Interest Rate Modelling for Counterparty Risk

Cian O'Duffy

CQF Final Project

July 2018

cian.oduffy@gmail.com

Contents

Abstract

This report outlines the CVA calculation for a five-year USD LIBOR interest rate swap as at 31/12/2017. A 6-month forward rate LIBOR market model calibrated to swaption implied volatilities is used to model the interest rates. A LIBOR-OIS spread is then used to adjust the simulations to carry out the CVA calculation using the dual curve (LOIS) methodology.

A range of relevant methodologies are discussed such as martingale testing, factor reduction, change of numéraire, and the Rebonato method. Additionally, a larger dataset is considered to test the limitations of the model.

Definitions

- A_i – The floating leg of a swap
- d_{ij} an element of the factor reduced dispersion matrix D at row i and column j.
- $\widetilde{d_{ij}}$ an element of the dispersion matrix $\widetilde{D}$ at row i and column j.
- c_{ij} an element of the covariance matrix $C(t)$ at row i and column j.
- t_0 - Calibration time taken as 0 and considered to be 31/12/17 throughout.
- τ = Term increment. Taken as 0.5 years throughout.
- CVA dataset = The reduced size dataset using swaptions that can be priced with a t_0 5-year yield curve.
- Extended dataset - The larger dataset used to investigate the robustness of the model.
- Option Expiry Numéraire – A zero coupon bond that matures at the same time as the option expiry and underlying swap start in the case of a swaption, that is chosen as a numéraire.
- $SR_i = SR_i^n$ - The rate of the fixed leg of a swap starting at time t_i and ending at t_n.
- Swaption term - Time between t_0 and the option expiry (swap beginning).
- Swaption tenor - The length of the swap the swaption is based on.
- Terminal Bond Numéraire – A zero coupon bond that matures at the same time as the final cashflow that is considered, chosen as numéraire. In the case of a swaption the bond will mature at the same time as the underlying swap expiry.

Acronyms

- ATM – At-The-Money
- CVA – Credit Value Adjustment
- EE – Expected Exposure
- LIBOR – London Interbank Offered Rate
- LMM – LIBOR Market Model
- LOIS – LIBOR – OIS Spread
- MtM – Mark to Market
- OIS – Overnight Indexed Swap
- PFE – Potential Future Exposure
- PV – Present Value
- TRF – Trust Region Reflective (a non-linear least squares optimizing algorithm).

Interest Rate Modelling

The first step towards calculating the Credit Value Adjustment (CVA) for an interest rate swap (IRS) is to model the interest rates upon which the swap is valued. The swap to be modelled is a 5-year interest rate swap written on 6-month LIBOR with payments being made semi-annually. For this reason a 6-month forward LIBOR market model (LMM) was chosen to model the rates. This has the benefits of allowing a wide range of possible forward curves, of it being possible to exactly match market prices, and that at each timestep there is a swap payment.

Additionally, it was decided to calibrate the model to European swaptions rather than caps and floors. This was to capture the covariance between terms accurately as well as to explore more complex calibration methods during the project.

Data

To calibrate the LMM, information on the level and volatility of projected forward rates is required. The methodology chosen was to use swap rates (i.e. the fixed rate of swaps) to determine the level of forward rates and swaption implied volatilities to determine the volatility of forward rates. An *n* by *m* swaption is the right but not the obligation to enter into a swap in *n* years that lasts for *m* years at a fixed rate set now. At-the-money (ATM) swaptions are priced so that the projected mark-to-market (MtM) of the swap at the option expiry (and in this project the swap beginning) is zero. This clearly has information on the projected volatility of interest rates over the length of the option due to the asymmetric payoff. Going forward I will describe *n* as the **term** of the swaption and *m* as the **tenor** of the swaption.

It was decided to model USD swaps as at 31/12/17. In order to model a 5-year swap, a 5-year interest rate curve and swaptions of term plus tenor less than 5 years are required. Within

the data there is a 0.5 term swaption but no swaps of term with half year increment after the first year. In order to include the 0.5Y by 5Y swaption within the calibration a swap curve up to 6-years maturity was used so as to include this swaption within the calibration.

The approach taken was to only include the swaptions that could be priced with the forward rates that a 5-year swap requires. This was so the volatility function (discussed in the calibration section) could be fitted to the most relevant volatility datapoints for a 5-year swap. The data used is shown below:

***Table 1**: USD Swap rates of terms 0.5 years to 5 years as at 31/12/17. CVA dataset. Ticker USSW1 CMPN Curncy onwards. (Bloomberg)*

Term	0.5	1	2	3	4	5	6
Rate	1.754%	1.899%	2.078%	2.169%	2.211%	2.244%	2.277%

***Table 2**: USD Swaption implied volatilities for ATM European swaptions as at 31/12/17. CVA dataset. Ticker: USSV011 SMKO Curncy onwards. (Bloomberg)*

		Tenor (years)				
		1	2	3	4	5
Term (Years)	0.5	16.24%	19.73%	22.54%	22.64%	22.76%
	1	19.35%	22.24%	23.88%	24.54%	
	2	23.07%	25.03%	25.83%		
	3	26.54%	27.68%			
	4	28.28%				

Interpolation

A full yield curve is required to implement the LMM and thus LIBOR rates at 6-month intervals were required. This required the swap curve to be interpolated to give swap rates at 6-month intervals. Svensson fitting was initially considered which has the benefit of returning an integrable, differentiable, exponentially smoothed function, however this could not fit the yield curve points to a high enough accuracy. Rather, linear interpolation implementing the following expression from (Hagan & West, 2005) was used:

$$s(t) = \frac{t-t_i}{t_{i+1}-t_i} s(t_{i+1}) + \frac{t_{i+1}-t}{t_{i+1}-t_i} s(t_i) \quad (1)$$

where

$$t_i < t < t_{i+1}.$$

This approach has the benefit of exactly fitting the market datapoints. The implementation for fitting svensson parameters can be seen in svensson.py, and for linear interpolation of the swap rates in the bootstrapping.py file.

Bootstrapping

Discounting the cashflows by LIBOR (Brigo & Mercurio, 2006) show that the forward swap rate of a swap $SR_i^N(t)$ starting at t_i and with final payment at t_N for constant payment frequency τ at t can be expressed in terms of zero coupon bond prices $P_i(t)$ which pay 1 at time t_i :

$$SR_i^N(t) = \frac{P_i(t) - P_N(t)}{\tau \sum_{j=i+1}^{N} P_j(t)}. \quad (2)$$

When considering the t_0 swap curve this becomes:

$$SR_0^N(t_0 = 0) = \frac{1 - P_N(t_0)}{\tau \sum_{j=1}^{N} P_j(t_0)}. \quad (3)$$

This is because ATM swaps are issued with the floating and fixed legs equal (and thus a PV of zero).

Rearranging this expression for a range of swap rates known at t_0 (as at date) of different maturities we can calculate the discount curve:

$$P(t_0; t_0) = 1$$

$$P(t_0; T_1) = \frac{1}{1 + \tau SR_0^1(t_0)}$$

$$P(t_0; T_N) = \frac{1 - \tau \sum_{i=1}^{N-1} P(t_0, T_i)}{1 + \tau SR_0^N(t_0)} \quad (4)$$

where

$$P_N(t) = P(t;\, T_N).$$

From the discount curve we determine the t_0 simple forward rates with the following expression:

$$f_i(t_0) = f(t_0, T_i, T_{i+1}) = \frac{\frac{P(t_0,T_i)}{P(t_0,T_{i+1})}-1}{\tau}. \quad (5)$$

Additionally, the discount curve can be calculated from simple forward rates as:

$$P_{i+1}(t) = \prod_{k=0}^{i}(1+\tau f_k(t))^{-1}. \quad (6)$$

From the swap curve we have derived our starting simple LIBOR 6-month forward rates and starting discount curve. This has been implemented both in the 'bootstrapping.py' file and LOIS.xlsx sheet for checking.

OIS Discounting

Since the financial crisis it has become apparent that LIBOR, which represents unsecured lending rate between banks, is not the risk-free rate. Consequently, there has been a move to discounting the cashflows of swaps by the Overnight Indexed Swap (OIS) rate rather than LIBOR as this is viewed as a better representation of the risk-free rate. As swaps are increasingly collateralized with a motivation to make them credit risk free, market practice is to discount them by the risk-free rate. So, a LIBOR swap using dual curve OIS discounting will have its floating payments equal to the rolling LIBOR rate multiplied by the notional but discounted by the OIS discount curve.

This has very little effect when bootstrapping the LIBOR rates from the swap rates, as in order for the present value (PV) of the swap to be zero the projected cashflows of both legs at t_0 are closely matched so the values that are discounted are very small. To illustrate, the LOIS.xlsx

spreadsheet details the bootstrapping of the OIS discount curve from OIS swaps where the floating payments and discount rates are both based on the OIS rate. The OIS discount curve is then used to discount the cashflows of a 5-year LIBOR swap and the fixed LIBOR swap rate is adjusted with a solver to set the value of the swap to zero. This results in a tiny **0.06 bps** change to the 5-year ATM swap rate.

Unsurprisingly OIS discounting only makes a difference when the cashflows that are being discounted are not close to zero. In this project that means the Expected Exposure (EE) and swaption values. As dual curve discounting is a relatively new development much of the LMM literature, most notably the analytical solutions, do not cover it. Consequently, the approach taken is to calibrate the LMM using the LIBOR discount curve but calculating the CVA using the LOIS adjusted discount curve.

Black Formula For Swaptions

(Gatarek, Bachert, & Maksymiuk, 2006) state the Black formula for a swaption with strike *K*, expiring at t_i written on a swap expiring at t_N as:

$$Swaption_i^N = \left(\sum_{j=i+1}^{N} \tau\, P(0,T_i)(SR_i^N(0)N(d_1) - KN(d_2)\right), \tag{7}$$

where

$$d_1 = \frac{ln\left(\frac{SR_i^N(0)}{K}\right) + \tau\frac{(\sigma_i^N)^2}{2}}{(\sigma_i^N)\sqrt{\tau}},$$

$$d_2 = d_1 - \sigma_i^N\sqrt{\tau}.$$

$N(\quad) = \text{Cumulative distribution function for the standard normal distribution.}$

ATM swaptions are priced with the strike *K* equal to the t_0 forward swap rate $SR_i^N(0)$. Thus, given the swaption price from the market and the t_0 yield curve one can invert (7) to determine the swaption volatility σ_i^N that the price implies.

Bloomberg quotes both prices and implied volatilities for swaptions. The approach taken in this project is to calibrate to market implied volatilities rather than swaption prices, however the prices are viewed to see the effect of OIS discounting.

The LIBOR Market Model

The following introduction to the LMM is based on work in (Jaeckel, 2002). The LMM is a family of interest rate models that evolve simply compounded forward rates through time. Within this framework we assume that each of *n* spanning forward rates f_i evolves according to the stochastic differential equation:

$$\frac{df_i}{f_i} = \mu_i(\boldsymbol{f}, t)dt + \sigma_i(t)d\widetilde{W_i}, \tag{8}$$

where $\widetilde{W_i}$ is a wiener process for the term index *i* and forward rate f_i. Correlation between terms is incorporated through a covariance matrix *C(t)* where:

$$\mathbb{E}\left[d\widetilde{W_i}d\widetilde{W_j}\right] = \rho_{ij}dt \tag{9}$$

$$c_{ij}(t) = \sigma_i(t)\,\sigma_j(t)\rho_{ij}. \tag{10}$$

By decomposing *C(t)* into a pseudo-square root $\widetilde{D}$ such that

$$C = \widetilde{D}\widetilde{D}^{\intercal}, \tag{11}$$

we can express the diffusion in (8) as a dispersion matrix multiplied by *n* independent standard Wiener processes so:

$$\sigma_i(t)d\widetilde{W_i} = \sum_j \widetilde{d_{ij}}dW_j. \tag{12}$$

The approach taken in this project is to decompose *C* using cholesky decomposition by calling the SciPy 'linalg.cholesky' function. Refer to (The SciPy Community, 2015) for further details on the algorithm.

In addition we can express independent wiener processes as random normal variate draws such that:

$$dW_j \sim \phi(0,1)\sqrt{\delta t} \tag{13}$$

where $\phi(0,1)$ is a random number taken from the standard normal distribution.

At this point we introduce numéraire denominated valuation. Here we select a tradeable security $N(t)$ that can be readily evaluated. From this we establish an equivalent martingale measure in which the values of all tradeable securities relative to the numéraire are martingales. We can then evaluate the present value of a cashflow in the future by evaluating its expectation relative to the numéraire in the future and multiplying it by the numéraire value today.

The value $v(t)$ of a security that pays a sequence of conditional cashflows $c_j(t_j)$ can then be calculated as:

$$v(t) = N(0)\mathbb{E}_{\mathbb{Q}_N}\left[\sum_j \frac{c_j(t_j)}{N(t_j)}\right]. \tag{14}$$

Another way of thinking about this is that the ratio between any tradeable asset and the numéraire within the numéraire denominated equivalent martingale measure is a martingale and thus time invariant.

We now consider a zero-coupon bond P_N paying 1 at time t_N as numéraire. As discount bonds are assumed to be traded assets it follows from (14) that:

$$\mathbb{E}_{\mathbb{Q}_N}\left[d\left(\frac{f_i P_{i+1}}{P_N}\right)\right] = 0. \tag{15}$$

$$\mathbb{E}_{\mathbb{Q}_N}\left[d\left(\frac{P_i}{P_N}\right)\right] = 0, \ \forall i = 0, \dots, n, \tag{16}$$

By considering these restrictions within the LMM equation (8) (Jaeckel, 2002) shows that for constant term increment τ the drift $\mu_i(\boldsymbol{f}(t), t)$ must satisfy the following relationship in order for the projection of forward rates to be within the numéraire denominated equivalent martingale measure:

$$\mu_i(\boldsymbol{f}(t), t) = \begin{cases} -\sigma_i \sum_{k=i+1}^{N-1} \frac{f_k(t)\tau}{1+f_k(t)\tau}\sigma_k\rho_{ik} \ \ for \ i < N-1, t \le T_i \\ 0 \ for \ i = N-1, t \le T_{N-1} \\ \sigma_i \sum_{k=N}^{i} \frac{f_k(t)\tau}{1+f_k(t)\tau}\sigma_k\rho_{ik} \ for \ i \ge N, t \ \le T_N \end{cases} \tag{17}$$

It is clear that once we have determined the volatility of the forward rates everything else is determined, as the drift is set by (17).

(Jaeckel, 2002) shows that when you assume piecewise constant drift you can carry out the integration over the time step δt analytically. Additionally, if we choose a terminal bond as numéraire P_N which pays 1 at the final cashflow of whichever product we wish to evaluate then we can discard the bottom two drift terms in (17), as for all forward rates of relevance $i \leq N - 1$.

Putting this together with (8) gives the following euler scheme in logarithmic coordinates:

$$f_i^{constant\ drift}(t + \delta t) = f_i exp\left[\left(-\tau\sigma_i \sum_{k=i+1}^{N} \frac{f_k \sigma_k \rho_{ik}}{1+\tau f_k} - \frac{c_{ii}}{2}\right)\delta t + \sum_{j=1}^{m} \widetilde{a_{ij}} dW_j\right] \quad (18)$$

Pseudocode 1: *Log-euler forward rate projection scheme for terminal bond numéraire P_N*

foreach simulation m:

instantiate $\boldsymbol{f}(t_0)$

instantiate $\widetilde{D}$

for n in range 0 to number_of_projection_periods:

for *i* in range *n*+1 to *N:*

$\text{diffusion}_i = \sqrt{\tau} \sum_{j=1}^{N} \phi(0,1)\ \widetilde{d_{ij}}$

$\mu_i(t_n, m) = -\sigma_k \sum_{k=i+1}^{N} \frac{f_k(t_n,m)\sigma_k \rho_{ik}}{1+\tau f_k(t_n,m)}$

$f_i(t_n + \tau, m) = f_i(t_n, m) exp\left(\left(\mu_i(t_n, m) - \frac{\sigma_i^2}{2}\right)\tau + diffusion_i\right)$

Note the volatility parameters that determine the matrix $\widetilde{D}$ are set in the next section.

Calibration

The purpose of calibration within the LMM framework is to determine the volatility terms in equation (8). In the following section we discuss the Rebonato method, which is an analytical solution that determines a volatility function for forward rate volatility that is calibrated to swaption implied volatilities. The following derivation is based on (Jaeckel, 2002) with adjustments that are specific to this project.

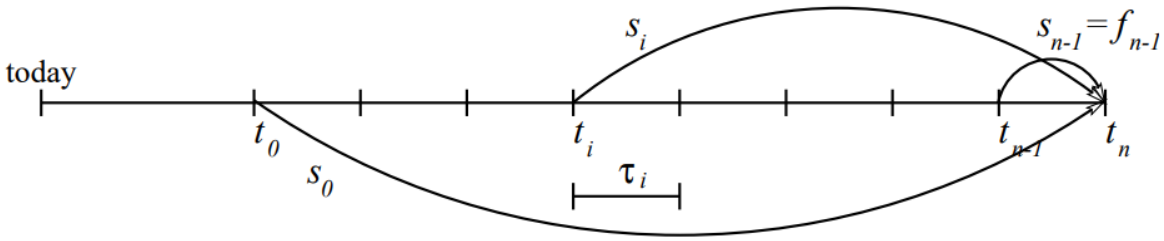

Figure 1: *A forward starting swap. (Jaeckel, 2002)*

For a notional of 1, a swap starting at t_i and ending at t_n, the swap rate SR_i is defined as:

$$s_i = SR_i = \frac{A_i}{B_i} \tag{19}$$

$$A_i = \tau \sum_{j=i}^{n-1} P_{j+1} f_j \tag{20}$$

$$B_i = \tau \sum_{j=i}^{n-1} P_{j+1} \tag{21}$$

The covariance between the relative swap rate terms becomes:

$$C_{ij}^{SR} = Cov\left[\frac{dSR_i}{SR_i}, \frac{dSR_j}{SR_j}\right] = \mathbb{E}_{\mathbb{Q}_n}\left[\frac{dSR_i}{SR_i} \cdot \frac{dSR_j}{SR_j}\right] - \mathbb{E}_{\mathbb{Q}_n}\left[\frac{dSR_i}{SR_i}\right] \mathbb{E}_{\mathbb{Q}_n}\left[\frac{dSR_j}{SR_j}\right]$$

$$\mathbb{E}_{\mathbb{Q}_n}\left[\frac{dSR_i}{SR_i}\right] = \mathbb{E}_{\mathbb{Q}_n}\left[\frac{dSR_j}{SR_j}\right] = 0$$

$$C_{ij}^{SR} = \mathbb{E}_{\mathbb{Q}_n}\left[\frac{\left(\sum_{k=0}^{n-1} \frac{\partial SR_i}{\partial f_k} df_k\right)\left(\sum_{l=0}^{n-1} \frac{\partial SR_j}{\partial f_l} df_l\right)}{SR_i SR_j}\right] = \mathbb{E}_{\mathbb{Q}_n}\left[\frac{\left(\sum_{k=0,l=0}^{n-1,n-1} \frac{\partial SR_i}{\partial f_k} \frac{\partial SR_j}{\partial f_l}\right)}{SR_i SR_j} df_k df_l\right]$$

$$= \frac{\left(\sum_{k=0,l=0}^{n-1,n-1} \frac{\partial SR_i}{\partial f_k}\frac{\partial SR_j}{\partial f_l}\right)}{SR_i SR_j} f_k f_l \mathbb{E}_{\mathbb{Q}_n}\left[\frac{df_k}{f_k}\frac{df_l}{f_l}\right]$$

so:

$$C_{ij}^{SR} = \sum_{k=0,l=0}^{n-1,n-1} \frac{\partial SR_i}{\partial f_k}\frac{f_k}{SR_i} C_{kl}^{FRA} \frac{f_l}{SR_j}\frac{\partial SR_j}{\partial f_l} \text{ where } C_{kl}^{FRA} = \mathbb{E}_{\mathbb{Q}_n}\left[\frac{df_k}{f_k}\frac{df_l}{f_l}\right].$$

Let the forward rate to swap rate mapping be:

$$Z_{ik}^{FRA\to SR} = \frac{\partial SR_i}{\partial f_k}\frac{f_k}{SR_i},$$

which gives the matrix equation:

$$C^{SR} = Z^{FRA\to SR} C^{FRA} (Z^{FRA\to SR})^{\intercal}.$$

In this project we only consider swaps where the fixed and floating payments occur simultaneously with the same frequency so it is possible to determine a simple expression for the mapping Z as follows:

$$\frac{\partial SR_i}{\partial f_k} = \frac{\partial}{\partial f_k}\left(\frac{A_i}{B_i}\right) = \frac{1}{B_i}\frac{\partial A_i}{\partial f_k} - \frac{A_i}{B_i^2}\frac{\partial B_i}{\partial f_k}.$$

$$\frac{\partial A_i}{\partial f_k} = \tau\left(P_{k+1} + \sum_{j=i}^{n-1} \frac{\partial P_{j+1}}{\partial f_k} f_j\right) \quad (22)$$

$$\frac{\partial B_i}{\partial f_k} = \tau\left(\sum_{j=i}^{n-1} \frac{\partial P_{j+1}}{\partial f_k}\right) \quad (23)$$

Differentiating (6) gives:

$$\frac{\partial P_{i+1}}{\partial f_k} = -\frac{\tau}{(1+\tau f_k)} P_{i+1}.\, \mathbf{1}_{\{\boldsymbol{k\geq i}\}}. \quad (24)$$

Substituting (24) into (22) and (23) gives:

$$\frac{\partial A_i}{\partial f_k} = \tau\left(P_{k+1} - \frac{\tau}{(1+\tau f_k)}\sum_{j=k}^{n-1} P_{j+1} f_j\right) = \tau\left(P_{k+1} - \frac{A_k}{(1+\tau f_k)}\right)$$

$$\frac{\partial B_i}{\partial f_k} = -\frac{\tau^2}{(1+\tau f_k)}\sum_{j=k}^{n-1} P_{j+1} = -\frac{\tau B_k}{(1+\tau f_k)}$$

$$\frac{\partial SR_i}{\partial f_k} = \tau\left\{\frac{P_{k+1}}{B_i} - \frac{1}{1+\tau f_k}.\frac{A_k}{B_i} + \frac{1}{1+\tau f_k}\frac{A_i B_k}{B_i^2}\right\}.\mathbf{1}_{\{k\geq i\}} \quad (25)$$

$$Z_{ik}^{f\to s} = \tau\left\{\frac{P_{k+1} f_k}{A_i} - \frac{f_k}{A_i(1+\tau f_k)}.\frac{B_i A_k}{B_i} + \frac{f_k}{(1+\tau f_k)}\frac{A_i B_k}{A_i B_i}\right\}.\mathbf{1}_{\{k\geq i\}}$$

$$Z_{ik}^{f\to s} = \tau\left[\frac{P_{k+1} f_k}{A_i} + \frac{f_k(A_i B_k - B_i A_k)}{A_i B_i(1+\tau f_k)}\right].\mathbf{1}_{\{k\geq i\}} \quad (26)$$

Unfortunately, the mapping *Z* matrix involves the state of the yield curve at any point in time, however (Jackel & Rebonato, 2002) show that if you make some simplifying assumptions there is an approximate analytical solution for the volatility of the swap rate using the time zero yield curve mapping:

$$\widehat{\sigma_{SR_i}} = \sqrt{\sum_{k=i,l=i}^{n-1} Z_{ik}^{f\to s}(0).\frac{\int_t^T \sigma_k(t')\sigma_l(t')\rho_{kl}dt'}{T-t}.Z_{il}^{f\to s}(0)} \quad (27)$$

We now choose the time-homogenous instantaneous volatility function for forward rate f_i proposed by (Rebonato, 1998):

$$\sigma_i(t) = \left(a + b(t_i - t)e^{-c(t_i-t)} + d\right).\mathbf{1}_{\{t<t_i\}}, \quad (28)$$

and the time static instantaneous correlation between forward rates proposed in (Jaeckel, 2002) :

$$\rho_{ij} = e^{-\beta(t_i - t_j)}. \quad (29)$$

$$\beta = 0.1.$$

As the correlation is time-independent and the volatility function only depends on the difference between the term and time the forward rate covariance's can by analytically integrated

across time with the indefinite integral given by (Jaeckel, 2002). Integrating between $t_0=0$ and the swaption expiry T and setting the k parameters to 1 gives the following expression:

$$\int_0^T \rho_{ij}\sigma_i(t)\sigma_j(t)\,dt = \frac{e^{-\beta|t_i-t_j|}}{4c^3}\cdot\begin{pmatrix} 4ac^2d\left[e^{c(T-t_j)}+e^{c(T-t_i)}-e^{-ct_j}-e^{-ct_i}\right]+4c^3d^2T \\ -4bcd\left(e^{c(T-t_j)}[c(T-t_i)-1]+e^{-ct_i}[ct_i+1]\right) \\ -4bcd\left(e^{c(T-t_j)}\left[c(T-t_j)-1\right]+e^{-ct_j}\left[ct_j+1\right]\right) \\ +e^{c(2T-t_i-t_j)}\begin{pmatrix}2a^2c^2+2abc\left[1+c(t_i+t_j-2T)\right] \\ +b^2\begin{bmatrix}1+2c^2(T-t_i)(T-t_j) \\ +c(t_i+t_j-2T)\end{bmatrix}\end{pmatrix} \\ -e^{-c(t_i+t_j)}\begin{pmatrix}2a^2c^2+2abc\left[1+c(t_i+t_j)\right] \\ +b^2\begin{bmatrix}1+2c^2t_it_j \\ +c(t_i+t_j)\end{bmatrix}\end{pmatrix}\end{pmatrix} \quad (30)$$

Volatility Optimization

Given (30) and evaluating $Z^{FRA\to SR}(0)$ from the t_0 yield curve using (26) provides everything we need to evaluate $\widehat{\sigma_{SR_i}}$ in (27). However this is the wrong way round. We know the swaption implied volatilities from **Table 2** and must determine are the *a*, *b*, *c* and *d* parameters in (28) for the instantaneous forward rate volatilities that return these swaption implied volatilities. Additionally, as there are 15 swaption volatilities to calibrate to with 4 volatility parameters it is impossible to capture all volatilities exactly in any stochastic system. We make the assumption that the option expires at the same time as the swap begins so $T = t_i$ and the swap expires at t_n. For a given t_0 yield curve:

$$\widehat{\sigma_{SR_i}} = \widehat{\sigma_{SR_i^n}}(a,b,c,d)$$

The approach taken is to use the python least squares optimizer set to the default Trust Region Reflective (TRR) to minimize the sum of the squared differences between the estimate $\widehat{\sigma_{SR_i^n}}$ and market value $\sigma_{SR_i^n}^{market}$. The pseudocode below details the optimization process as far as

the code written for this project is concerned. For further information on the workings of the optimizer refer to (The SciPy community, 2018).

***Pseudocode 2**: Optimisation of volatility parameters for analytical solution to swaption implied volatilities*

intitialize $Z^{FRA \to SR}(0)$

set initial guess *a, b, c, d* = 0.5

sum = 0

while least_squares optimizer parameters are above success tolerances do:

foreach $\sigma_{SR_i^n}^{market}$ do:

compute $\widehat{\sigma_{SR_i^n}}(a, b, c, d)$ from (27)

foreach $\sigma_{SR_i^n}^{market}$ do:

sum += $\left(\sigma_{SR_i^n}^{market} - \widehat{\sigma_{SR_i^n}}(a, b, c, d)\right)^2$

pass sum to optimizer

receive optimizer next *a, b, c, d* guess parameters

set *a, b, c, d*

return *a ,b, c, d* parameters

For each row in **Table 2** we fix the swap rate start index *i* and vary the swap expiry index *n*. Additionally, as *n* is absolute, $T_n = t_i + \text{tenor}$ as referred to in **Table 2**.

(26) indexes elements under the assumption that *n* is fixed however as we iterate through **Table 2** *n* varies. $Z^{FRA \to SR}(0)$ becomes a three-dimensional matrix with the third-dimension referring to the swap expiry index *n*. This is computed before the parameter optimization in order to increase solution speed.

This calibration approach has the benefit of determining the forward rate volatilities from the market implied swaption volatilities without a single monte-carlo simulation. As a result, the solution performance is orders of magnitude greater than a calibration optimization that involved monte-carlo which, if a larger dataset was consider (such as shown in **Table 7**) would be impracticable.

If the forward rate volatilities are known the calculated swaption implied volatilities can be substituted into the Black swaption formula (7) to produce a swaption price without the need for a single monte-carlo simulation.

The following figure and tables summarize the results from the Rebonato method calibration for the CVA dataset.

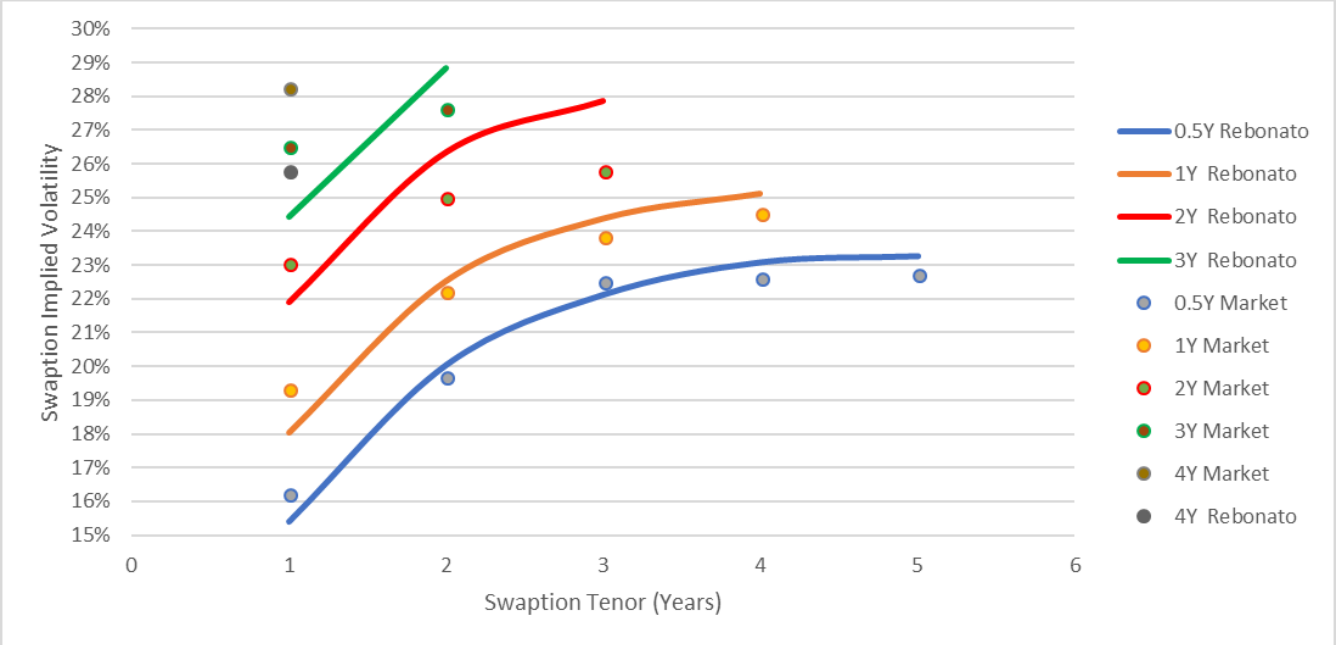

***Figure 2**: Rebonato method versus market ATM swaption implied volatilities for swaptions of varying term. CVA dataset*

***Table 3**: Rebonato method swaption implied volatilities. CVA dataset*

		Tenor				
		1	2	3	4	5
Term	0.5	15.41%	20.05%	22.13%	23.09%	23.27%
	1.0	18.02%	22.53%	24.39%	25.13%	
	2.0	21.89%	26.34%	27.86%		
	3.0	24.42%	28.84%			
	4.0	25.81%				

***Table 4:** Market versus Rebonato method ATM swaption implied volatility relative differences. CVA dataset*

		Tenor				
		1	2	3	4	5
Term	0.5	5.4%	-1.6%	1.8%	-1.9%	-2.2%
	1.0	7.4%	-1.3%	-2.1%	-2.4%	
	2.0	5.4%	-5.0%	-7.3%		
	3.0	8.7%	-4.0%			
	4.0	9.6%				

***Table 5**: Rebonato method volatility parameters. CVA dataset*

Parameter	*a*	*b*	*c*	*d*
Value	*2.812*	*0.310*	*0.080*	*-2.704*

To further investigate the robustness of this analytical solution and calibration approach an extended dataset was considered. The results are as follows:

***Table 6**: USD Swap rates of terms 0.5 years to 50 years as at 31/12/17. Extended dataset. Ticker USSW1 CMPN Curncy onwards. (Bloomberg)*

Term	Rate
0.5	1.75%
1	1.90%
2	2.08%
3	2.17%
4	2.21%
5	2.24%
6	2.28%
7	2.31%
8	2.34%
9	2.37%
10	2.40%
11	2.42%
12	2.44%
13	2.46%
14	2.47%
15	2.49%
16	2.50%
17	2.51%
18	2.52%
19	2.53%
20	2.53%
25	2.54%
30	2.54%
35	2.53%
40	2.52%
50	2.49%

***Table 7**: USD Swaption implied volatilities for ATM European swaptions as at 31/12/17 Extended Dataset. Ticker: USSV011 SMKO Curncy onwards. (Bloomberg)*

		Tenor (Years)											
		1	2	3	4	5	6	7	8	9	10	15	20
Term (Years)	1	19.4%	22.2%	23.9%	24.5%	24.5%	24.5%	24.4%	24.3%	24.2%	24.1%	23.2%	22.6%
	2	23.1%	25.0%	25.8%	25.9%	26.0%	25.9%	25.7%	25.6%	25.4%	25.3%	23.9%	23.3%
	3	26.5%	27.7%	27.4%	27.1%	26.9%	26.6%	26.4%	26.1%	25.9%	25.7%	24.2%	23.4%
	4	28.3%	28.5%	27.9%	27.5%	27.1%	26.8%	26.5%	26.3%	26.1%	25.9%	24.4%	23.5%
	5	28.4%	28.3%	27.9%	27.6%	27.3%	26.9%	26.6%	26.3%	26.1%	25.9%	24.3%	23.3%
	7	27.3%	27.2%	26.9%	26.5%	26.2%	25.9%	25.7%	25.4%	25.2%	25.0%	23.6%	22.5%
	10	25.6%	25.1%	25.0%	24.8%	24.7%	24.5%	24.3%	24.1%	23.9%	23.7%	22.0%	21.3%
	15	23.1%	22.4%	22.0%	21.7%	21.6%	21.6%	21.6%	21.6%	21.6%	21.6%	20.5%	20.0%
	20	21.1%	20.4%	20.1%	19.9%	20.0%	20.0%	20.0%	20.0%	20.0%	19.9%	19.0%	19.0%

***Table 8**: Rebonato method swaption implied volatilities. Extended dataset.*

		Tenor (Years)											
		1	2	3	4	5	6	7	8	9	10	15	20
Term (Years)	1	19.9%	24.0%	25.3%	25.6%	25.4%	25.1%	24.6%	24.1%	23.5%	23.0%	21.0%	19.6%
	2	22.3%	26.3%	27.3%	27.4%	27.2%	26.8%	26.3%	25.8%	25.3%	24.8%	22.8%	21.5%
	3	23.5%	27.3%	28.2%	28.3%	28.0%	27.5%	27.1%	26.5%	26.1%	25.6%	23.7%	22.5%
	4	24.0%	27.7%	28.5%	28.5%	28.1%	27.7%	27.2%	26.7%	26.2%	25.8%	24.0%	22.8%
	5	24.0%	27.6%	28.3%	28.2%	27.9%	27.5%	27.0%	26.5%	26.0%	25.6%	23.9%	22.7%
	7	23.4%	26.7%	27.3%	27.3%	26.9%	26.5%	26.1%	25.6%	25.2%	24.8%	23.2%	22.1%
	10	21.9%	25.0%	25.5%	25.5%	25.2%	24.8%	24.4%	24.0%	23.6%	23.3%	21.8%	20.8%
	15	19.8%	22.6%	23.1%	23.1%	22.8%	22.5%	22.2%	21.8%	21.5%	21.2%	20.0%	19.1%
	20	18.4%	21.0%	21.5%	21.4%	21.2%	21.0%	20.7%	20.4%	20.1%	19.8%	18.7%	17.9%

***Table 9**: Rebonato method versus market swaption implied volatility relative differences. Extended dataset*

		Tenor (Years)											
		1	2	3	4	5	6	7	8	9	10	15	20
Term (Years)	1	-2.8%	-7.4%	-5.5%	-4.0%	-3.5%	-2.4%	-0.8%	1.0%	2.9%	4.6%	10.4%	15.5%
	2	3.5%	-4.7%	-5.4%	-5.5%	-4.4%	-3.5%	-2.3%	-0.8%	0.6%	1.9%	4.4%	8.0%
	3	12.9%	1.3%	-2.9%	-4.1%	-3.9%	-3.5%	-2.6%	-1.7%	-0.6%	0.5%	2.1%	4.0%
	4	17.9%	2.8%	-1.8%	-3.2%	-3.6%	-3.1%	-2.5%	-1.5%	-0.3%	0.7%	1.7%	3.0%
	5	18.5%	2.5%	-1.2%	-2.3%	-2.2%	-2.1%	-1.5%	-0.7%	0.2%	1.0%	1.8%	2.5%
	7	16.8%	2.0%	-1.5%	-2.7%	-2.6%	-2.2%	-1.5%	-0.7%	0.1%	0.8%	1.9%	2.0%
	10	17.0%	0.6%	-2.1%	-2.5%	-2.0%	-1.3%	-0.5%	0.3%	1.1%	1.7%	1.0%	2.2%
	15	16.7%	-0.8%	-4.6%	-5.8%	-5.2%	-3.9%	-2.4%	-0.9%	0.6%	1.9%	2.9%	4.9%
	20	14.9%	-2.8%	-6.3%	-7.1%	-6.0%	-4.7%	-3.4%	-2.0%	-0.6%	0.5%	1.7%	5.9%

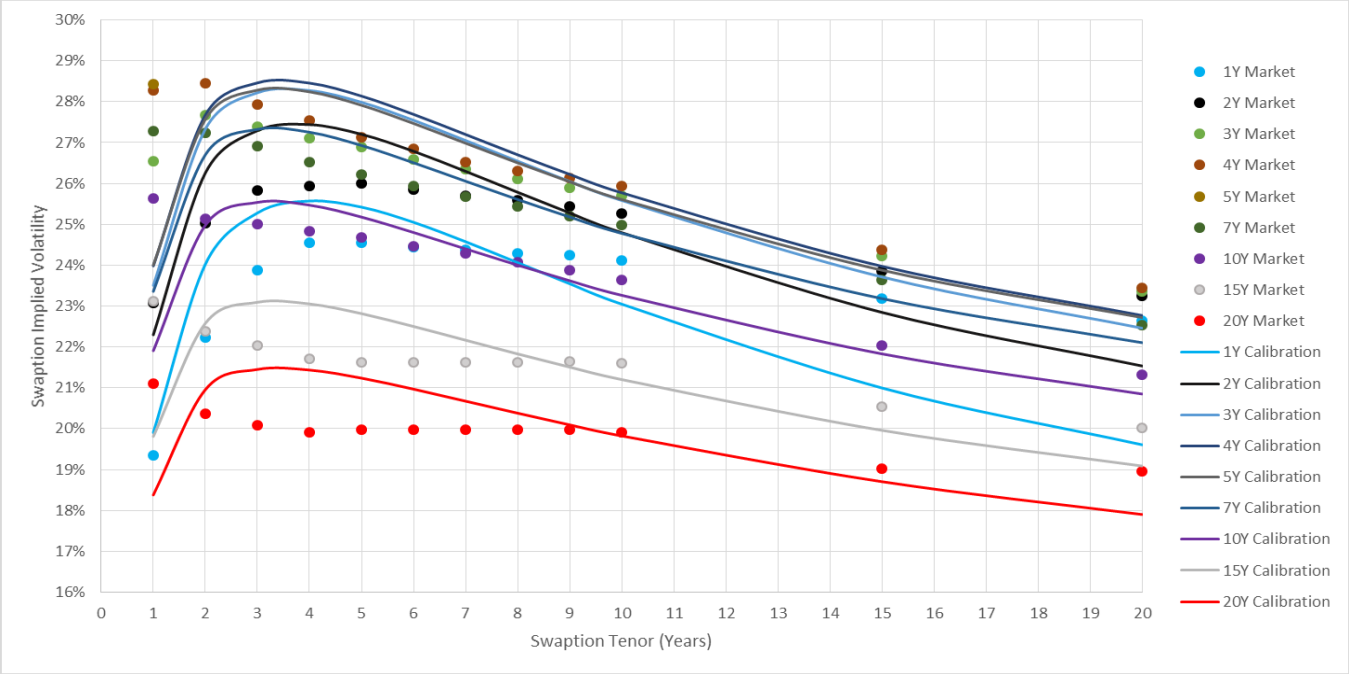

***Figure 3**: Rebonato method versus market swaption implied volatilities for swaptions of varying term. Extended dataset*

***Table 10:** Rebonato method volatility parameters. Extended dataset*

Parameter	*a*	*b*	*c*	*d*
Value	*0.0137*	*0.0792*	*0.3392*	*0.0842*

In both the CVA dataset and the extended dataset the volatility function when calibrated from the equation (27) understates the volatility for shorter length swaps. This can be reduced by calibrating a volatility parameter vector ***k*** as discussed in (Jackel & Rebonato, 2002). This method provides an additional number of parameters to calibrate to the data equal to the number of forward rates in your starting curve.

In this project the approach taken is to only calibrate the *a*, *b*, *c*, *d* parameters but to reduce the data points to those swaptions relevant to the CVA swap in question to achieve the best fit to the relevant data. This is for clarity and to reduce the dimensionality of the optimizer problem.

Overall the Rebonato method has achieved a good fit to the data with high performance given the simplicity of the volatility function.

Simulation

Pseudocode 1 was implemented in python with the calibration discussed above and monte-carlo forward rate simulations were generated for both the CVA dataset and the extended dataset.

Table 11: Statistics across all forward rates during a monte-carlo run with 50,000 simulations. Extended dataset

	Min	Lower Quartile	Median	Mean	Upper Quartile	Max
Forward Rate	0.06%	1.66%	0.98%	1.46%	2.62%	102.68%

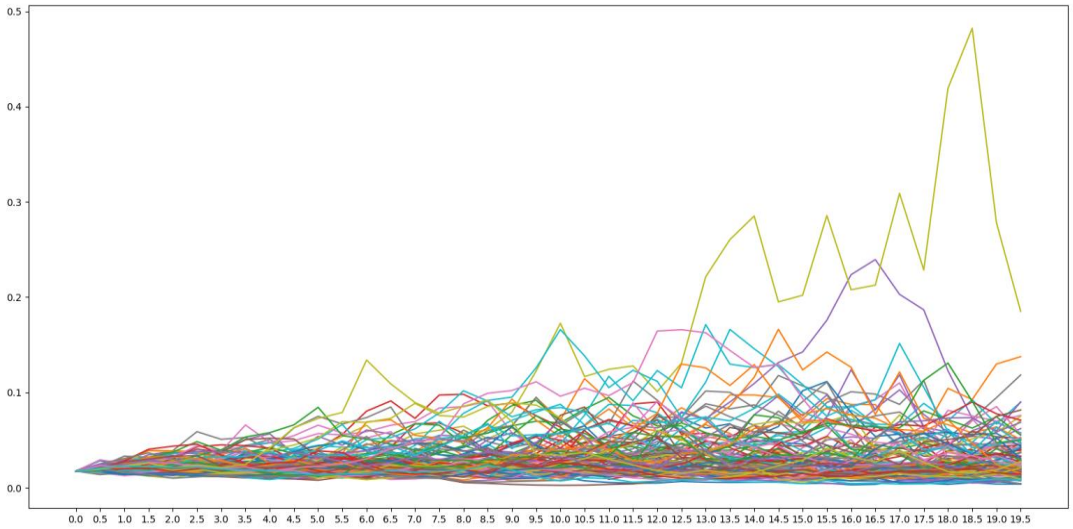

Figure 4: *Cash rate simulations for a 100 simulation run. Extended dataset.*

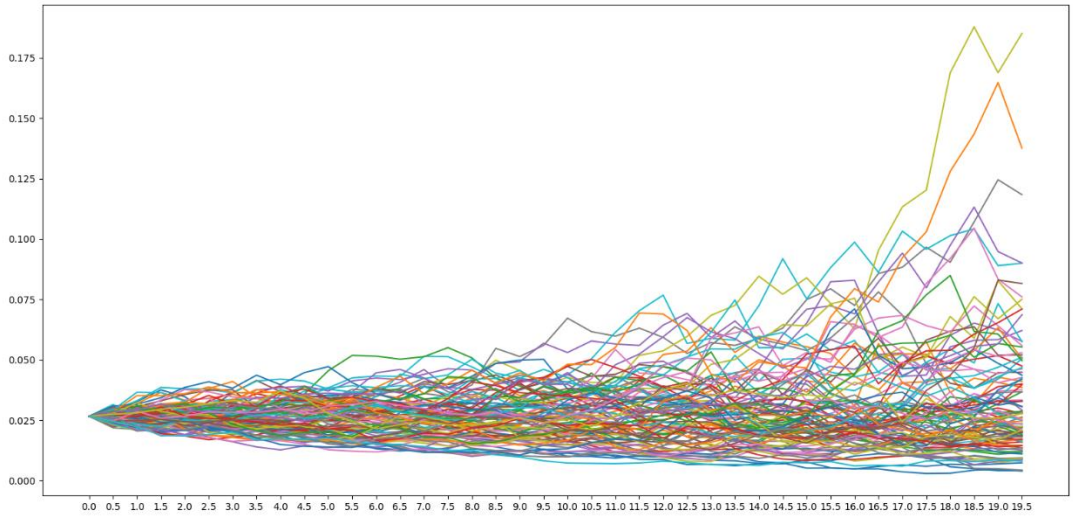

Figure 5: *100 simulations for* $f_{39.5}(t)$. *Extended dataset.*

As the LMM involves the evolution of entire forward rate curves it is difficult to draw any conclusions when looking at statistics across all terms. What one can see is that forward rates are floored at zero. This is because of the log-normal euler scheme does not allow negative forward rates. Post financial crisis negative interest rates have shifted from being considered impossible, to being considered unlikely. For this reason, shifted LMM schemes are sometimes used which allow for negative rates.

In addition very high forward rates are possible. This is because there is no mean-reversion in the risk neutral LMM which pulls rate simulations back to the mean when they drift away. One could argue that USD forward rates of 100% are impossible however, for the generation of real world simulations a real world LMM should be implemented. Both of these model observations seem reasonable for the purpose of CVA calculation of an IRS in the risk neutral measure.

Monte-Carlo Model Validation

Before carrying out the CVA calculation the calibrated LMM was validated in two ways. The first was to price zero coupon bonds in order to carry out martingale tests and the second was to price the swaptions upon which the model was calibrated to assess the goodness of fit of the volatility. Model validation is important because without it we cannot be sure of any of our model results and we cannot know the limitations of the model. Additionally, if we cannot price zero coupon bonds how can we hope to price more complex products?

Martingale Tests

Consider a zero-coupon bond $P_i(t)$ that pays 1 at time t_i within the equivalent martingale measure of numéraire $P_N(t)$. As $P_i(t)/P_N(t)$ is a martingale:

$$\frac{P_i(0)}{P_N(0)} = \mathbb{E}_{\mathbb{Q}_N}\left[\frac{P_i(t)}{P_N(t)}\right]. \tag{31}$$

Therefore, for the drift to have been correctly calculated and thus (14) valid:

$$\frac{\mathbb{E}_{\mathbb{Q}_N}\left[\frac{P_i(t)}{P_N(t)}\right]}{\frac{P_i(0)}{P_N(0)}} = 1 \tag{32}$$

This ratio will be referred to hereafter as the martingale ratio. (32) can be further simplified by considering that $P_i(t_i) = 1$ by definition so:

$$\frac{\mathbb{E}_{\mathbb{Q}_N}\left[\frac{1}{P_N(t_i)}\right]}{\frac{P_i(0)}{P_N(0)}} = 1 \tag{33}$$

This is a helpful result as the only stochastic element comes from $P_N(t_i)$, reducing the error that comes from working with multiple stochastic variables.

The following table and figures detail the martingale tests carried out on both the CVA and extended datasets. A martingale test is where you calculate the ratio in (32) and see how far the value is from 1.

***Table 12**: Martingale ratios for zero coupon bonds of different maturities at their respective expiries using a 5-year zero coupon bond as numéraire. CVA dataset calibration.*

Term	0.5	1	1.5	2	2.5	3	3.5	4	4.5
Martingale Ratio	1.001	1.001	1.002	1.002	1.002	1.002	1.002	1.001	1.001

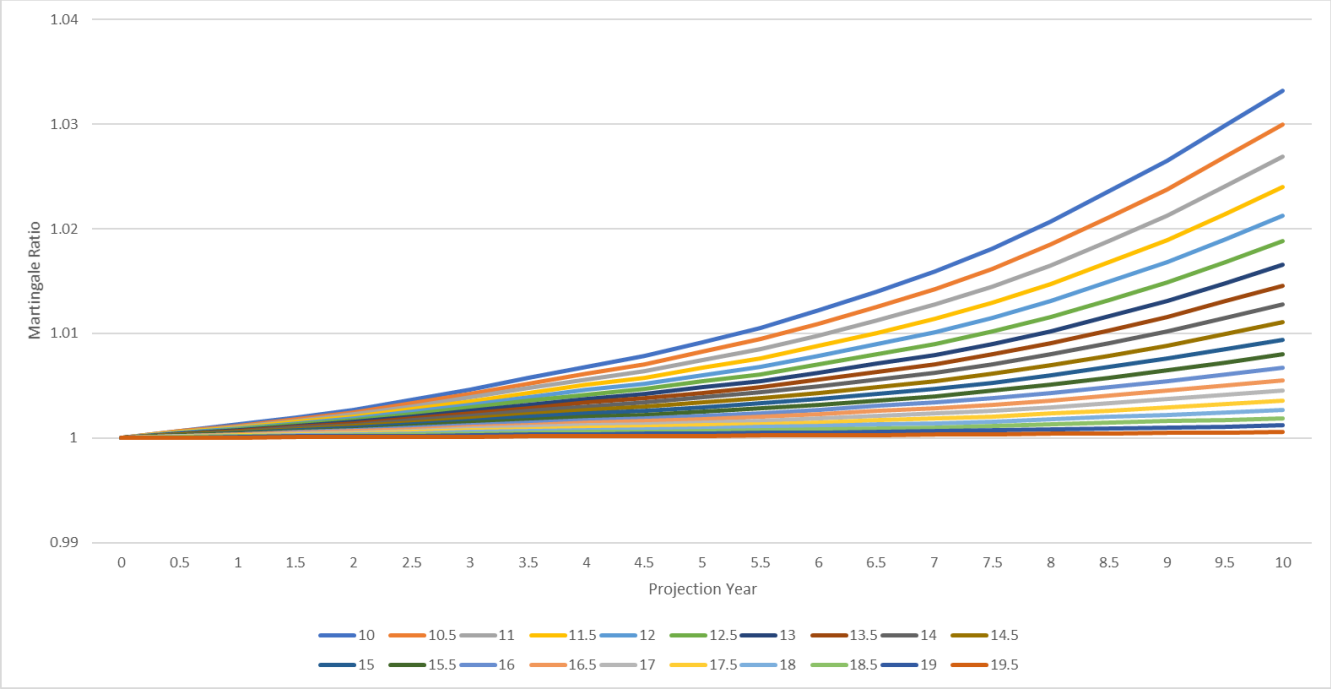

***Figure 6**: Projection of martingale ratio using for a range of zero coupon bonds of different maturity. The numéraire chosen is a 20-year zero coupon bond. Extended dataset calibration.*

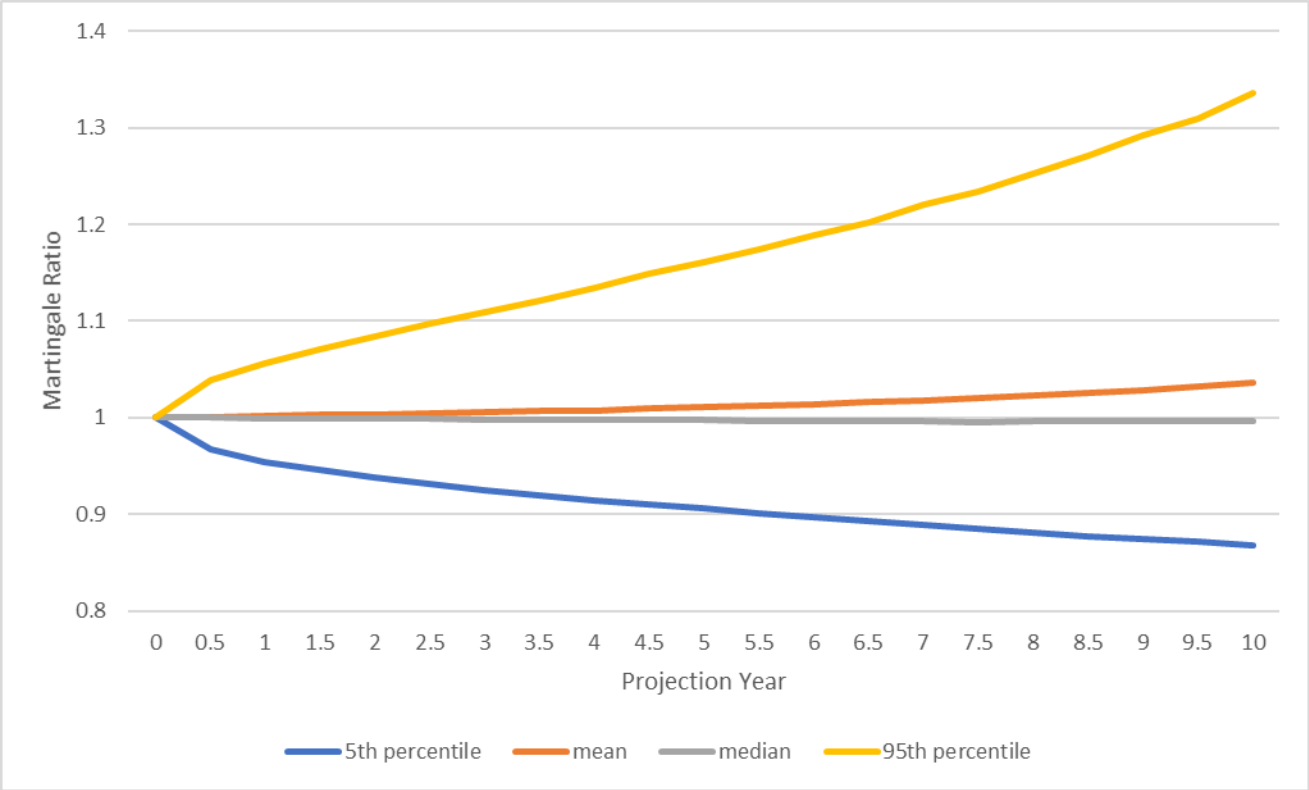

***Figure 7**: Projection of 10-year zero coupon bond martingale ratio statistics using a 20-year zero coupon bond numéraire equivalent martingale measure. Extended dataset calibration.*

While the results above look very close to 1, the differences are significant as will be seen in the next section. One adjustment that was trialed to see if it could improve the martingale ratios was the predictor-corrector scheme discussed in (Jaeckel, 2002). This scheme attempts to make up for the piecewise constant drift assumption by averaging the drift across time steps. This only caused a marginal improvement in the martingale ratios and so was not used, however the implementation can be seen in the code.

It is not clear where the error in the drift is coming from. An explanation could be discretization error caused by large timesteps but most likely it is a coding bug in the calculation of the drift.

Swaption Pricing

(Brigo & Mercurio, 2006) show that the payoff $v_i^n(t_i)$ of an ATM payer swaption that expires at t_i on a swap that begins at t_i and expires at t_n with strike K is:

$$v_i(t_i) = \tau(SR_i^n(T_i) - K)^+ \sum_{j=i+1}^{n} P(T_i, T_j), \tag{34}$$

and conversely the payoff of an ATM receiver swaption is:

$$u_i(t_i) = \tau\big(K - SR_i^n(T_i)\big)^+ \sum_{j=i+1}^{n} P(T_i, T_j). \tag{35}$$

Choosing the terminal bond numéraire $P_N(t)$ with expiry $t_N = t_n$ from (14) we evaluate the t_0=0 value of the payer swaption as:

$$v_i^N(0) = P_N(0)\mathbb{E}_{\mathbb{Q}_N}\left[\frac{\tau\big(SR_i^N(T_i)-K\big)^+ \sum_{j=i+1}^{N} P(T_i,T_j)}{P_N(t_i)}\right]. \tag{36}$$

The expectation in the above equation is evaluated from the LMM forward rate monte-carlo simulations. $SR_i^N(T_i)$ is evaluated using the simulated forward rates to calculate the forward discount factors and substituting this into equation (2). Once $v_i^N(0)$ is determined we substitute it into the Black swaption formula (7) and use SciPy's 'minimize_scalar' function set to the default Brent algorithm to invert the equation to determine the volatility that the price $v_i^N(0)$ implies. For information on the minimize scalar function see (The SciPy Community, 2018)

For 50,000 simulations this gave the following implied swaption volatilities for payer and receiver swaptions respectively.

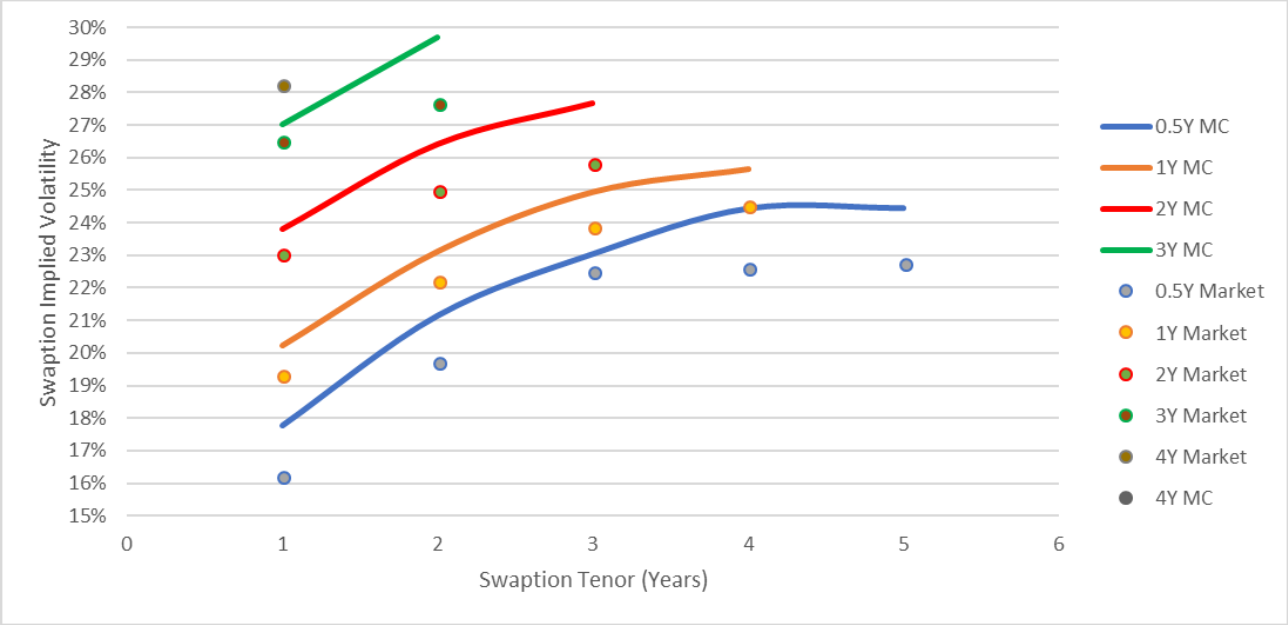

Figure 8: *Market verus monte-carlo* ***payer*** *swaption implied volatilities of varying term with no volatility adjustment. CVA dataset.*

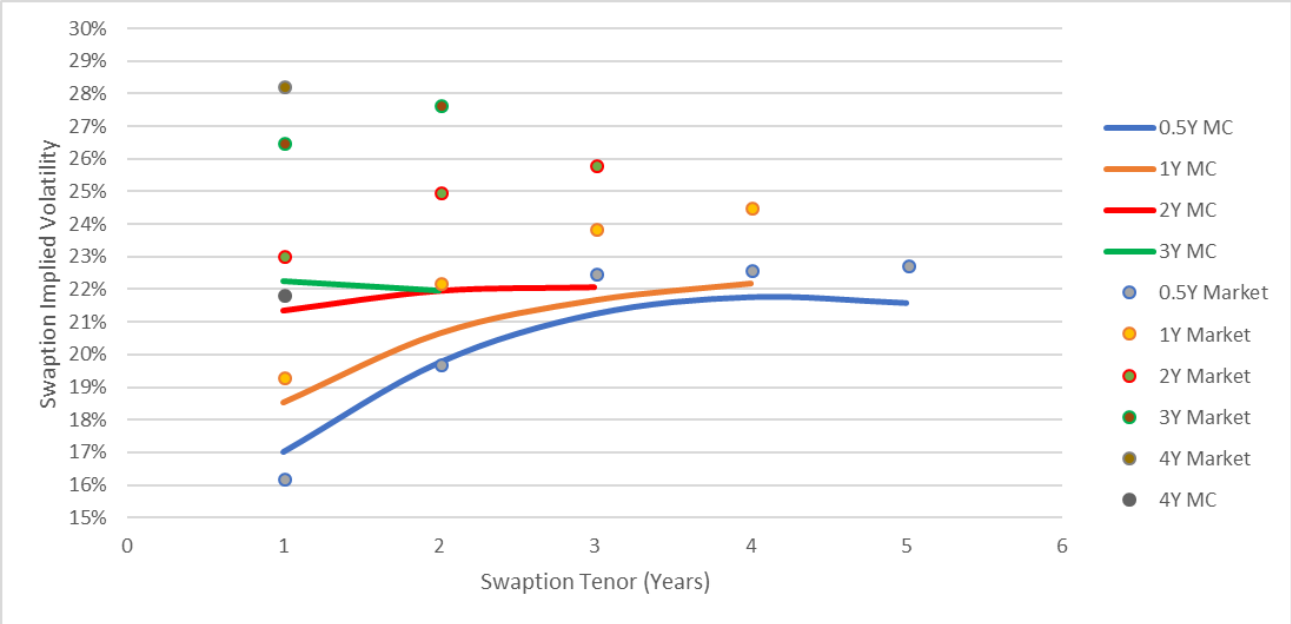

Figure 9*: Market verus monte-carlo* ***receiver*** *swaption implied volatilities of varying term with no volatility adjustment. CVA dataset.*

Swaption put-call parity states that if you are long a payer swaption, and short a receiver swaption on the same underlying swap this is equivalent to being long the underlying swap. As ATM swaptions are priced such that the underlying swap has a PV of zero when it begins:

$$ATM\ Payer\ Swaption_i^N(t_0) = ATM\ Receiver\ Swaption_i^N(t_0). \quad (37)$$

This also means the implied volatility of ATM payer and receiver swaptions on the same underlying with the same expiry should be the same. **Figure 8** and **Figure 9** show that this is clearly this is not case.

The reason for this is the positive error in the drift discussed in the previous section. $SR_i^N(T_i)$ is increasingly being overstated, with its mean at option expiry becoming increasingly larger than the strike *K*. This decreases the implied volatility of receiver swaptions in two ways. First, those simulations with a non-zero payoff have their payoff reduced as $K - SR_i^N(T_i)$ is smaller than if there were no error. Second, those simulations with a non-zero payoff have their payoffs more heavily discounted by higher forward rates. This causes the collapse of the volatility term structure seen in **Figure 9**.

Similarly, while the positive drift error increases the payoff of payer swaptions, this payoff is more heavily discounted by higher forward rates and so the impact is less pronounced for payer swaptions as the effects oppose each other.

If the martingale ratios stated in **Table 12** are multiplied by the forward discount factors in order to adjust for the positive drift there are changes in the payer and receiver swaption volatilities of the order required to explain these differences. Instead a different method of adjusting for the drift error is considered the next section.

Error Adjustment

The ultimate goal of the project is to carry out the CVA calculation for a **payer** swap, i.e. a swap that pays floating LIBOR and receives fixed. We therefore target the volatility of payer swaptions going forward. One way to do this is to adjust the volatility downwards such that the implied payer swaption volatility fits the market data. This is not ideal as we are introducing an error in the volatility to adjust for an error in the drift however, because the Expected Exposure ($EE(t)$) only considers the positive mark-to-market of the swap this does make some sense. This is because the payoff of a payer swaption on an n-payment swap is the same as the exposure of a payer swap at the time when it has n-payments remaining.

It is for this reason that we introduce the k^{MC} volatility parameter. (28) then becomes:

$$\sigma_i^{MC}(t) = k^{MC}\left(a + b(t_i - t)e^{-c(t_i - t)} + d\right).\mathbf{1}_{\{t<t_i\}}. \quad (38)$$

This adjustment factor is similar to the $\boldsymbol{k}$ vector discussed in (Jaeckel, 2002), except it is a scalar that is calibrated on the monte-carlo simulations rather than during the Rebonato method.

k^{MC} is calculated using the scipy minimize scalar function with the objective function being the sum of the squared differences between the market swaption implied volatilities and their respective monte-carlo calculated swaption implied volatilities. This adjustment is only carried out for the CVA dataset.

Computationally this method is very effective as the calibration is split into two optimization problems. The first (discussed in the calibration section) is a 4-dimensional optimization in which the objective function is an analytical function. The second is a 1-dimensional scalar optimization problem where the objective function utilizes monte-carlo simulation.

One could be tempted to instead solve this as a 5-dimensional optimization problem where the objective function utilizes monte-carlo simulation. This would take far longer to solve and be less robust.

Results

The following tables detail the swaption implied volatilities calculated through monte-carlo simulation for both datasets. In all cases 50,000 simulations were used and the values were determined using an ATM payer swaption payoff under the terminal bond numéraire. No adjustment was made to the extended dataset.

***Table 13**: k^{MC} adjustment factor calculated for the CVA dataset.*

k^{MC}	0.955

***Table 14**: Payer Swaption implied volatilities calculated from monte-carlo. CVA dataset.*

		Tenor				
		1	2	3	4	5
Term	0.5	17.0%	20.3%	22.4%	23.2%	23.3%
	1	19.0%	22.0%	23.7%	24.4%	
	2	22.6%	24.9%	26.2%		
	3	25.7%	28.0%			
	4	28.5%				

***Table 15**: Market versus monte-carlo payer swaption implied volatility relative differences.*

		Tenor				
		1	2	3	4	5
Term	0.5	-5%	-3%	1%	-2%	-2%
	1	2%	1%	1%	0%	
	2	2%	1%	-1%		
	3	3%	-1%			
	4	-1%				

Figure 10: *Market verus monte-carlo swaption implied volatilities of varying term. CVA dataset.*

Table 16*:Payer swaption implied volatilities from monte-carlo. Extended dataset.*

		Tenor											
		1	2	3	4	5	6	7	8	9	10	15	20
Term	1	21.9%	23.3%	23.8%	23.8%	23.3%	23.0%	22.0%	21.0%	20.1%	19.3%	16.3%	14.5%
	2	23.9%	25.1%	25.5%	25.5%	24.5%	23.6%	22.6%	22.0%	20.7%	20.2%	17.1%	15.2%
	3	26.0%	26.7%	26.7%	26.2%	25.1%	24.5%	23.7%	23.0%	21.8%	21.0%	17.7%	15.8%
	4	27.6%	28.0%	27.9%	27.2%	26.5%	25.2%	24.1%	23.3%	22.7%	21.2%	18.2%	16.1%
	5	29.2%	28.8%	28.8%	27.6%	26.4%	25.7%	24.7%	23.8%	22.7%	22.1%	18.7%	16.6%
	7	29.6%	30.0%	29.4%	28.4%	27.6%	26.1%	25.5%	24.3%	23.1%	22.5%	19.4%	17.6%
	10	30.7%	29.9%	29.8%	28.9%	27.2%	26.7%	25.2%	25.0%	23.7%	23.1%	20.2%	18.2%
	15	30.0%	29.5%	29.2%	28.0%	26.5%	25.8%	25.6%	25.1%	24.1%	23.0%	20.4%	18.6%
	20	29.3%	29.0%	28.0%	27.7%	26.3%	25.5%	24.9%	24.3%	23.4%	22.7%	20.5%	19.4%

***Table 17**: Market versus monte-carlo payer swaption implied volatility relative differences*

		Tenor											
		1	2	3	4	5	6	7	8	9	10	15	20
Term	1	-11%	-5%	0%	3%	6%	6%	11%	16%	21%	25%	43%	57%
	2	-3%	0%	1%	2%	6%	10%	14%	17%	23%	25%	40%	53%
	3	2%	4%	3%	3%	7%	9%	11%	14%	19%	22%	37%	48%
	4	2%	2%	0%	1%	2%	7%	10%	13%	15%	22%	34%	45%
	5	-2%	-2%	-3%	0%	3%	4%	7%	10%	15%	17%	30%	40%
	7	-8%	-9%	-9%	-7%	-5%	-1%	1%	5%	9%	11%	22%	28%
	10	-16%	-16%	-16%	-14%	-9%	-8%	-4%	-4%	1%	2%	9%	17%
	15	-23%	-24%	-25%	-22%	-18%	-16%	-15%	-14%	-10%	-6%	1%	8%
	20	-28%	-30%	-28%	-28%	-24%	-22%	-20%	-18%	-15%	-12%	-7%	-2%

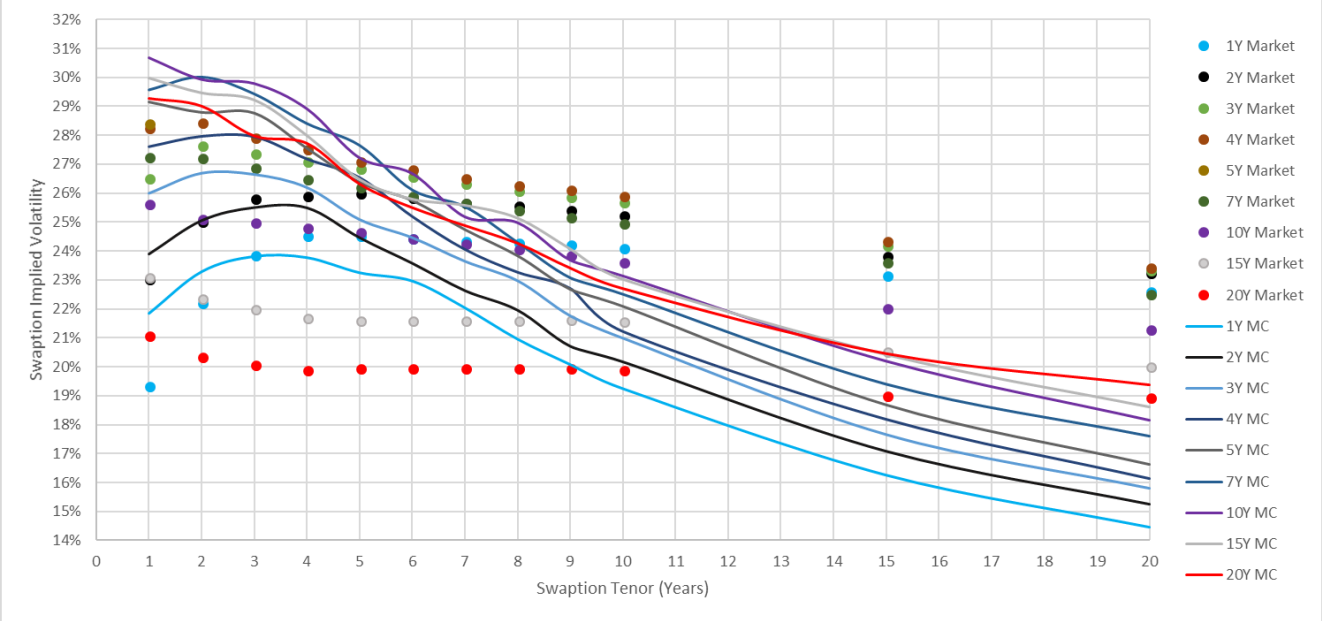

***Figure 11:** Market versus monte-carlo payer swaption implied volatilities of varying term. Extended dataset*

Following adjustment the monte-carlo simulations fit the payer swaption implied volatilities to the market data for the CVA dataset as well as the Rebonato method. The same cannot be said for the extended dataset where the monte-carlo calculated swaption volatilities do not fit the market data nearly as well as the Rebonato method. This is due to the positive drift error which affects the extended dataset more due to the greater swaption expiries that were considered.

Change of numéraire

The fact is that the terminal zero coupon bond is not the most appropriate numéraire for pricing swaptions. Consider instead a zero coupon bond $P_N(t)$ that expires at t_i the swaption expiry and swap start time such that $N = i$ and $t_i = t_N$. Today's swaption value $v_i^N(0)$ from (36) becomes:

$$v_i^N(0) = P_i(0)\mathbb{E}_{\mathbb{Q}_i}\left[\frac{\tau\left(SR_i^N(T_i)-K\right)^+\sum_{j=i+1}^N P(T_i,T_j)}{P_i(t_i)}\right]. \quad (39)$$

However, $P_i(t_i) = 1$ by definition so the payer swaption value under the option expiry zero coupon bond numéraire equivalent martingale measure becomes:

$$v_i^N(0) = P_i(0)\mathbb{E}_{\mathbb{Q}_i}\left[\tau(SR_i^N(T_i) - K)^+\sum_{j=i+1}^N P(T_i,T_j)\right]. \quad (40)$$

Clearly this is a simpler expression for evaluating swaptions as there is no need to evaluate a stochastic $P_N(t_i)$ term. It does require use of the upper drift term in equation (17) as the term of the forward rates that are required to price the swaption are higher than the term of the numéraire bond:

$$\mu_i(\boldsymbol{f}(t),t) = \sigma_i\sum_{k=N}^i \frac{f_k(t)\tau}{1+f_k(t)\tau}\sigma_k\rho_{ik}\ for\ i \geq N\ , t\ \leq T_N. \quad (41)$$

Using this choice of numéraire, and coding the drift for the other side of the numéraire produces a better fit to the data for the extended dataset. Unfortunately this fit is still not as good as that of the Rebonato method:

***Table 18**: Payer swaption implied volatilities from monte-carlo using the option expiry bond numéraire. Extended dataset.*

		Tenor											
		1	2	3	4	5	6	7	8	9	10	15	20
Term	1	25.8%	27.0%	27.5%	27.4%	27.0%	26.5%	25.9%	25.4%	24.8%	24.3%	22.6%	22.1%
	2	28.2%	28.9%	29.0%	28.6%	28.0%	27.4%	26.7%	26.0%	25.4%	24.9%	23.3%	22.8%
	3	29.3%	29.6%	29.3%	28.7%	28.0%	27.3%	26.6%	25.9%	25.3%	24.8%	23.3%	22.9%
	4	29.6%	29.6%	29.1%	28.5%	27.7%	27.0%	26.2%	25.6%	25.1%	24.7%	23.2%	23.0%
	5	30.1%	29.9%	29.3%	28.6%	27.8%	27.0%	26.4%	25.8%	25.3%	24.8%	23.5%	23.3%
	7	29.3%	28.9%	28.2%	27.5%	26.8%	26.1%	25.5%	24.9%	24.4%	24.1%	23.1%	23.1%
	10	28.3%	27.9%	27.3%	26.7%	25.9%	25.3%	24.8%	24.3%	24.0%	23.7%	23.0%	23.2%
	15	26.4%	26.0%	25.4%	24.9%	24.4%	24.0%	23.6%	23.3%	23.0%	22.8%	22.5%	23.0%
	20	24.7%	24.3%	23.9%	23.4%	23.0%	22.6%	22.3%	22.1%	21.9%	21.8%	21.8%	22.3%

***Table 19**: Market versus monte-carlo payer swaption implied volatility relative differences using the option expiry bond as numéraire. Extended dataset.*

		Tenor											
		1	2	3	4	5	6	7	8	9	10	15	20
Term	1	-25%	-18%	-13%	-10%	-9%	-8%	-6%	-4%	-2%	-1%	2%	2%
	2	-18%	-13%	-11%	-9%	-7%	-5%	-4%	-2%	0%	1%	2%	2%
	3	-9%	-6%	-6%	-6%	-4%	-3%	-1%	1%	2%	3%	4%	2%
	4	-5%	-4%	-4%	-3%	-2%	0%	1%	3%	4%	5%	5%	2%
	5	-6%	-5%	-5%	-4%	-2%	-1%	1%	2%	3%	4%	4%	0%
	7	-7%	-6%	-5%	-3%	-2%	-1%	1%	2%	3%	4%	2%	-2%
	10	-10%	-10%	-8%	-7%	-5%	-3%	-2%	-1%	0%	0%	-4%	-8%
	15	-13%	-14%	-13%	-13%	-11%	-10%	-8%	-7%	-6%	-5%	-9%	-13%
	20	-14%	-16%	-16%	-15%	-13%	-12%	-11%	-10%	-9%	-9%	-13%	-15%

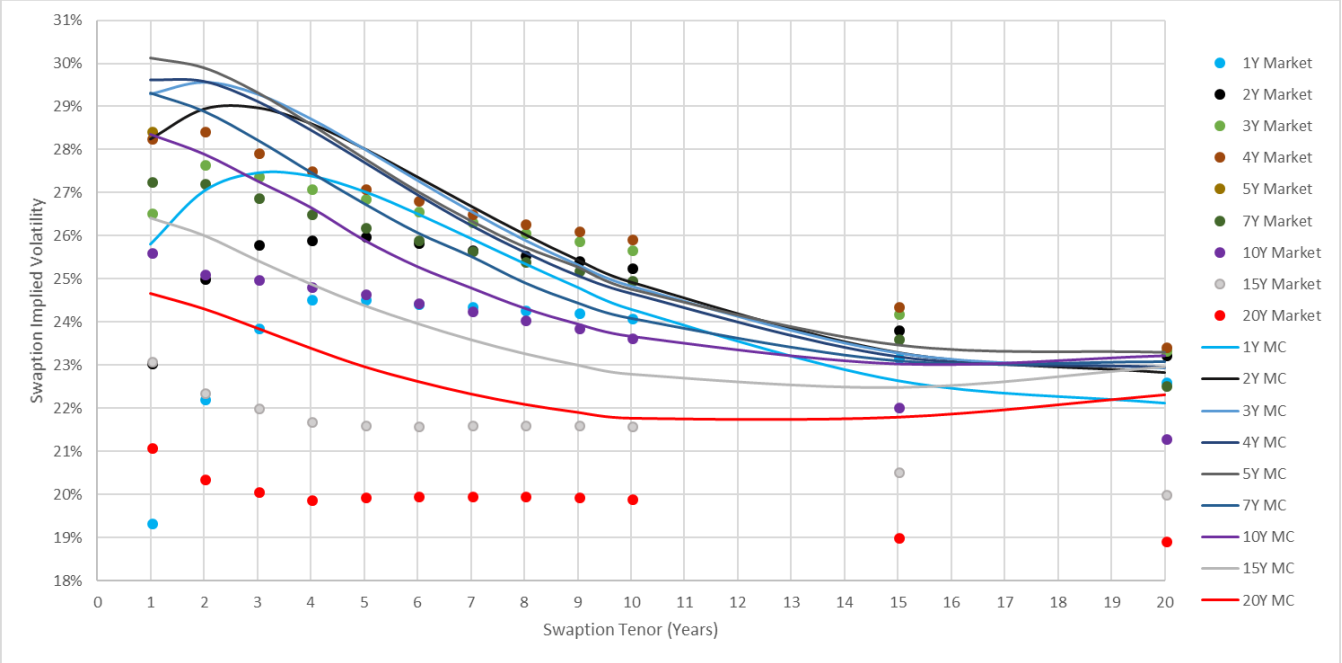

***Figure 12**:Market versus monte-carlo payer swaption implied volatilities using the option expiry bond as numéraire. Extended dataset.*

This shows the benefits of choosing the right numéraire for each situation. In general, this is the numéraire that reduces the number of stochastic variables that are required and thereby reducing the error in whichever simulation approach we have chosen.

In this case the terminal bond is a good choice of numéraire for evaluating swaps during the CVA calculation and so was used for pricing the validation swaptions. This way we can determine the model limitations for pricing swaption volatility and therefore know the model limitations for the swap volatility during the CVA calculation.

In addition, this suggests the error in the drift is due to a coding bug as the ‘change of numéraire’ simulations used separately coded drift in which the drift does not appear to have the same positive error.

Factor Reduction

(Jaeckel, 2002) shows that it is possible to drive the evolution of the *n* forward rates with fewer underlying independent standard Wiener process than there are forward rates, say only *m* of them. In this case the coefficient matrix $\widetilde{D} \in \mathbb{R}^{n\times n}$ in (11) becomes $D \in \mathbb{R}^{n\times m}$. In order to ensure the magnitude of the volatility is unchanged during the factor reduction D must be rescaled such that:

$$d_{ij} = \widetilde{d_{ij}} \sqrt{\frac{c_{ii}}{\sum_{k=1}^{m} \widetilde{d_{ik}^2}}}. \tag{42}$$

This method is useful when pricing derivatives on investment portfolios. In this case multiple yield curves must be simulated such as rate curves for different economies. The correlation between each curve must also be captured. In order to implement the LMM the covariance matrix must be inverted often using nearest correlation matrix algorithms such as those discussed in (Higham, 2002). If the correlation matrices for each yield curve are not factor reduced then the covariance matrix can become very large, making matrix inversion increasingly difficult. Examples are risk neutral Economic Scenario Generators (ESG) used by the insurance industry when calculating guaranteed annuity options based on investment portfolios which may include a mix of global equities and credit products.

Additionally, factor reduction can lead to significantly reduced computation time and memory. In this project a factor reduction has been tested where the number of independent Wiener processes has been reduced to three. This produced the following implied volatilities:

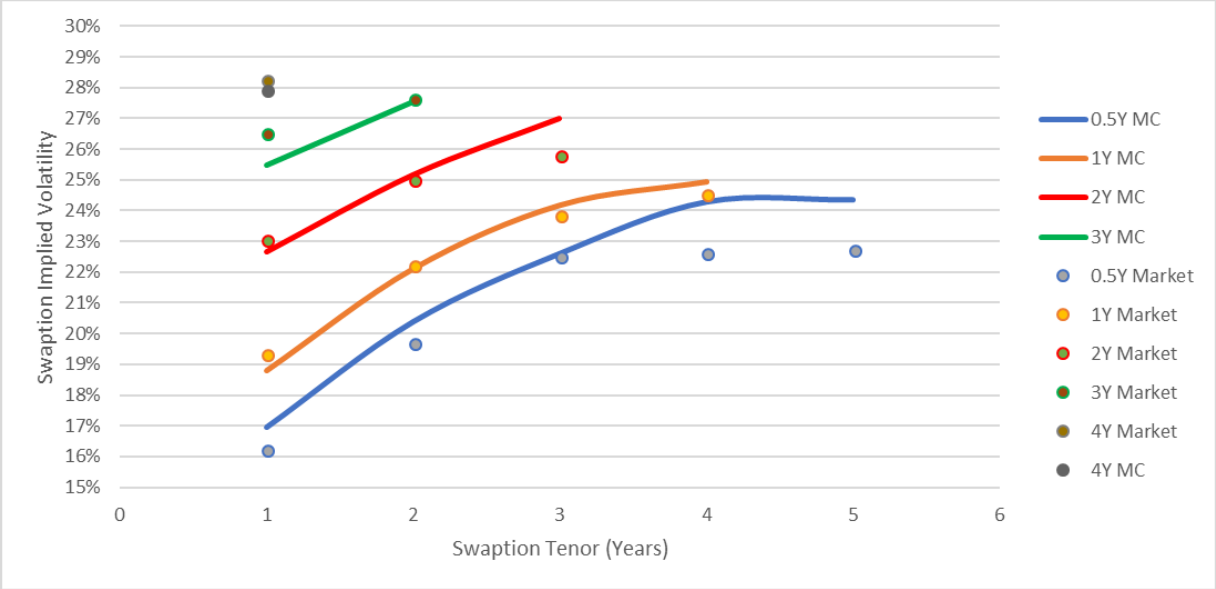

Figure 13: *Market versus monte-carlo payer swaption implied volatilities using a 3-factor, factor reduced dispersion. CVA dataset.*

Table 20: *Market versus monte carlo payer swaption implied volatility relative differences using a 3-factor, factor reduced dispersion. CVA dataset*

		Tenor				
		1	2	3	4	5
Term	0.5	-4.3%	-3.2%	-0.3%	-6.8%	-6.5%
	1.0	3.0%	0.6%	-1.2%	-1.6%	
	2.0	1.8%	-0.5%	-4.3%		
	3.0	4.1%	0.5%			
	4.0	1.1%				

The general effect is to cause a slightly worse fit to the market data than without the factor reduction. This effect can be mitigated by incorporating the factor reduction within the Rebonato method. In this project we are modelling a single LIBOR curve so there is no benefit to factor reduction. Thus, the entire covariance matrix will be used to price the CVA swap.

Swaption Market Premiums

The Bloomberg market convention in quoting swaption premiums is to quote the price of a straddle. Holding a swaption straddle is to hold a payer swaption and receiver swaption on the same underlying swap with the same expiry. ATM swaptions are priced such that receiver swaption prices are equal to payer swaption prices. Unfortunately, that is not the case in our monte-carlo swaption pricing due to the positive drift, so instead we calculate the swaption straddle price by doubling the payer swaption price.

***Table 21**: USD ATM European Swaption Straddle Premiums as at 31/12/17. CVA Dataset. Ticker USSP0F1 CMPL Curncy onward. (Bloomberg)*

		Tenor				
		1	2	3	4	5
Term	0.5	0.19%	0.47%	0.75%	1.07%	1.43%
	1	0.34%	0.79%	1.26%	1.72%	
	2	0.60%	1.27%	1.95%		
	3	0.87%	1.65%			
	4	1.02%				

***Table 22:** ATM payer swaption doubled premiums calculated from Monte-Carlo. CVA Dataset*

		Tenor				
		1	2	3	4	5
Term	0.5	0.20%	0.50%	0.83%	1.15%	1.43%
	1	0.34%	0.78%	1.26%	1.73%	
	2	0.57%	1.24%	1.94%		
	3	0.77%	1.68%			
	4	0.96%				

***Table 23**: USD Swaption implied volatilities for ATM European swaptions as at 31/12/17. CVA dataset. Ticker: USSV011 SMKO Curncy onwards. (Bloomberg)*

		Tenor				
		1	2	3	4	5
Term	0.5	-5%	-5%	-9%	-7%	0%
	1	1%	2%	0%	-1%	
	2	5%	3%	0%		
	3	13%	-2%			
	4	6%				

The approach taken has been to calibrate the model to the market implied volatilities, not the market prices. As the market uses a dual curve discounting approach to price the swaptions and we are using a single curve discounting approach to calibrate the model we cannot hope to reconcile with both the market volatilities and the market prices.

Clearly, as the option expiry and swap length become larger the model understates the market prices by an increasing amount. This is because the market discounts the swaption payoffs less heavily with the OIS curve thereby increasing the swaption prices. This occurs even as the drift is overstated. Despite this, it is encouraging how close both our monte-carlo calculated payer swaption prices and implied volatilities are to the market.

Credit Value Adjustment

The CVA equation for a product with counterparty credit risk between time 0 and time *T* is as follows:

$$CVA = \int_0^T LGD_{MKT} EE_t D_t dPD_t \quad (43)$$

where:

dPD_t = Default probability

LGD_{MKT} = The loss given default of the counterparty.

D_i = The risk-free, risk neutral discount factor

Practically this is difficult so solve so it can be discretized. Article 383 from the EU Capital Requirements Regulation (CRR) describes the advanced method for the calculation of the CVA for derivative instruments as:

$$CVA = LGD_{MKT} \sum_{i=1}^{T} max\left\{0, exp\left(-\frac{s_i t_{i-1}}{LGD_{MKT}}\right) - exp\left(-\frac{s_i t_i}{LGD_{MKT}}\right)\right\} \frac{EE_{i-1} D_{i-1} + EE_i D_i}{2} \quad (44)$$

where

s_i = is the credit spread of the counterparty at tenor t_i.

EE_i = The expected exposure to the counterparty at revaluation time t_i.

The following section describes this calculation for an institution trading USD 5-year IRS with Goldman Sachs plc as at 31/12/17.

Probability of Default

Where they are available the probability of default of a counterparty can be bootstrapped from the Credit Default Swap (CDS) spreads of the counterparty to a trade. (Pena) shows by equating the premium and default legs that the fair spread s_N of an N period CDS is given by:

$$s_N = \frac{LGD_{MKT} \sum_{n=1}^{N} D_n(0)(Prob(T_{n-1}) - Prob(T_n))}{\sum_{n=1}^{N} D_n(0) Prob(T_n)(\Delta t_n)} \quad (45)$$

where

$Prob(T_n)$ = The probability of survival up to time T_n.

$D_n(0)$ = The risk free discount factor of term n at t_0.

Given a set of N CDS's, of constant term increment τ this equation can be inverted to determine the term structure of probability of survival:

$$Prob(T_1) = \frac{LGD_{MKT}}{LGD_{MKT} + \tau s_1},$$

$$Prob(T_N) = \frac{\sum_{n=1}^{N-1} D_n(0)[LGD_{MKT} Prob(T_{n-1}) - (LGD_{MKT} + \tau s_N) Prob(T_n)]}{D_n(0)(LGD_{MKT} + \tau s_N)} + \frac{Prob(T_{N-1}) LGD_{MKT}}{(LGD_{MKT} + \tau s_N)}. \quad (46)$$

The following table details market CDS spreads for Goldman Sachs that are used in this project. CDS spreads are quoted with a recovery rate (RR), for Goldman Sachs this was **40%**. This means for a notional of 1, the $LGD_{MKT} = (1 - RR) = 60\%$**.**

Table 24: *USD CDS Spreads for Goldman Sachs plc as at 31/12/17. Ticker CGS1U5 CBIL Curncy onward. (Bloomberg)*

Term (Years)	0.5	1	2	3	4	5
CDS Spread	0.12%	0.19%	0.25%	0.33%	0.42%	0.52%

Just as we found when boostrapping swap rates, we must interpolate the CDS spreads to calculate spreads at 6-month intervals. The approach taken was to linearly interpolate the spreads using equation (1). These interpolated spreads were then used to calculate the probability of survival at each time increment using (46).

Table 25: *Probability of survivals for Goldman Sach plc as at 31/12/17 bootstrapped from CDS spreads using OIS adjusted discount curve.*

Time (years)	0	0.5	1	1.5	2	2.5	3	3.5	4	4.5	5
Probability	1	0.999	0.997	0.994	0.992	0.988	0.983	0.978	0.972	0.965	0.957

Equation (44) calculates the marginal probability of default using the following approximation for the piecewise constant default intensity λ_i for a notional of 1:

$$\lambda_i \approx \frac{s_i}{LGD_{MKT}}. \tag{47}$$

As we have already bootstrapped the probabilities of survival from CDS spreads in **Table 25**, we do not need to make this approximation. (44) therefore becomes:

$$CVA = LGD_{MKT} \sum_{i=1}^{T} max\{0, Prob(t_{i-1}) - Prob(t_i)\} \frac{EE_{i-1}D_{i-1} + EE_iD_i}{2}. \tag{48}$$

Expected Exposure

The exposure *E*(t) at time *t* to an IRS is its positive market to market (MtM). Its expected exposure *EE*(t) is the mean of its simulated exposures at time *t* and its potential future exposure *PFE*(t) is the 97.5th percentile and median of its simulated MtM values. Thus:

$$E(t) = \max(MtM(t), 0) \tag{49}$$

$$EE(t) = \mathbb{E}[max(MtM(t), 0)] \tag{50}$$

$$PFE(t) = Percentile_{97.5th}[max(MtM(t), 0)], Percentile_{50th}[max(MtM(t), 0)] \tag{51}$$

(Brigo & Mercurio, 2006) state the MtM at time t_j, $m^n(t_j)$ of an IRS starting at t_j and ending at t_n is given by:

$$m^n(t_j) = (SR_j^n(t_j) - K) \sum_{i=j+1}^{n} P(t_j, t_i)$$

where $SR_j^n(t_j)$, is the swap rate at t_j required for the PV of the swap to be zero.

This MtM has been defined for LIBOR discounting and LIBOR floating payment. We now move to an OIS discounting, LIBOR floating payment leg setup. As discussed in the OIS Discounting section (and shown in the LOIS.xlsx file), discounting by OIS rather than LIBOR has very little effect on the ATM swap rate $SR_j^n(t_j)$ as the fixed and floating payments are closely matched at each payment. This is true for the 31/12/17 USD LIBOR and OIS curves up to 5 years maturity and does not necessarily hold for other datasets. We therefore make the approximation that the ATM LIBOR swap rate discounted by LIBOR is equal to the ATM LIBOR swap rate discounted by OIS, i.e.:

$$SR_j^n(t_j)^{LIBOR\ Discounting} \approx SR_j^n(t_j)^{OIS\ Discounting} \tag{52}$$

The OIS discounted MtM $m_{OIS}^n(t_j)$ of the LIBOR swap then becomes:

$$m_{OIS}^n(t_j) = (SR_j^{LIBOR,n}(t_j) - K) \sum_{i=j+1}^{n} P_{OIS}(t_j, t_i). \tag{53}$$

We now have an expression with which to calculate the MtM of our 5-year interest rate swap throughout its life. We can calculate its MtM in the future by repricing the swap with the cashflows that are left in the future.

The problem is that this depends on OIS curve simulations which we don't have. The approach taken in this project is to generate the OIS forward rate simulations by adjusting the LIBOR forward rate simulations by an adjusted, simply compounded LIBOR-OIS (LOIS) flat spread. This is calculated by solving for the spread across the 5-year LIBOR curve that gives the same discount factor as a 5-year OIS discount factor. We choose this point with which to reconcile the two curves as the 5-year zero coupon bond has been chosen as the numéraire. See the LOIS.xlsx spreadsheet for more details.

Table 26: *USD OIS Swap rates of terms 0.5 years to 5 years as 31/12/17. Ticker USSOF CMPL Curncy onwards. (Bloomberg)*

Term	0.5	1	2	3	4	5	6
Rate	1.523%	1.672%	1.837%	1.912%	1.942%	1.963%	1.986%

Table 27*: Simply compounded 6-month USD LOIS spread.*

LOIS Spread	0.28%

***Figure 14**: USD LIBOR and OIS Swap rates as at 31/12/17. (Bloomberg)*

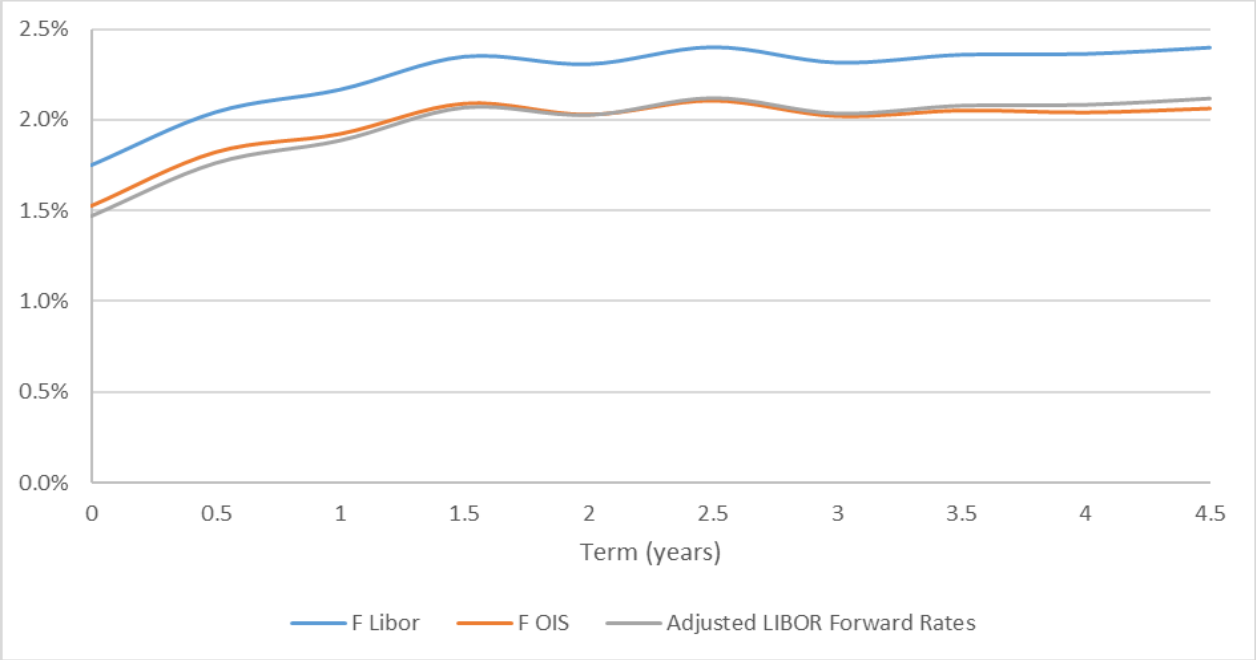

***Figure 15:** Simply compounded, USD LIBOR, OIS and LIBOR adjusted by constant LOIS spread, 6-month forward rates as at 31/12/17. See LOIS.xlsx spreadsheet for calculation details.*

$P_{OIS}(t_j, t_i)$ is then calculated as:

$$P_{OIS}(t_j, t_i) = \frac{1}{\prod_{k=j}^{i-1}(1+\tau(f_k(t_j)-LOIS))} \tag{54}$$

We now move on to how these MtM simulations are used to calculate the CVA. Up until this point there has been no discussion of the terminal bond numéraire. As our forward rate drift is dependent on our choice of numéraire, our MtM(t) values will depend on which numéraire we have chosen. Thus if we changed the numéraire our MtM(t) values would also change.

Clearly the CVA calculation must not be dependent on the model choice of numéraire. This is achieved by the correct choice of discount rate. The $EE_j D_j$ term in (44), is equivalent to saying what is the value now of the expected exposure at time t_j. Thus, for a terminal bond numéraire the discounted expected exposure becomes:

$$EE_j D_j = N(0)\mathbb{E}_{\mathbb{Q}_N}\left[\frac{\max(m^n_{OIS}(t_j),0)}{N(t_j)}\right] \tag{55}$$

Becoming less general for our 5-year IRS with 6-month intervals (10 intervals total) using a 5-year zero coupon bond discounted by OIS as numéraire the expression becomes:

$$EE_j D_j = P^{OIS}_{10}(0)\mathbb{E}_{\mathbb{Q}_{P^{OIS}_{10}}}\left[\frac{\max(m^{10}_{OIS}(t_j),0)}{P^{OIS}_{10}(t_j)}\right] \tag{56}$$

The t_0 terminal bond value can be removed from the summation so (48) becomes:

$$CVA^{10}_{OIS} = 0.5 * LGD_{MKT} P^{OIS}_{10}(0) \sum_{i=1}^{10} max\{0, Prob(t_{i-1}) - Prob(t_i)\}\mathbb{E}_{\mathbb{Q}_{P^{OIS}_{10}}}\left[\frac{\max(m^{10}_{OIS}(t_{i-1}),0)}{P^{OIS}_{10}(t_{i-1})} + \frac{\max(m^{10}_{OIS}(t_i),0)}{P^{OIS}_{10}(t_i)}\right] \tag{57}$$

This CVA calculation method using the OIS adjusted terminal bond numéraire is original work and has not been referenced in any literature viewed by the author.

It is helpful to consider, not the projection of $EE(t)$ which is a numéraire dependent quantity, but $EE(t)D(t)$ which is numéraire independent. The following tables and graphs summarize the expected exposure and CVA calculations for this project.

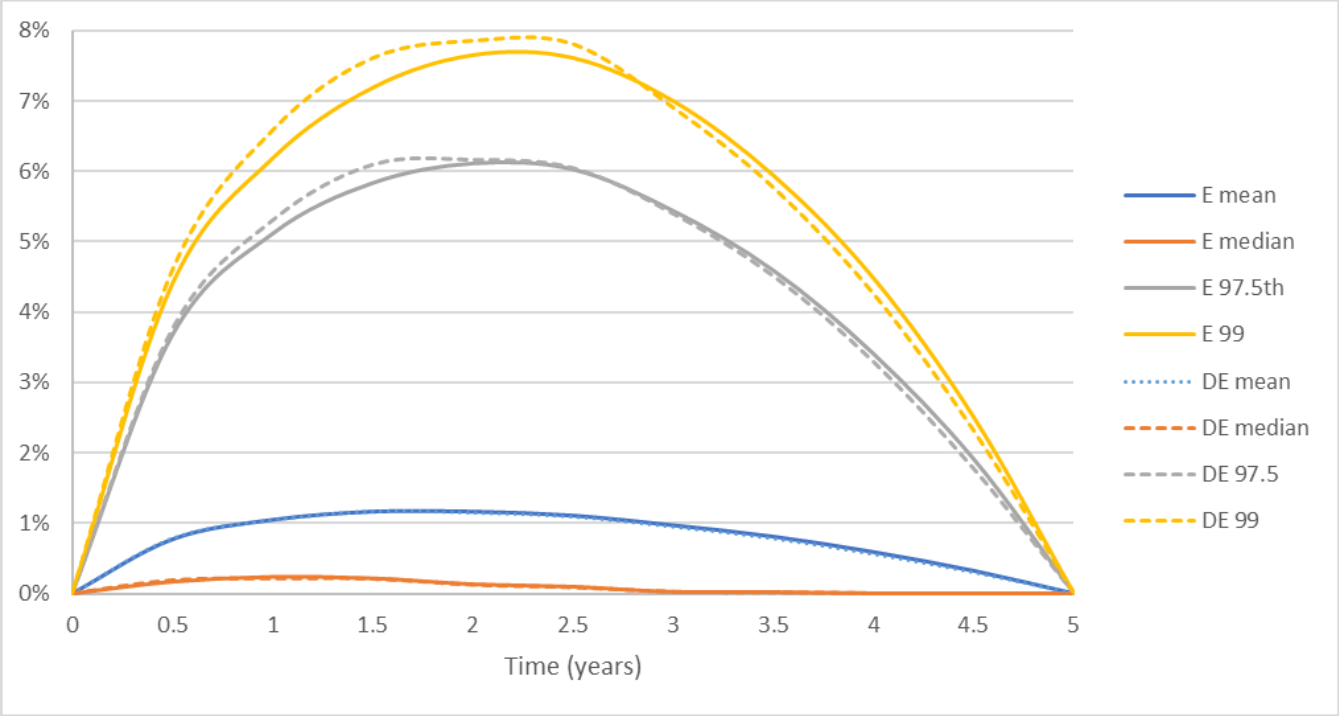

Figure 16: Exposure (*E*) and discounted exposure (*DE*) statistics for 5-year USD LIBOR IRS payer swap using OIS discounting

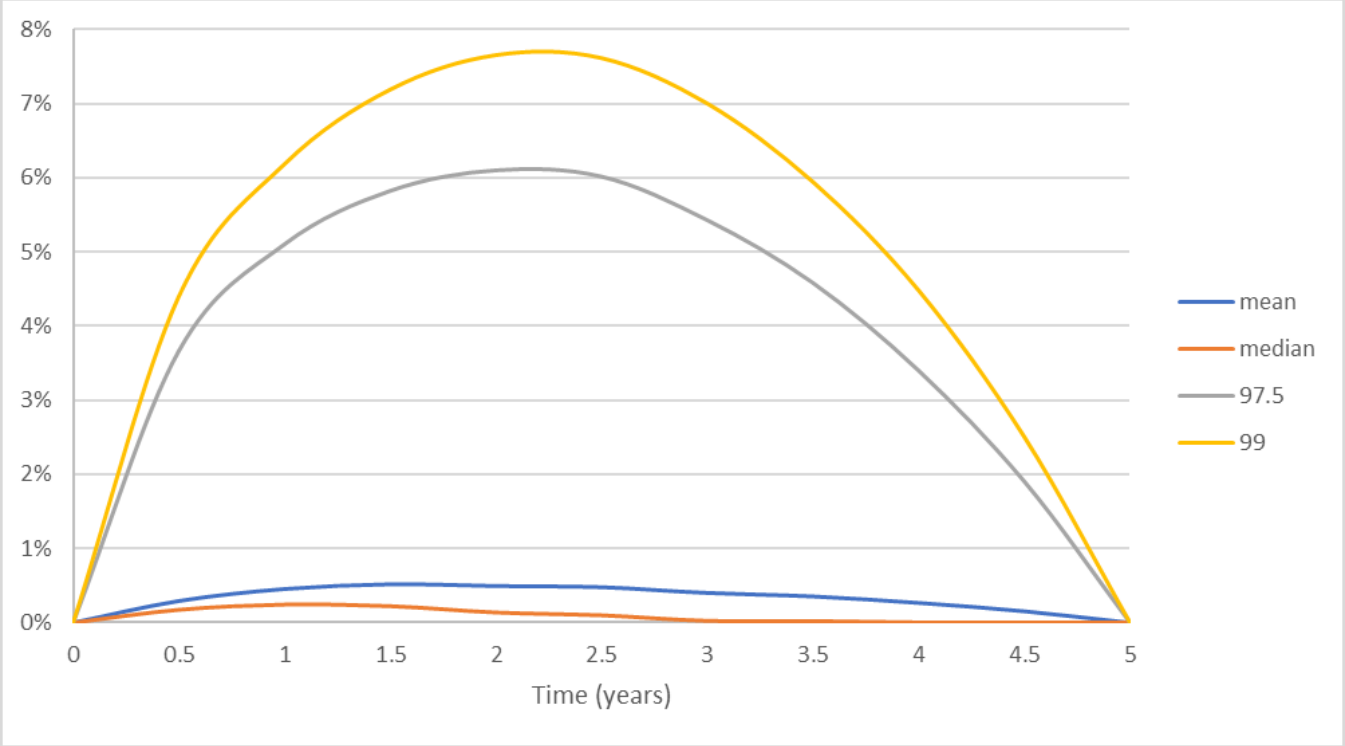

Figure 17: *MtM*(*t*) *values of 5-year USD LIBOR IRS payer swap using OIS discounting.*

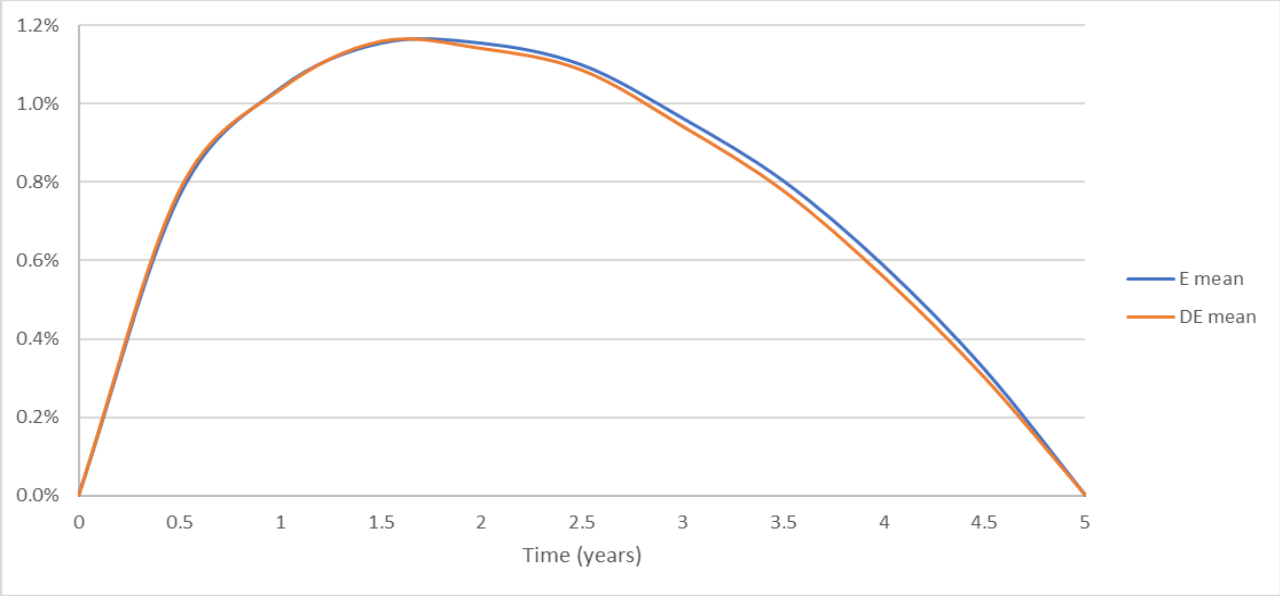

Figure 18: Exposure (*E*) and discounted exposure (*DE*) means for 5-year USD LIBOR IRS payer swap using OIS discounting

Table 28: CVA value for 5-year USD LIBOR IRS traded with Goldman Sachs plc as at 31/12/17

	25th Percentile	mean	75th Percentile	97.5th Percentile	99th Percentile
CVA (bps)	0.06	1.75	2.48	8.62	10.85

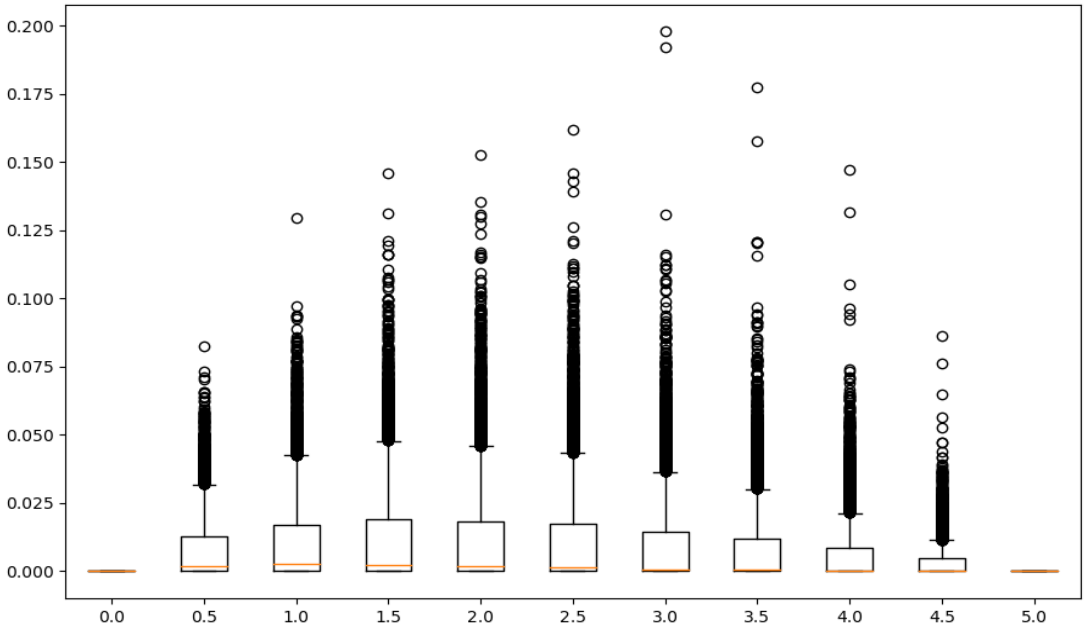

Figure 19: Box and whisker plot of Exposure (*t*) for 5-year USD LIBOR IRS. 10,000 simulations. The box edges detail the quartiles. The whiskers detail 1.5 times past the quartiles.

The exposure mean is around twice that of the MtM. This is because those simulations that generate negative MtM are set to zero in the exposure calculation increasing the mean. Flooring the MtM has no effect on the 97.5th and 99th percentiles of the exposure as the simulations at those percentiles are unchanged as they have positive MtM anyway. Discounting the expectations positively skews the shape of the exposure curves.

Were we to decrease the time-step we would see a drop in the exposure of the swap immediately after each payment in the IRS rather than the smooth curves in **Figure 16**, **Figure 17** and **Figure 18**. As the project model uses a large timestep with a swap payment at each time step so we do not see these steps.

There are a large number of outliers in **Figure 19** due to the large range of forward rate values the risk-neutral LMM produces. Banks using the advanced method for CVA calculation will specify a 99% confidence level for a 10-day time horizon. Using a 99% confidence level, the CVA capital charge for trading a 5-year IRS with Goldman Sachs as at 31/12/17 would be **11 bps**.

Martingale Testing

We have calculated OIS forward rate simulations by adjusting downwards LIBOR forward rates generated by a risk-neutral LMM by a constant LOIS spread. The LIBOR forward rates were generated using a risk neutral drift. This drift has not been re-adjusted when calculating the OIS rates and therefore the model may no longer be risk-neutral when discounting by OIS. Therefore, we must conduct martingale tests for zero-coupon bonds discounted by our simulated OIS discount factors using the following expression:

$$martingale\ ratio = \frac{\mathbb{E}_{\mathbb{Q}_{P_{10}^{OIS}}}\left[\frac{1}{P_{10}^{OIS}(t_i)}\right]}{\frac{P_i^{OIS}(0)}{P_{10}^{OIS}(0)}} \tag{58}$$

Table 29: *Martingale ratios for OIS zero coupon bonds of different maturities at their respective maturities using a 5-year OIS zero coupon bond as numéraire*

Term (years)	0.5	1	1.5	2	2.5	3	3.5	4	4.5
Martingale Ratio	1.001	1.001	1.002	1.002	1.002	1.002	1.002	1.001	1.001
Error (bps)	8.5	14.5	18.1	20.7	22.3	21.4	19.0	14.5	8.1
Error from OIS adjustment (bps)	-0.3	1.0	0.5	0.6	1.4	2.4	2.4	1.8	1.3

While adjusting the LIBOR forward rates by LOIS does introduce an additional positive drift error in the calculation of OIS discount rates, this error is an order of magnitude less than the positive drift error that already existed in the LIBOR forward rates. As this error has been somewhat adjusted for by reducing the volatility, our calculation of simulated OIS discount rates is valid.

Wrong Way Risk

Wrong way risk is when correlation between variables in the CVA calculation are such that when a dealer's exposure to a counterparty increases, the likelihood of default of the counterparty is also increased. This is applicable in our case due to the positive correlation between LOIS and the probability of default of major banks. LOIS is considered a good 'weathervane' for the health of the banking sector as it is evaluating the increased riskiness of lending to a bank rather than a risk-free institution. During the financial crisis LOIS increased dramatically, as did the likelihood of default of most banks.

While LOIS and credit spreads for major banks are correlated, this correlation changes with time and in general correlation is a difficult quantity to model. Therefore sensitivity analysis must be conducted to determine the impact of different correlations between LOIS and CDS spreads.

Figure 20 details CVA values in this sensitivity analysis. LOIS is varied between zero and the lowest forward LIBOR rate in the dataset (1.75%) for fixed LIBOR forward rates. A LOIS value of zero represents the situation where lending to the Fed is as risky as lending to Goldman Sachs. A LOIS value of 1.75% represents the situation where the Fed cash rate is 0% but lending to Goldman Sachs requires a return of 1.75% to compensate the lender for the increased risk of default.

Correlation between CDS spreads and LOIS is captured by a flat adjustment across the t_0 CDS rates equal to the correlation multiplied by the change in LOIS relative to the market LOIS value, i.e.:

$$s_i^{sensitivity} = \frac{LOIS^{sensitivity}}{LOIS} s_i \rho_{CDS,LOIS} \tag{59}$$

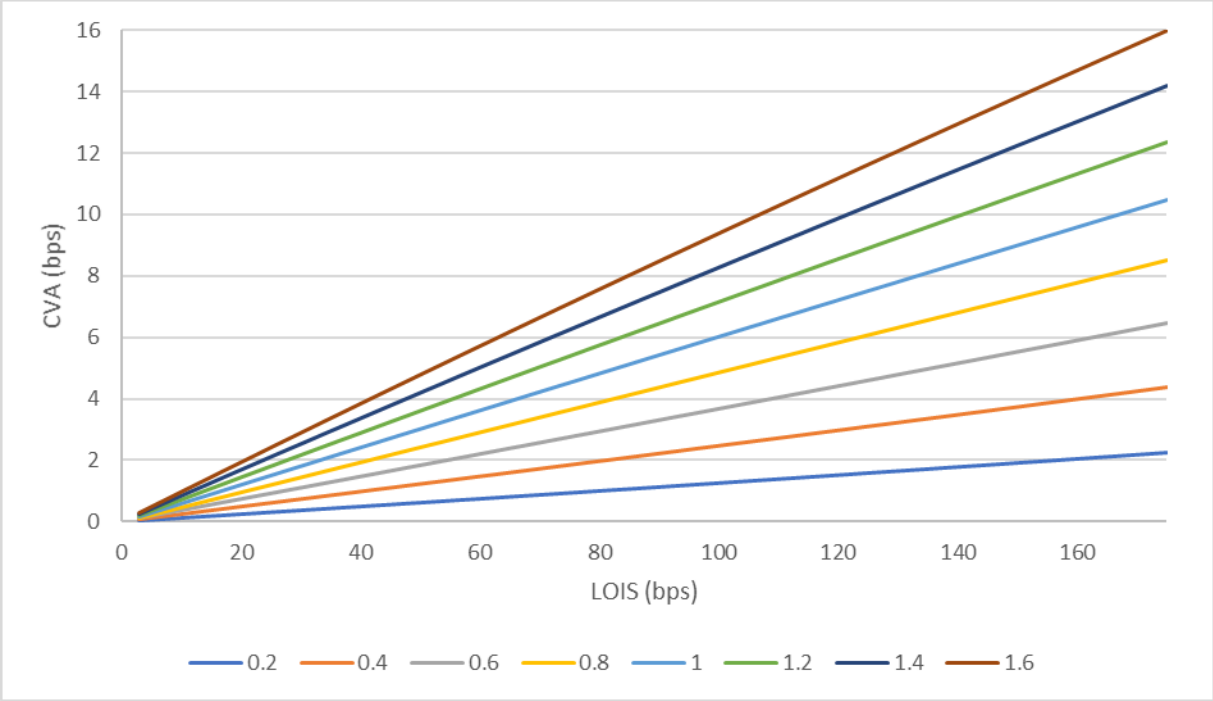

Figure 20: *CVA values for 5-year IRS at varying LOIS and correlation between CDS spreads and LOIS*

For constant CDS spread, decreasing the discount rate (increasing LOIS) decreases the marginal probability of default implied by that spread due to the discount rates role in equation (46). However, this effect is far outweighed by increasing the discounted exposure of the IRS by increasing its PV. When you consider that the CDS spread and LOIS are positively correlated, increasing LOIS both increases the exposure and the marginal probability of default. Thus, the CVA is increasingly affected the more correlated CDS spread and LOIS are.

Conclusion

Both risk-neutral and real world interest rate models are suitable for the CVA calculation, however a risk neutral model is far easier to calibrate to market prices and that has been the approach taken. This project highlights the challenges involved with interest rate modelling.

In addition to correcting the error in the drift there are a number of areas where the model could be improved such as:

- Shorter timesteps to decrease the discretization error and capture the step behavior of the IRS exposure immediately after payments are made.
- Implementation of the predictor-corrector drift method to adjust for the assumption of piecewise constant drift.
- More complex volatility and correlation functions to provide more degrees of freedom to fit the volatility to the market data.
- Stochastic volatility using a model such as the SABR model to capture the volatility smile.

The similarity between the exposure of an IRS and a suitably chosen swaption is such that given the swaption price we are effectively given the IRS expected exposure. By adjusting the volatility to compensate for the error in the drift the model has achieved a good fit to market payer swaption prices (see **Figure 10**) which provides high confidence in the CVA calculation of 11 bps at the 99% confidence interval. Thus the model is suitable for calculating the CVA capital charge for a 5-year payer IRS within a short computing time.

References

Bloomberg. (n.d.).

Brigo, D., & Mercurio, F. (2006). *Interest Rate Models - Theory and Practice.* Springer Finance.

European Banking Authority. (n.d.). *Article 383 - EU Capital Requirements Regulation (Basel III, Pillar I).* Retrieved from eba.europa.eu: http://www.eba.europa.eu/regulation-and-policy/single-rulebook/interactive-single-rulebook/-/interactive-single-rulebook/toc/504/article-id/1576

Gatarek, D., Bachert, P., & Maksymiuk, R. (2006). *The LIBOR MarketModel in Practice.* John Wiley & Sons Ltd.

Hagan, & West. (2005).

Higham, N. J. (2002). Computing the nearest correlation matrix—a problem from finance. *IMA Journal of Numerical Analysis*, 329–343.

Jackel, P., & Rebonato, R. (2002, July 18). *The Link between Capet and Swaption Volatilities in a BGM/J Fraework: Approximate Soluations and Empirical Evidence.* Retrieved from http://www.jaeckel.org/LinkingCapletAndSwaptionVolatilities.pdf

Jaeckel, P. (2002). *Monte Carlo methods in finance.* Chichester, UK: Wiley Finance.

Pena, A. (n.d.). *Credit Default Swaps - Lecture Notes.* Certificate in Quantitative Finance.

Rebonato, R. (1998). *Interest Rate Option Models.* John Wiley and Sons.

The SciPy Community. (2015, Jan 18). *scipy.linalg.cholesky.* Retrieved from docs.scipy.org: https://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.linalg.cholesky.html

The SciPy Community. (2018, May 5). *minimize_scalar.* Retrieved from https://docs.scipy.org: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize_scalar.html

The SciPy community. (2018, May 5). *scipy.optimize.least_squares*. Retrieved from scipy.org: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.least_squares.html

Code

Comments

The programming has been carried out in python using the Anaconda libraries. Effort has been made to implement good software development principles within the code that has been written for this project, specifically, the use of dependency injection and object oriented design. The main.py script calls methods in the lmm.py, validate.py and cva_swap.py files to allow reviewers to reproduce the results shown in this report. Many of the methods write data to .csv files in the project folder. If users wish to inspect this data it is recommended they debug through each method section in the main.py file and inspect the files then as some methods will overwrite the .csv files as the main.py script progresses.

The five classes with the most numerical techniques are included in the following section however all files in the folder submitted with the report were used during the project.

LMM

```
import numpy as np
import volatility as vol
import pandas as pd
import bootstrapping as boot
import copy as copy
import black_sholes_solver as bss
from scipy.optimize import least_squares
from math import *
from scipy.optimize import minimize_scalar

# 1. runs the libor market model and generates forward rate simulations
# 2. produces swaption prices and implied volatilities
# 3. fits an adjustment factor to adjust for drift error
class lmm():
    def __init__(self, swaption_vol_matrix_path, swap_curve_path):
        self.number_of_factors = 3
        self.use_factor_reduction = False
        self.number_of_sims  = 10000
        self.max_projection_time = 40
        self.iterations = 0
        self.bootstrapping = boot.bootstrapping(swap_curve_path)
        self.volatility = vol.volatility(self.number_of_factors, self.bootstrapping,
                                         self.use_factor_reduction, swaption_vol_matrix_path)
        self.time_increment = self.bootstrapping.term_increment
        self.bs_solver = bss.black_sholes_solver(self.volatility)
        self.number_of_terms = self.bootstrapping.number_of_terms
        ##[terms,time, sim]
        self.starting_forward_curve = self.bootstrapping.forward_curve

    def get_random_numbers(self):
        if self.use_factor_reduction:
            return np.random.normal(0, 1, (self.number_of_factors, self.number_of_sims))
        return np.random.normal(0, 1, (self.number_of_terms, self.number_of_sims))

    def set_implied_volatilities_from_prices(self):
        self.implied_volatilities_model_payer =
copy.deepcopy(self.swaption_prices_calibration_payer)
        self.implied_volatilities_model_receiver =
copy.deepcopy(self.swaption_prices_calibration_receiver)

        for row_index, row in self.swaption_prices_calibration_payer.iterrows():
            start = row[self.volatility.term_name]
            values = row.drop([self.volatility.term_name])
            for column_index, v in values.items():
                if False == isnan(v):
                    swap_length = float(column_index)
                    self.bs_solver.set_parameters(start, swap_length, v)
                    implied_volatility_payer =
self.bs_solver.solve_and_get_implied_volatility_payer()
                    self.implied_volatilities_model_payer.at[row_index, column_index] =
implied_volatility_payer

        for row_index, row in self.swaption_prices_calibration_receiver.iterrows():
            start = row[self.volatility.term_name]
            values = row.drop([self.volatility.term_name])
            for column_index, v in values.items():
                if False == isnan(v):
                    swap_length = float(column_index)
                    self.bs_solver.set_parameters(start, swap_length, v)
                    implied_volatility_receiver =
```

```
self.bs_solver.solve_and_get_implied_volatility_receiver()
                    self.implied_volatilities_model_receiver.at[row_index, column_index] =
implied_volatility_receiver

        np.savetxt('implied_volatility_model_payer.csv', self.implied_volatilities_model_payer,
delimiter=',')
        np.savetxt('implied_volatility_model_receiver.csv',
self.implied_volatilities_model_receiver, delimiter=',')


    def objective_function(self, parameters):
        self.iterations += 1
        self.volatility.set_parameters_swap(parameters)
        self.volatility.instantiate_arrays()
        self.set_swaption_prices_for_atm_calibration()
        sum = np.zeros(15)
        N=0
        for row_index, row in self.volatility.swaption_prices.iterrows():
            values = row.drop([self.volatility.term_name])
            for column_index, v in values.items():
                if False == isnan(v):
                    difference = v -  self.swaption_prices_calibration_payer.at[row_index,
column_index]
                    sum[N] = difference
                    N += 1
        return sum

    def objective_function_line_search(self, factor):
        self.iterations += 1
        self.volatility.mc_adjustment_factor = factor
        self.volatility.instantiate_arrays()
        self.set_swaption_prices_for_atm_calibration()
        self.set_implied_volatilities_from_prices()
        sum = np.zeros(15)
        N = 0
        for row_index, row in self.volatility.vol_matrix.iterrows():
            values = row.drop([self.volatility.term_name])
            for column_index, v in values.items():
                if False == isnan(v):
                    difference = v - self.implied_volatilities_model_payer.at[row_index,
column_index]
                    sum[N] = difference
                    N += 1
        value = np.sum(np.power(sum,2))
        return value

    def fit_adjustment_factor(self):
        result = minimize_scalar(self.objective_function_line_search, bounds=(0.1, 0.999),
method='bounded')

    def set_swaption_prices_for_atm_calibration(self):
        self.swaption_prices_calibration_payer = copy.deepcopy(self.volatility.vol_matrix)
        self.swaption_prices_calibration_receiver = copy.deepcopy(self.volatility.vol_matrix)
        self.put_call_difference = copy.deepcopy(self.volatility.vol_matrix)

        for row_index, row in self.volatility.vol_matrix.iterrows():
            number_of_time_steps_to_option_expiry = int(row[self.volatility.term_name] /
self.time_increment)
            start = row[self.volatility.term_name]
            values = row.drop([self.volatility.term_name])
            for column_index, v in values.items():
                if False == isnan(v):
                    swap_length = float(column_index)
```

```
                    swap_length_steps = int(swap_length/self.time_increment)
                    beta = swap_length_steps + number_of_time_steps_to_option_expiry
                    numeraire_index = beta
                    self.run_projection(numeraire_index, number_of_time_steps_to_option_expiry)
                    forward_swap_rate =
self.get_forward_swap_rate(number_of_time_steps_to_option_expiry, numeraire_index)
                    strike = self.bootstrapping.get_forward_swap_rates(start, swap_length)
                    strike_vector = np.ones(self.number_of_sims) * strike
                    sum = np.zeros(self.number_of_sims)

                    for i in range(number_of_time_steps_to_option_expiry + 1, numeraire_index +
1):
                        sum += self.time_increment * self.DF[i,
number_of_time_steps_to_option_expiry, :]

                    payoff_receiver = np.maximum(strike_vector - forward_swap_rate, 0) * sum \
                                      / self.DF[numeraire_index,
number_of_time_steps_to_option_expiry, :]

                    payoff_payer = np.maximum(forward_swap_rate - strike_vector, 0)*sum\
                                   /self.DF[numeraire_index,
number_of_time_steps_to_option_expiry,:]

                    receiver_swaption = np.mean(payoff_receiver) * self.DF[numeraire_index, 0, 0]
                    payer_swaption = np.mean(payoff_payer) * self.DF[numeraire_index, 0,0]


                    self.swaption_prices_calibration_receiver.at[row_index, column_index] \
                        = receiver_swaption

                    self.swaption_prices_calibration_payer.at[row_index, column_index] \
                        = payer_swaption

        np.savetxt('swap_rate_price_payer_model.csv', self.swaption_prices_calibration_payer,
delimiter=',')
        np.savetxt('swap_rate_price_receiver_model.csv',
self.swaption_prices_calibration_receiver, delimiter=',')

    def set_forward_sims(self, numeraire_index, number_of_projection_periods):
        self.forward_sims = np.zeros((numeraire_index,
                                      number_of_projection_periods+1,
                                      self.number_of_sims))
        self.forward_sims[:, 0, :] = np.tile(self.starting_forward_curve[:numeraire_index],
                                             (self.number_of_sims, 1)).transpose()

    def get_forward_swap_rate(self, time_index, numeraire_index):
        sum = np.zeros(self.number_of_sims)

        for i in range(time_index+1, numeraire_index + 1):
            sum += self.time_increment*self.DF[i, time_index, :]
        output = (1 - (self.DF[numeraire_index, time_index,:]))/sum
        return output

    def set_discount_factors(self, numeraire_index, number_of_projection_periods):
        self.DF = np.ones((numeraire_index+1,
                           number_of_projection_periods+1,
                                      self.number_of_sims))
        for n in range(number_of_projection_periods+1):
            for i in range(n + 1, numeraire_index + 1):
                df_prod = np.ones(self.number_of_sims)
                for k in range(n, i):
                    df_prod = df_prod / (np.ones(self.number_of_sims) + self.time_increment *
self.forward_sims[k,n,:])
```

```
                self.DF[i,n,:] = df_prod


    def run_projection(self, numeraire_index, number_of_projection_periods):
        self.set_forward_sims(numeraire_index, number_of_projection_periods)

        for n in range(number_of_projection_periods):
            diffusion = self.get_diffusion()

            for i in range(n+1, numeraire_index):
                    summation = np.zeros(self.number_of_sims)

                    for k in range(i + 1, numeraire_index):
                        forward_sims = self.forward_sims[k, n, :]
                        top = forward_sims * self.time_increment
                        bottom = np.ones(self.number_of_sims) + forward_sims *
self.time_increment
                        quotient = top / bottom
                        correlation = self.volatility.correlation_matrix[i, k]
                        volatility = self.volatility.working_vol_array[k]
                        summation += quotient * correlation * volatility

                    drift = summation * self.volatility.working_vol_array[i]
                    correction = np.ones(self.number_of_sims)*self.volatility.covariance[i,i]/2
                    # as diffusion is time-homogenous it is the difference between term and time
that counts
                    step_diffusion = diffusion[i-n-1,:]
                    step =  self.forward_sims[i, n, :] * np.exp((-drift - correction) *
self.time_increment + step_diffusion)
                    self.forward_sims[i, n+1, :] = step
        self.set_discount_factors(numeraire_index, number_of_projection_periods)

    def run_projection_predictor_corrector(self, numeraire_index, number_of_projection_periods):
        self.set_forward_sims(numeraire_index, number_of_projection_periods)
        previous_drift = np.zeros((numeraire_index, self.number_of_sims))

        for n in range(number_of_projection_periods):
            diffusion = self.get_diffusion()

            for i in range(n + 1, numeraire_index):
                summation = np.zeros(self.number_of_sims)

                for k in range(i + 1, numeraire_index):
                    forward_sims = self.forward_sims[k, n, :]
                    top = forward_sims * self.time_increment
                    bottom = np.ones(self.number_of_sims) + forward_sims * self.time_increment
                    quotient = top / bottom
                    correlation = self.volatility.correlation_matrix[i, k]
                    volatility = self.volatility.working_vol_array[k]
                    summation += quotient * correlation * volatility

                constant_drift = summation * self.volatility.working_vol_array[i]

                if n == 0:
                    working_drift = constant_drift
                else:
                    working_drift = (previous_drift[i,:] + constant_drift)/2

                correction = np.ones(self.number_of_sims) * self.volatility.covariance[i, i] / 2
                step_diffusion = diffusion[i - n - 1, :]
                step = self.forward_sims[i, n, :] * np.exp((-working_drift - correction) *
self.time_increment + step_diffusion)
                self.forward_sims[i, n + 1, :] = step
```

```
            previous_drift[i,:] = working_drift
    self.set_discount_factors(numeraire_index, number_of_projection_periods)

def get_diffusion(self):
    random = self.get_random_numbers()
    vol_matrix = self.volatility.working_chol_matrix
    output = vol_matrix.dot(random)
    return output
```

Volatility

```
import numpy as np
import scipy.optimize as op
from scipy.optimize import least_squares
from math import *
import matplotlib.pyplot as plt
import pandas as pd
from scipy.stats import norm
import copy

# 1. implements Rebonato method for calibration to swaption implied volatilities
# 2. generates correlation and diffusion matrices for use in the LMM module
# 3. Calculates swaption price from swaption volatility using Black formula for swaptions
class volatility():
    def __init__(self, number_of_factors, bootstrapping, use_factor_reduction,
swaption_vol_matrix_path):
        path = swaption_vol_matrix_path
        self.use_factor_reduction = use_factor_reduction
        self.vol_matrix = pd.read_csv(path)
        self.bootstrapping = bootstrapping
        self.term_name = 'expiry'
        self.terms = np.array(self.vol_matrix[self.term_name], float)
        self.tenors = np.array(self.vol_matrix.columns)[1:].astype(np.float)
        self.a =  2.81170636
        self.b = 0.30994865
        self.c = 0.08030116
        self.d =  -2.74048478
        self.beta = 0.1
        self.number_of_factors = number_of_factors
        self.min_term = self.bootstrapping.min_term
        self.time_increment = self.min_term
        self.max_term = self.bootstrapping.max_term
        self.number_of_terms = self.bootstrapping.number_of_terms
        self.mc_adjustment_factor = 0.954816569302
        # self.mc_adjustment_factor = 1

        self.constant_increment_times = \
            np.arange(0, self.max_term, self.time_increment)
        self.instantiate_arrays()

    def instantiate_arrays(self):
        self.set_vol_array()
        self.set_correlation_and_covariance_matrix()
        self.set_diffusion_matrix()

    def set_vol_array(self):
        time_array = np.zeros(self.number_of_terms)
        term_array = np.linspace(self.time_increment, self.max_term, num=(self.number_of_terms))
        self.working_vol_array = ((self.a + self.b*(term_array - time_array))*np.exp(-
self.c*(term_array - time_array)) + self.d)*self.mc_adjustment_factor
        self.forward_vol_matrix = np.diag(self.working_vol_array)

    def set_calibrated_vol_matrix(self):
        self.bootstrapping.set_Z_matrix(self.number_of_terms)
        self.calibrated_swaption_vol_matrix = copy.deepcopy(self.vol_matrix)
        start_time = 0.0

        for row_index, row in self.calibrated_swaption_vol_matrix.iterrows():
            values = row.drop([self.term_name])
            for column_index, v in values.items():
                if False == isnan(v):
```

```
                    swap_start_index = int(row[self.term_name] / self.time_increment)
                    swap_end_index = int(float(column_index) / self.time_increment +
swap_start_index)
                    calculated_vol = self.get_swap_volatility(start_time, swap_start_index,
swap_end_index)
                    self.calibrated_swaption_vol_matrix.at[row_index, column_index] =
calculated_vol
        np.savetxt('calibrated_vols.csv', self.calibrated_swaption_vol_matrix, delimiter=',')


    def calculate_volatility(self, time, term):
        output = ((self.a + self.b*(term - time))*np.exp(-self.c*(term - time)) + self.d)
        return output

    def set_parameters_swap(self, parameters):
        self.a = parameters[0]
        self.b = parameters[1]
        self.c = parameters[2]
        self.d = parameters[3]

    def get_parameters_swap(self):
        output = np.zeros(4)
        output[0] = self.a
        output[1] = self.b
        output[2] = self.c
        output[3] = self.d
        return output

    def get_swaption_price_t0_payer(self, start, swap_length, strike, swaption_volatility):
        d1 = (np.log(self.bootstrapping.get_forward_swap_rates(start, swap_length)/strike) +
              start*np.power(swaption_volatility, 2)/2)/(swaption_volatility*sqrt(start))
        d2 = d1 - swaption_volatility*sqrt(start)
        Nd1 = norm.cdf(d1)
        Nd2 = norm.cdf(d2)

        sum = 0
        start_index = int((start/self.time_increment) - 1)
        end_index = int(((start + swap_length)/self.time_increment)-1)

        for i in range(start_index + 1, end_index + 1):
            sum += self.time_increment\
                   *self.bootstrapping.zero_coupon_prices[i]\
                   *(self.bootstrapping.get_forward_swap_rates(start, swap_length)
                     *Nd1 - strike*Nd2)
        return sum

    def get_swaption_price_t0_receiver(self, start, swap_length, strike, swaption_volatility):
        d1 = (np.log(self.bootstrapping.get_forward_swap_rates(start, swap_length)/strike) +
              start*np.power(swaption_volatility, 2)/2)/(swaption_volatility*sqrt(start))
        d2 = d1 - swaption_volatility*sqrt(start)
        Nd1 = norm.cdf(-d1)
        Nd2 = norm.cdf(-d2)

        sum = 0
        start_index = int((start/self.time_increment) - 1)
        end_index = int(((start + swap_length)/self.time_increment)-1)

        for i in range(start_index + 1, end_index + 1):
            sum += self.time_increment\
                   *self.bootstrapping.zero_coupon_prices[i]\
                   *(strike*Nd2 -self.bootstrapping.get_forward_swap_rates(start,
swap_length)*Nd1)
        return sum
```

```
    def set_swaption_price_matrix(self):
        self.swaption_prices = copy.deepcopy(self.vol_matrix)

        for row_index, row in self.vol_matrix.iterrows():
            values = row.drop([self.term_name])
            for column_index, v in values.items():
                if False == isnan(v):
                    swap_length = float(column_index)
                    start = row[self.term_name]
                    # if swap_length < 15.0 and start < 15.0:
                    K = self.bootstrapping.get_forward_swap_rates(start, swap_length)
                    swaption_price_t0 = self.get_swaption_price_t0_payer(start, swap_length, K, v)
                    self.swaption_prices.at[row_index, column_index] =  swaption_price_t0

    def fit_parameters(self):
        initial_parameters = self.get_parameters_swap()
        self.bootstrapping.set_Z_matrix(self.number_of_terms)
        result = least_squares(self.objective_function_swap, initial_parameters)
        return result


    def objective_function_swap(self, parameters):
        num_rows = self.vol_matrix[self.term_name].count()
        num_columns = len(self.vol_matrix.columns) - 1
        number_of_iterations = int(num_rows*num_columns)
        sum = np.zeros(number_of_iterations)
        N = 0
        start_time = 0.0
        self.set_parameters_swap(parameters)

        for row_index, row in self.vol_matrix.iterrows():
            values = row.drop([self.term_name])
            for column_index, v in values.items():
                if False == isnan(v):
                    swap_start_index = int(row[self.term_name]/self.time_increment)
                    swap_end_index = int(float(column_index)/self.time_increment +
swap_start_index)
                    calculated_vol = self.get_swap_volatility(start_time, swap_start_index,
swap_end_index)
                    sum[N] = calculated_vol - v
                    N += 1
        return sum

    def get_swap_volatility(self, option_start_time, swap_start_index, swap_end_index):
        total_sum = 0
        for k in range(swap_start_index, swap_end_index):
            for l in range(swap_start_index, swap_end_index):
                time_k = self.constant_increment_times[k]
                time_l = self.constant_increment_times[l]
                swap_start_time = swap_start_index*self.time_increment

                integrated_covariance_end = self.get_integrated_covariance(swap_start_time,
time_k, time_l)
                integrated_covariance_start = self.get_integrated_covariance(option_start_time,
time_k, time_l)

                integrated_covariance = (integrated_covariance_end - integrated_covariance_start)\
                                        /(swap_start_time - option_start_time)
                first_Z = self.bootstrapping.Z_matrix[swap_start_index, swap_end_index, k]
                second_Z = self.bootstrapping.Z_matrix[swap_start_index, swap_end_index, l]
                total_sum += first_Z*integrated_covariance*second_Z
        output = np.sqrt(total_sum)
```

```
        return output


    def get_correlation(self, term1, term2):
        return np.exp(-self.beta * abs(term1 - term2))

    def set_correlation_and_covariance_matrix(self):
        self.correlation_matrix = np.ones((self.number_of_terms, self.number_of_terms))

        for i in range(0, self.number_of_terms):
            for j in range(0, self.number_of_terms):
                self.correlation_matrix[i, j] = self.get_correlation((i+1)*self.time_increment,
(j+1)*self.time_increment)
        self.covariance = np.matmul(self.forward_vol_matrix, np.matmul(self.correlation_matrix,
self.forward_vol_matrix))

    def set_diffusion_matrix(self):
        if self.mc_adjustment_factor == 0:
            self.chol_covariance = self.covariance
            self.working_chol_matrix = self.covariance
        else:
            self.chol_covariance = np.linalg.cholesky(self.covariance)

            if self.use_factor_reduction:
                self.working_chol_matrix = self.chol_covariance[:, :self.number_of_factors]

                for i in range(0, self.number_of_terms):
                    for j in range(0, self.number_of_factors):
                        sum = np.sum(np.power(self.chol_covariance[i,:self.number_of_factors], 2))

                        quotient = self.covariance[i, i] / sum
                        self.working_chol_matrix[i, j] = self.working_chol_matrix[i, j] *
np.sqrt(quotient)
            else:
                self.working_chol_matrix = self.chol_covariance


    def get_integrated_covariance(self, time, term_i, term_j):
        first_line = 4*self.a*np.power(self.c, 2)*self.d*(np.exp(self.c*(time - term_j))
                                                          + np.exp(self.c*(time - term_i)))
\
                     + 4*np.power(self.c, 3)*np.power(self.d, 2)*time
        second_line = -4*self.b*self.c*self.d*np.exp(self.c*(time - term_i))*(self.c*(time -
term_i) - 1) \
                      - 4*self.b*self.c*self.d*np.exp(self.c*(time - term_j))*(self.c*(time -
term_j) - 1)
        third_line = np.exp(self.c*(2*time - term_i - term_j))\
                     *(2*np.power(self.a, 2)*np.power(self.c, 2)
                       + 2*self.a*self.b*self.c*(1 + self.c*(term_i + term_j - 2*time))
                       + np.power(self.b, 2)*(1 + 2*np.power(self.c, 2)*(time - term_i)*(time -
term_j)
                                              + self.c*(term_i + term_j - 2*time)))
        multiplier = self.get_correlation(term_i, term_j)/(4*np.power(self.c, 3))
        return multiplier*(first_line + second_line + third_line)

    def get_correlated_volatility_cholesky_matrix(self, time):
        c_matrix = np.ones([self.number_of_terms,self.number_of_terms])

        for i in range(0, self.number_of_terms):
            term1 = i*self.time_increment
            for j in range(0, self.number_of_terms):
                term2 = j*self.time_increment
                c_matrix[i, j] = self.get_correlation(term1, term2)\
```

```
                        *self.calculate_volatility(time, term1)\
                        *self.calculate_volatility(time, term2)
    a_bar_matrix = np.linalg.cholesky(c_matrix)

    if self.use_factor_reduction:
        reduced = np.zeros([self.number_of_terms, self.number_of_factors])

        for i in range(0, self.number_of_terms):
            for j in range(0, self.number_of_factors):
                sum = 0

                for k in range(0, self.number_of_factors):
                    sum += np.power(a_bar_matrix[i, k],2)

                quotient = c_matrix[i, i]/sum
                reduced[i, j] = a_bar_matrix[i, j] * np.sqrt(quotient)
        return reduced
    return a_bar_matrix
```

Bootstrapping

```
import math as math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import svensson as sven


# 1. Imports swap curve and interpolates using linear interpolation
# 2. Using linear interpolation bootstraps zero coupon bond values from swap rates
# 3. Generates Z matrix for use in Rebonato method
# 4. Calculates forward swap rates from forward rates for use in swaption pricing.
class bootstrapping():
    def __init__(self, swap_curve_path):
        # path = 'SpotCurve cut down.csv'
        path = swap_curve_path
        self.term_name = 'term'
        self.yield_name = 'yield'
        self.term_increment = 0.5
        self.notional = 1.0
        self.min_term = 0.5

        self.imported_swap_curve = pd.read_csv(path)
        self.max_term = int(max(self.imported_swap_curve[self.term_name]))
        self.number_of_terms = int((self.max_term) / self.term_increment)
        self.set_linear_interpolated_swap_rates_vector()
        self.set_forward_curve()

    # Ai in Jaekal notes. t = 0. start_index = i
    def get_t0_floating_leg_value(self, start_index, end_index):
        sum = 0
        for j in range(start_index, end_index):
            sum +=
self.zero_coupon_prices[j+1]*self.forward_curve[j]*self.term_increment*self.notional
        return sum

    # Bi in Jaekal notes. t = 0
    def get_t0_fixed_leg_value(self, start_index, end_index):
        sum = 0
        for j in range(start_index, end_index):
            sum += self.zero_coupon_prices[j+1]*self.term_increment*self.notional
        return sum

    def convert_index_to_term_forward(self, i):
        return i*self.term_increment

    def set_Z_matrix(self, number_of_indexes):
        self.Z_matrix = np.zeros((number_of_indexes, number_of_indexes, number_of_indexes))

        for start_index in range(0, number_of_indexes):
            for end_index in range(start_index + 1, number_of_indexes):
                for k in range(start_index, end_index):
                    self.Z_matrix[start_index, end_index, k] = self.get_Z(start_index,
end_index, k)

    def get_Z(self, start_index, end_index, k):
        if k < start_index:
            return 0
        if k == start_index:
            return 1
```

```
        const_weights_approx = self.notional*self.zero_coupon_prices[k+1]\
                               *self.forward_curve[k]*self.term_increment\
                               /self.get_t0_floating_leg_value(start_index, end_index)
        top = (self.get_t0_floating_leg_value(start_index, end_index)
               *self.get_t0_fixed_leg_value(k, end_index) - self.get_t0_floating_leg_value(k,
end_index)
               *self.get_t0_fixed_leg_value(start_index, end_index))\
              *self.forward_curve[k]*self.term_increment
        bottom = self.get_t0_floating_leg_value(start_index, end_index)*(1 +
self.forward_curve[k]*self.term_increment)
        shape_correction = top/bottom
        return const_weights_approx + shape_correction

    def set_linear_interpolated_swap_rates_vector(self):
        self.interpolated_swap_rates = np.zeros(self.number_of_terms)
        previous_swap_length_steps = 0
        previous_swap_length = 0
        previous_swap_rate = 0
        count = 0

        for row_index, row in self.imported_swap_curve.iterrows():
            swap_length = row[self.term_name]
            swap_length_steps = int((swap_length-self.min_term)/ self.term_increment)
            swap_rate = float(row[self.yield_name])

            if row_index == 0:
                self.interpolated_swap_rates[count] = swap_rate
                previous_swap_length_steps = swap_length_steps
                previous_swap_length = swap_length
                previous_swap_rate = swap_rate
                count += 1
            else:
                distance = swap_length_steps - previous_swap_length_steps

                for i in range(1, distance):
                    interpolate_length = previous_swap_length + self.term_increment*i
                    interpolate_swap_rate = (interpolate_length -
previous_swap_length)*swap_rate/(swap_length - previous_swap_length)\
                                + (swap_length -
interpolate_length)*previous_swap_rate/(swap_length - previous_swap_length)
                    self.interpolated_swap_rates[count] = interpolate_swap_rate
                    count += 1
                self.interpolated_swap_rates[count] = swap_rate
                previous_swap_rate = swap_rate
                previous_swap_length = swap_length
                previous_swap_length_steps = swap_length_steps
                count += 1
        # np.savetxt('linear_interpolated_swap_values.csv', self.interpolated_swap_rates,
delimiter=',')

    def get_forward_swap_rates(self, start, swap_length):
        start_index = int(start/self.term_increment)
        end_index = int((start + swap_length)/self.term_increment)
        top = self.zero_coupon_prices[start_index] - self.zero_coupon_prices[end_index]
        bottom = 0

        for i in range(start_index + 1, end_index + 1):
            bottom += self.term_increment * self.zero_coupon_prices[i]
        return top/bottom



    # start index = alpha, end_index = beta
```

```
    def get_forward_swap_rates_from_forward_rates(self, start_index, end_index, forward_rates):
        number_of_sims = len(forward_rates[0,:])

        product = 1/(1+ self.term_increment*forward_rates[start_index, :])
        for i in range(start_index + 1, end_index):
            product = product/(1 + self.term_increment*forward_rates[i, :])
        floating_leg = np.ones(number_of_sims) - product

        bottom = 0

        for i in range(start_index, end_index):
            product = 1/(1+ self.term_increment*forward_rates[start_index, :])

            for j in range(start_index + 1, i+1):
                product = product/(1 + self.term_increment*forward_rates[j, :])

            bottom += self.term_increment*product
        return floating_leg/bottom

    # forward_rates [term, sim]
    def get_discount_factors_from_forward_rates(self, start_index, end_index, forward_rates,
lois):
        number_of_sims = len(forward_rates[0, :])
        lois_vector = np.ones(number_of_sims)*lois

        product = 1/(1 + self.term_increment*(forward_rates[start_index, :] - lois_vector))
        for i in range(start_index + 1, end_index):
            product/(1 + self.term_increment*(forward_rates[i, :] - lois_vector))
        return product

    def set_ois_adjusted_discount_factors(self, lois):
        self.ois_discount_factors = np.zeros(self.number_of_terms + 1)
        self.ois_discount_factors[0] = 1
        self.ois_discount_factors[1] = 1 / (1 + self.term_increment * (self.forward_curve[0] -
lois))

        for i in range(2, self.number_of_terms + 1):
            self.ois_discount_factors[i] = self.ois_discount_factors[i-1]/ (1 +
self.term_increment * (self.forward_curve[i-1] - lois))

    # zero_coupon_prices[0] = 1
    def set_zero_coupon_prices_from_swap_rates(self):
        self.zero_coupon_prices = np.zeros(self.number_of_terms + 1)

        self.zero_coupon_prices[0] = 1
        self.zero_coupon_prices[1] = 1/(1 +
self.term_increment*self.interpolated_swap_rates[0])

        for i in range(2, self.number_of_terms + 1):
            rN = self.interpolated_swap_rates[i - 1]
            sum = 0
            for k in range(1, i):
                sum += self.zero_coupon_prices[k]
            self.zero_coupon_prices[i] = (1 - rN*sum*self.term_increment)/(1+
rN*self.term_increment)

    def get_pv_of_swap_t0(self, swap_length_steps, swap_rate, forward_sims):
        fixed_leg = 0
        floating_leg = 0

        for i in range(swap_length_steps):
            zero_coupon_bond = 1 / (1 + self.term_increment * (forward_sims[0, :]))
```

```
            for j in range(1, i + 1):
                zero_coupon_bond = zero_coupon_bond / (
                    1 + self.term_increment * (forward_sims[j, :]))
                fixed_leg += zero_coupon_bond * swap_rate * self.term_increment
                floating_leg += zero_coupon_bond * forward_sims[i, :] * self.term_increment
        return floating_leg - fixed_leg


    # forward_curve[0] = t = 0, 0.5 term spot rate
    def set_forward_curve(self):
        self.forward_curve = np.zeros(self.number_of_terms)
        self.forward_curve[0] = self.interpolated_swap_rates[0]

        for i in range(1, self.number_of_terms):
            self.forward_curve[i] = (self.zero_coupon_prices[i]/self.zero_coupon_prices[i+1] -
1)/self.term_increment
```

CVA

```
import numpy as np
import volatility as vol
import pandas as pd
import bootstrapping as boot
import copy as copy
import black_sholes_solver as bss
from lmm import *
import matplotlib.pyplot as plt
import cds_bootstrap as cds_boot

# 1. calculates LOIS adjusted OIS simulations
# 2. calculates CVA for 5 year IRS with Goldman Sachs as at 31/12/17
# 3. Conducts sensitivity analysis to LOIS and LOIS-CDS correlation
class cva_swap():
    def __init__(self):
        self.recovery_rate = 0.4
        swaption_vol_cva_dataset_path = 'SwaptionVolMatrix_5Y.csv'
        swap_curve_cva_dataset_path = 'SpotCurve_5Y.csv'
        path = 'ois.csv'
        self.lmm = lmm(swaption_vol_cva_dataset_path, swap_curve_cva_dataset_path)
        self.time_increment = self.lmm.time_increment
        self.bootstrapping = self.lmm.bootstrapping
        self.lois = 0.00282385038573462
        self.working_lois = self.lois
        self.numeraire_index = 10
        self.set_ois_discount_curve(self.lois)
        self.cds_bootstrapping = cds_boot.cds_bootstrap(self.recovery_rate,
                                                        self.time_increment)
        self.lmm.run_projection(self.numeraire_index, self.numeraire_index)
        ##[terms,time, sim]
        self.forward_sims = self.lmm.forward_sims
        self.number_of_sims = self.lmm.number_of_sims
        self.notional = 1
        self.LGD = self.notional*(1- self.recovery_rate)
        self.swap_term = 5
        self.set_ois_discount_sims()
        self.calculate_CVA_for_5_year_swap()
        self.calculate_martingale_ratios_for_CVA_dataset_bonds_at_expiry_ois()
        self.run_sensitivity_analysis()

    def set_ois_discount_curve(self, lois):
        libor_forward_rates = self.lmm.starting_forward_curve
        self.ois_discount_factors = np.ones(self.numeraire_index + 1)

        for i in range(1, self.numeraire_index + 1):
            self.ois_discount_factors[i] = self.ois_discount_factors[i-1]/(1 +
self.time_increment*(libor_forward_rates[i-1] - lois))

    def set_ois_discount_sims(self):
        lois_sims = self.working_lois * np.ones(self.number_of_sims)
        self.ois_DF = np.ones((self.numeraire_index + 1,
                               self.numeraire_index + 1,
                               self.number_of_sims))
        for n in range(self.numeraire_index + 1):
            for i in range(n + 1, self.numeraire_index + 1):
                df_prod = np.ones(self.number_of_sims)
                for k in range(n, i):
                    df_prod = df_prod / \
                              (np.ones(self.number_of_sims) + self.time_increment *
(self.forward_sims[k, n, :] - lois_sims))
```

```
                self.ois_DF[i, n, :] = df_prod

    def get_pv_of_swap_alternate(self, swap_length_steps, strike, time_index):
        swap_rate_libor = self.lmm.get_forward_swap_rate(time_index, swap_length_steps)
        annuity = np.zeros(self.number_of_sims)

        for i in range(time_index + 1, self.numeraire_index+1):
            annuity += self.ois_DF[i,time_index,:]
        output = (swap_rate_libor - strike)*self.time_increment*annuity
        return output

    def calculate_exposure(self, swap_pv):
        output = np.maximum(swap_pv, np.zeros(self.number_of_sims))
        return output

    def run_sensitivity_analysis(self):
        swap_rate =
float(self.bootstrapping.imported_swap_curve.iloc[self.swap_term][self.bootstrapping.yield_name])
        swap_length_steps = int(self.swap_term / self.time_increment)
        correlation_increments = 8
        lois_increments = 62
        cva_matrix = np.zeros((correlation_increments, lois_increments))
        lois_values = np.zeros(lois_increments)
        correlations = np.zeros(correlation_increments)
        cds_multipliers = np.zeros(lois_increments)

        for i in range(correlation_increments):
            correlation = 0.2 + 0.2*i
            correlations[i] = correlation
            for j in range(lois_increments):
                lois_multiplier = 0.1 + j*0.1
                lois_values[j] = self.lois*lois_multiplier
                self.working_lois = self.lois*lois_multiplier
                self.set_ois_discount_curve(self.working_lois)
                self.set_ois_discount_sims()
                cds_multiplier = correlation*lois_multiplier
                cds_multipliers[j] = cds_multiplier
                cva_matrix[i,j] = self.calculate_cva(swap_length_steps, swap_rate, cds_multiplier)
        np.savetxt('cva_matrix.csv', cva_matrix, delimiter=',')
        np.savetxt('lois_values.csv', lois_values, delimiter=',')

    def calculate_CVA_for_5_year_swap(self):
        swap_rate =
float(self.bootstrapping.imported_swap_curve.iloc[self.swap_term][self.bootstrapping.yield_name])
        swap_length_steps = int(self.swap_term / self.time_increment)
        cds_multiplier = 1
        cva = self.calculate_cva(swap_length_steps, swap_rate, cds_multiplier)

        exposure_stats = self.get_stats(self.exposure, swap_length_steps)
        mtm_stats = self.get_stats(self.swap_mtm, swap_length_steps)
        discounted_exposure_stats = self.get_stats(self.discounted_exposures, swap_length_steps)

        np.savetxt('exposure_stats.csv', exposure_stats, delimiter=',')
        np.savetxt('mtm_stats.csv', mtm_stats, delimiter=',')
        np.savetxt('discounted_exposure_stats.csv', discounted_exposure_stats, delimiter=',')

    def calculate_martingale_ratios_for_CVA_dataset_bonds_at_expiry_ois(self):
        bonds = np.zeros(self.numeraire_index)
        ratio = np.zeros(self.numeraire_index)
        difference = np.zeros(self.numeraire_index)

        for i in range(1, self.numeraire_index):
            numeraire_value = self.ois_DF[self.numeraire_index, i, :]
```

```
            t0_value = self.ois_DF[self.numeraire_index, 0, 0]
            bonds[i] = np.mean(1 / numeraire_value) * t0_value
            difference[i] = bonds[i] - self.ois_DF[i, 0, 0]
            ratio[i] = bonds[i] / self.ois_DF[i, 0, 0]
        np.savetxt('martingale_ratio_at_bond_expiry_CVA_dataset_ois_adjustment.csv', ratio,
delimiter=',')

    def get_stats(self, sims, swap_length_steps):
        output = np.zeros((swap_length_steps + 1, 4))
        output[:, 0] = np.mean(sims, axis=1)
        output[:, 1] = np.median(sims, axis=1)
        # no non-int percentiles in numpy so average 97 and 98th percentiles to get 97.5th
percentile
        output[:, 2] = (np.percentile(sims, 97, axis=1) + np.percentile(sims, 98, axis=1)) / 2
        output[:, 3] = np.percentile(sims, 99, axis=1)
        return output


    def calculate_exposures(self, swap_length_steps, swap_rate):
        self.exposure = np.zeros((swap_length_steps + 1, self.number_of_sims))
        self.swap_mtm = np.zeros((swap_length_steps + 1, self.number_of_sims))

        for t in range(swap_length_steps):
            swap_pv = self.get_pv_of_swap_alternate(swap_length_steps, swap_rate,t)
            self.swap_mtm[t] = swap_pv
            swap_exposure = self.calculate_exposure(swap_pv)
            self.exposure[t] = swap_exposure
        return self.exposure

    def get_discounted_exposures(self, exposures, swap_length_steps):
        self.discounted_exposures = np.zeros((swap_length_steps + 1, self.number_of_sims))

        for i in range(swap_length_steps+1):
            self.discounted_exposures[i,:] = self.ois_discount_factors[self.numeraire_index]\
                                             *exposures[i]/self.ois_DF[self.numeraire_index,i,:]
        return self.discounted_exposures


    def calculate_cva(self, swap_length_steps, swap_rate, cds_multiplier):
        exposures = self.calculate_exposures(swap_length_steps, swap_rate)
        discounted_exposures = self.get_discounted_exposures(exposures, swap_length_steps)
        probability_of_survival =
self.cds_bootstrapping.get_probability_of_survival(self.ois_discount_factors, cds_multiplier)
        # np.savetxt('probability_of_survival.csv', probability_of_survival, delimiter=',')
        sum = np.zeros(self.number_of_sims)
        for i in range(1, swap_length_steps):
            marginal_probability_of_default = probability_of_survival[i-1] \
                                              - probability_of_survival[i]
            intepolated_exposure = (discounted_exposures[i - 1] + discounted_exposures[i]) / 2
            sum += np.maximum(0, marginal_probability_of_default)*intepolated_exposure
        self.CVA_stochastic = sum*self.LGD
        self.CVA_mean = np.mean(self.CVA_stochastic)
        # CVA_99 = np.percentile(self.CVA_stochastic, 99)
        # CVA_ninety_seventh = (np.percentile(self.CVA_stochastic, 97) +
np.percentile(self.CVA_stochastic, 98)) / 2
        # CVA_seventy_fifth = np.percentile(self.CVA_stochastic, 75)
        # # CVA_twenty_fifth = np.percentile(self.CVA_stochastic, 25)
        # s = 'The mean CVA value of a 5 year USD libor interest rate swap as at 31/12/17 is ' +
str(self.CVA_mean) + ' for a notional of 1.'
        # print(s)
```

Validation

```
import numpy as np
import lmm as lmm
import matplotlib.pyplot as plt

# validates the LMM forward rate simulations using martingale tests and other
# tests
class validation():
    def __init__(self):
        swaption_vol_cva_dataset_path = 'SwaptionVolMatrix_5Y.csv'
        swap_curve_cva_dataset_path = 'SpotCurve_5Y.csv'
        self.lmm = lmm.lmm(swaption_vol_cva_dataset_path, swap_curve_cva_dataset_path)
        self.calculate_martingale_ratios_for_CVA_dataset_bonds_at_expiry()

        swaption_vol_extended_dataset_path = 'SwaptionVolMatrix.csv'
        swap_curve_extended_dataset_path = 'SpotCurve.csv'
        self.lmm = lmm.lmm(swaption_vol_extended_dataset_path, swap_curve_extended_dataset_path)
        self.calculate_10_year_ZC_martingale_test()

        self.calculate_zero_coupon_bond_projections()
        # uncomment to do diffusion check
        # self.check_diffusion_has_zero_mean()

        ##[terms,time, sim]
        self.forward_sims = self.lmm.forward_sims
        self.number_of_terms = self.lmm.number_of_terms
        self.time_increment = self.lmm.time_increment
        self.bootstrapping = self.lmm.bootstrapping
        self.number_of_sims = self.lmm.number_of_sims

    def set_martingale_differences_for_zero_coupon_bond(self):
        self.martingale_differences = np.ones((self.number_of_terms, 3))
        # loop through zero coupon bonds
        for i in range(1, self.number_of_terms+1):
            bond_pv = self.get_expectation_of_zero_coupon_bond(i)
            t0_bond_pv = self.bootstrapping.zero_coupon_prices[i]
            self.martingale_differences[i - 1,0] = bond_pv
            self.martingale_differences[i - 1, 1] = t0_bond_pv
            self.martingale_differences[i - 1, 2] = bond_pv / t0_bond_pv - 1
        np.savetxt('martingale_test.csv', self.martingale_differences, delimiter=',')

    def calculate_martingale_ratios_for_CVA_dataset_bonds_at_expiry(self):
        numeraire_index = 10
        self.lmm.run_projection(numeraire_index, numeraire_index)
        bonds = np.zeros(numeraire_index)
        ratio = np.zeros(numeraire_index)
        difference = np.zeros(numeraire_index)

        for i in range(1, numeraire_index):
            numeraire_value = self.lmm.DF[numeraire_index, i,:]
            t0_value = self.lmm.DF[numeraire_index,0,0]
            bonds[i] = np.mean(1/numeraire_value)*t0_value
            difference[i] = bonds[i] - self.lmm.DF[i,0,0]
            ratio[i] = bonds[i]/self.lmm.DF[i,0,0]
        np.savetxt('martingale_ratio_at_bond_expiry_CVA_dataset.csv', ratio, delimiter=',')

    def calculate_zero_coupon_bond_projections(self):
        numeraire_index = 40
        start_bond = 20
        self.lmm.volatility.mc_adjustment_factor = 1
        self.lmm.volatility.a = 0.01368861
```

```
        self.lmm.volatility.b = 0.07921976
        self.lmm.volatility.c = 0.33920146
        self.lmm.volatility.d = 0.08416935
        self.lmm.volatility.instantiate_arrays()
        self.lmm.run_projection(numeraire_index, numeraire_index)
        bonds = np.zeros((numeraire_index - start_bond, numeraire_index))
        ratio = np.zeros((numeraire_index - start_bond, numeraire_index))

        for i in range(start_bond, numeraire_index):
            for j in range(i+1):
                numeraire_value = self.lmm.DF[numeraire_index, j, :]
                t0_numeraire_value = self.lmm.DF[numeraire_index, 0, 0]
                t0_ratio =  self.lmm.DF[i, 0, 0]/t0_numeraire_value
                bonds[i-start_bond,j] = np.mean(self.lmm.DF[i, j,:] / numeraire_value) *
t0_numeraire_value
                ratio[i-start_bond,j] = (np.mean(self.lmm.DF[i, j,:] / numeraire_value))/t0_ratio
        np.savetxt('matingale_test_ratio_projections.csv', ratio, delimiter=',')

    def calculate_10_year_ZC_martingale_test(self):
        numeraire_index = 40
        start_bond = 20
        self.lmm.volatility.mc_adjustment_factor = 1
        self.lmm.volatility.a = 0.01368861
        self.lmm.volatility.b = 0.07921976
        self.lmm.volatility.c = 0.33920146
        self.lmm.volatility.d = 0.08416935
        self.lmm.volatility.instantiate_arrays()
        self.lmm.run_projection(numeraire_index, numeraire_index)
        # self.lmm.run_projection_predictor_corrector(numeraire_index, numeraire_index)

        bonds = np.zeros((numeraire_index - start_bond, numeraire_index))
        ratio = np.zeros((4, numeraire_index))
        for j in range(start_bond + 1):
            numeraire_value = self.lmm.DF[numeraire_index, j, :]
            t0_numeraire_value = self.lmm.DF[numeraire_index, 0, 0]
            t0_ratio = self.lmm.DF[start_bond, 0, 0] / t0_numeraire_value
            bonds[start_bond - start_bond, j] = np.mean(self.lmm.DF[start_bond, j, :] /
numeraire_value) * t0_numeraire_value
            ratio[0, j] = np.percentile(self.lmm.DF[start_bond, j, :] / numeraire_value, 5) /
t0_ratio
            ratio[1, j] = (np.mean(self.lmm.DF[start_bond, j, :] / numeraire_value)) / t0_ratio
            ratio[2, j] = np.percentile(self.lmm.DF[start_bond, j, :] / numeraire_value, 50) /
t0_ratio
            ratio[3, j] = np.percentile(self.lmm.DF[start_bond, j, :] / numeraire_value, 95) /
t0_ratio
        np.savetxt('10_year_ZC_martingale_test.csv', ratio, delimiter=',')

    def get_expectation_of_zero_coupon_bond(self, zero_coupon_index):
        forward_rate_index = zero_coupon_index - 1
        product = 1/(np.ones(self.number_of_sims) + self.time_increment*self.forward_sims[0,0,:])

        for i in range(1, forward_rate_index+1):
            product = product/(np.ones(self.number_of_sims) +
self.time_increment*self.forward_sims[i,i,:])
        output = np.mean(product)
        return output

    def check_diffusion_has_zero_mean(self):
        number_of_tests = 40
        mean = np.zeros((number_of_tests,12))

        for i in range(number_of_tests):
            diffusion = self.lmm.get_diffusion()
```

```
        mean[i,:] = np.mean(diffusion, axis=1)
    mean_of_mean = np.mean(mean)
```