = np.flipud(np.flipud(tmp).cumsum())
#final drift
u_k = (u_k_1+u_k_2)/2
```

As mentioned before, we first compute $\mu_n(t)$ normally and then we re-calculate it with $\tilde{L_n}(t)$. *Numpy*'s function *flipud()* helps us to flip the $\mu_n(t)$ vector -since we are simulating all the tenors at once- and with *cumsum()* we compute the sum of all terms. This procedure helps avoid nested Python *for* loops and makes the code more efficient. The curves obtained with the model for $\Delta t = 0.5$ are shown in figure [6].

Figure 6: LMM Output



7 Swaps and CVA

After running our rate simulations, we are almost ready to compute CVA. The credit value adjustment is a pricing adjustment done to derivatives in order to consider the credit risk of a counterparty. Under this framework, risk arrises from the posiblility that one of the parties that entered into the transaction might not pay today's value of the future derivative's cashflows, or mark-to-market. This concept is called expected exposure and is calculated as:

$$EE_t = E_t[\max(MTM_{t,i}, 0)] \quad i = 1..N$$

where N is the number of simulations. Once we have the EE_t, CVA is calculated as:

$$CVA = (1 - R) \int_0^T EE_t DF_t dP_t$$

where R stand for the recovery rate, DF_t is the discount factor and dP_t is the default probability given by CDS contracts. In order to be able to compute CVA, first we need to set up the product we want to price. In this project we price a 6 month Libor Swap, with semi-annual payments.

7.0.1 Code - *IRS* Class

The IRS class, which stands for Interest Rate Swap, is able to get the rate of a swap starting today with 0 value, the par rate, and then compute the evolution of the expected mark-to-market under an specific model. It's usage is simple and self explanatory:

```
swap = IRS(years=5, model=model, side=0)
swap.MTM()
swap.CVA(cds_curve, ois_curve, recovery = 0.4)
```

8 Data

The data used in this project corresponds to market information obtained from Bloomberg, at the end of Decemeber 2019. In order to use the Libor Market Model and calculate CVA of a 6 month Libor (L6M) swap, we need:

- OIS Swap quotes: Is the risk free rate, needed for discounting all products.
- 3 month Libor (L3M) Swap quotes: most liquid swap curve in the market. It is used as base curve to construct the L6M swap.
- 3x6 month Libor Basis Swap quotes: Basis spread used to transform L3M into a L6M swaps. For simplicity we use the L3M swap rate and add the spread, for the same maturity, to get the L6M swap rate.
- OIS Discounted Cap volatility for L6M rates: we need a consistency between our model and the volatility we are using.
- CDS term structure for a company: In our case, JP Morgan Bank. Premiums are needed to bootstrap the default probabilities.
- Historical forward rates for L6M swaps: matrix of forward rates for the L6M swap. Bloomberg provides this information ready to use, under the *BCurveView()* Excel function.

9 Results

First, we analyse the simulation results. A good benchmark for the model's implementation is the available analytical formula for caplets. Table [2] presents a comparison of the two pricing methods.

Years	***LMM***	***BS***	***%***
1	817,67	822,31	-0,56%
2	1.541,05	1.540,34	0,05%
3	2.083,21	2.076,53	0,32%
4	2.507,25	2.528,24	-0,83%
5	2.832,27	2.831,35	0,03%
6	3.073,52	3.064,58	0,29%
7	3.225,47	3.253,70	-0,87%
8	3.422,14	3.438,50	-0,48%
9	3.583,42	3.586,22	-0,08%

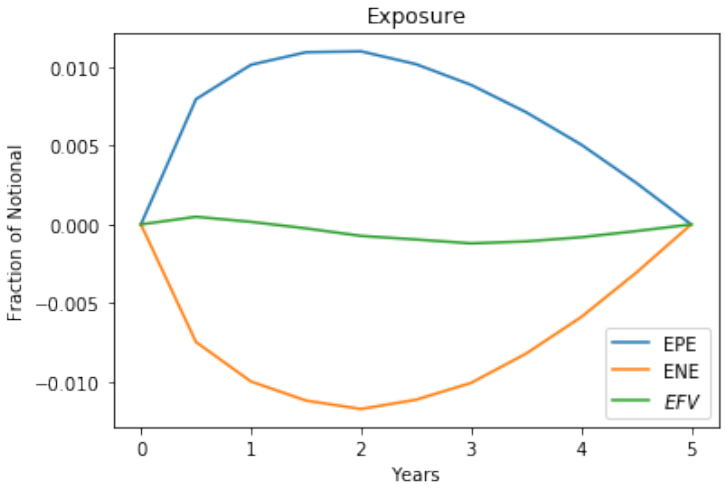
Table 2: Caplet Prices for 100.000 simulations.

After checking the correctness of our implementation, we proceed to evaluate CVA. Here we price a receiver L6M swap with maturity of 5 years, over a 10 million USD notional, with JP Morgan as counterparty. DVA is not considered in this analysis.

Maturity	***Par Rate (%)***	***Notional***	***CVA***	***EPE***	***ENE***
5	1.77	10.000.000	985.18	67.310,08	-71.688,00

Table 3: Swap deal with JP Morgan

Figure 7: Swap Exposure



As can be seen, CVA is relevant even for a high credit quality counterpart as JP Morgan. An important thing to mention is that CVA should be evaluated from portfolio perspective, since exposure to a client arises from all products with them and wrong-way risk has an impact in the valuation process.

10 Conclusions

As shown in the previous sections, developing the things needed for LM model is not a minor task. Care needs to be taken in order to successfully implement each step, but after all we are rewarded with a powerful model. Our results show how significant can be CVA in derivatives and emphasise the importance of having solid pricing equipment for pricing -not so anymore- vanilla instruments.

In our experience, xVA modeling and pricing is one of the hot topics in the banking sector and having the opportunity to work on this is an enormous opportunity to stay at the front in today's markets.

References

[1] Steffen, M. (1990). A simple method for monotonic interpolation in one dimension.

[2] Bootstrapping an interest-rate curve, https://www.implementingquantlib.com /2013/10/chapter-3-part-3-of-n-bootstrapping.html

[3] Crispoldi, C., WiggerPeter, G., Larkin, P. (2015). SABR and LIBOR Market Models in Practice.

[4] Rebonato, R. (2003). Modern Pricing of Interest-Rate Derivatives: The LIBOR Market Model and Beyond.

[5] Brigo, D., Mercurio, F. (2006). Interest Rate Models - Theory and Practice

[6] Jackel, P. (2019). The Practicalities of Libor Market Models