Comparative Study of Interest Rate Pricing Models

FM4U1/FM413 Fixed Income Markets Dissertation 2025

Candidate Number: 41526

Word Count: 5997

ABSTRACT

This paper aims to explore the empirical differences between the commonly used interest rate diffusion models: Hull White 1-Factor, Hull White 2-Factor (G2++) and the Libor Market Model. There exists a research gap primarily along two dimensions. The first is the empirical difference between the model performance in diffusing negative interest rates. The second is the practical applicability of the models in post-crisis environments. The relevance of this is emphasized post Libor fallback, which has contributed to some change in market microstructure. The empirical performance of the models is tested by modelling the mechanics in code, calibrating the models to market data, and then analysing the accuracy, speed and usability of the models. Consistent with the literature, find that the LMM continues to dominate in accuracy, the Hull White 1 Factor dominates in compute, and the G2++ retains its good balance between accuracy, speed and simplicity. However, the paper finds that the hedging capabilities of the models are underwhelming for exotic derivatives. We conclude that the models continue to have the performance they have had in the past for EURIBOR rates, including both their advantages as well as their disadvantages.

Keywords: Interest Rate Pricing Model, Libor Market Model, Hull White, G2++, Negative Rate, Risk, Hedging

Contents

Chapter 1

INTRODUCTION

FICC derivatives consist of a sizeable proportion of all traded assets in the market. Pricing models are mathematical models that are used to price and hedge these derivatives. The most popular interest rate diffusion models in use are the Hull White 1-Factor model (**HW**), the G2++ or Hull White 2-Factor model (**G2**), and the Libor Market Model (**LMM**). The HW and G2 models model the instantaneous short-rate and the Libor Market Model models forward Ibor curves. These models give us the forward diffusion of interest rates that can be used to calculate the fair value price of any payoff or derivative.

1.1 Background and Motivation

The literature that empirically compares these models has not been updated for the current state of the market. There have been changes in both, the market structure as well as the pricing models themselves.

- **Market Structure**:

  Firstly, Libor has been discontinued which has caused banks to modify their IBOR calculations or to adopt RFRs (SOFR, SONIA). Secondly, the nature of the macroeconomy and markets are fundamentally different post crises. The world has witnesses major historical events over the last couple of years. Covid, quantitative easing and

macroeconomic instability have caused structural changes to the market.

- **Model Updates:**

  There have been a few extensions to the LMM that have not been studied alongside the other short-rate models during periods of negative interest rates. The Constant Elasticity of Variance, (CEV) Libor Market Model is now the standard for LMM implementations. However, there have been few studies empirically comparing the two in the negative rates regime.

The goal of this effort is to address this research gap by comparing the fixed/local vol models on Euribor 3M Swapions across different strikes, expiries and configurations across time. The models are judged on accuracy, speed, resource requirements and greeks. The research design is elaborated in section 3.2.

1.2 Problem Statement

This paper aims to gauge the performance of the constant/local vol interest rate models in post crisis markets. This will allow us to understand the limitations of the models and their stochastic vol extensions in the current market. In particular, this paper aims to elaborate on the following:

1. To comparatively study the performance of the Hull White, G2++ and Libor Market Model across accuracy, compute and calibration complexity.

2. To study the effects of negative rates on displaced LMM and Local Vol CEV LMM.

Chapter 2

LITERATURE REVIEW

Interest rates are one of the more challenging asset classes to model due to their complexity and simultaneous rate curves. The major work around IR pricing models started around the 1970s and continued until the early 2000s with the Libor Market Model coming out in 1997 and the SABR model coming out in 2002. The number of studies comparing these models peaked alongside this model development period. The G2 and HW models are generally used for simpler derivatives. The LMM dominates in pricing complex, non-Markovian payoffs, and is hence the go-to model for exotics and curve consistent pricing.

The literature on IR pricing models suggests a rich variety of model use cases based on each model's trade-offs. The HW model (12) has been used to get quick and interpretable closed form analytical formulae for vol derivatives (2). However, research has also consistently shown that the model is not specified enough to capture rich market dynamics, causing the model to be inaccurate, overly simplistic, or both.

The G2 model has long existed as the middle ground between the rich model dynamics of models based on the HJM framework (and LMM), and the simplistic 1 factor HW model. Brigo and Mercurio (3) show that just 2 negatively correlated OU processes give rich enough dynamics to represent yield curves. This improvement comes without loss in analytical interpretability that comes with the simplicity of the 2 factor short rate model. However, the

model is complex enough to attract problems like misspecification and unstable parameter fitting.

The Libor Market Model has long been the forerunner in both accuracy and complexity since its inception in the late 1990s. The model typically diffuses 20 - 100 Ibors simultaneously, which allows it to fit the market curve while encapsulating its rich dynamics. Dariusz Gatarek and Maksymiuk (7) and **(author?)** (Lesniewski) provide an in depth guide on implementing the LMM and its many specifications and parametrisations. Rebonato (19) and Brigo et al. (4) have done extensive work on calibrations of LMMs to swaption vols, which this paper does not use due to direct calibration to market prices.

2.1 Relevant Research

2.1.1 Comparative Analysis

Literature on the comparative studies of these models reveals that there are there are many trade-offs one must consider between deciding between these models. Studies consistently demonstrate that although LMMs achieve superior in sample calibration accuracy, G2 is better at hedging. (9) show that a two factor short rate model performs better at hedging than a LMM that is able to fit market skew. (8) provide empirical evidence on the performance of multi-factor term structure models for pricing and hedging caps and swaptions, supporting the claim. The superior hedging performance is also noted by Mikkelsen (17) post 2008, and is one of the more recent comparative studies published.

In terms of speed, the HW model is able to calibrate to portfolios in real time. G2 models take around 5 minutes for calibration. LIBOR Market Models demand substantially longer processing times, with recent benchmarking studies documenting 10-30 minutes even with modern optimization techniques (7)

These studies show that market microstructure and product complexity are the driving factors between the decision to use one model over the other. This paper will test the models on in-sample and greek accuracy, computational resource requirements and speeds.

2.1.2 Model Limitations under Market Stress

A study by Sebastien Gurrieri and Wong (20) does something similar to the aims of this paper by studying the effect of the 2008 financial crisis on the calibrations of IR models. It was demonstrated that the calibration errors rose from an average of 1.6 % to an average of 5 % during and immediately after the crisis. There have been many studies that document all three models failing under changing market conditions (K. C. Chan (14), (18)).

2.2 Research Gaps

There exist several limitations in the literature in regards to these models. Most empirical comparisons in academia relate to earlier implementations of the LMM. There exists little literature on the comparative study of the present state of the three models.

Despite the clear risk of post crisis markets, most comparative studies done on the topic rely on pre 2008 data. The interest rate markets today exist in a different paradigm. There have been structural changes in the market and the macroeconomy, including great interest rate shocks such as negative BOJ/Euribor rates.

This paper seeks to explore these gaps in literature.

Chapter 3

METHODOLOGY

3.1 Pricing Models

3.1.1 Hull White 1-Factor (HW)

The Hull White model diffuses the instantaneous short rate as an Ornstein–Uhlenbeck (OU) process. The Hull White model extends the Vasicek model by allowing the mean reversion term to be time-dependent, and thus fit to the term stricture of the IR asset.

$$dr_t = k\left(\theta(t) - r_t\right)dt + \sigma\, dW_t^{\mathbb{Q}}, \tag{1}$$

Where:

- k > 0 is the mean reversion speed
- $\theta(t)$ is the mean term structure at t
- σ is the volatility
- $dW_t^{\mathbb{Q}}$ is a standard Brownian under its Risk Neutral measure $\mathbb{Q}$

We can integrate the OU process to derive the a closed for solution to the short rate r:

$$r(t) = e^{-kt}\, r(0) + \big(\, 1 - e^{-kt} \,\big)\big[\, \theta(t) - \theta(0) \,\big] + \sigma\, e^{-kt} \int_0^t e^{ks}\, dW^{\mathbb{Q}}(s)$$

$$Var\big(\, r(t) \,\big) = \frac{\sigma^2}{2\,k}\big(\, 1 - e^{-2kt} \,\big)$$

We calibrate $\theta(t)$ by using the relation (3):

$$B(\, 0,\, t,\, T\,) = e^{-\int_t^T f(0,s)ds} \tag{2}$$

$$\theta(t) = \frac{\partial f(0,t)}{k\,\partial t} + f(0,t) + \frac{\sigma^2}{2k^2}\big(1 - e^{-2kt}\big)$$

f is the continuously compounded forward rate and can be calculated from Bond prices. We can see that this depends on our interpolation scheme via the derivative of f.

Since (2) is log-affine (because of the exponent), we can linearly separate options on rates and solve each component separately. This gives us closed form solutions to Caps, Floors and Swaptions making HW extremely fast for simple derivatives (Jamshidian Pricing Jamshidian (13)).

3.1.2 G2++, Hull White 2-Factor (G2)

The G2++ model (3) builds on top of the HW 1F model by superimposing two correlated OU factors.

$$\begin{aligned} r(t) \;&=\; x(t) \;+\; y(t) \;+\; \theta(t), \qquad (1)\\ dx(t) \;&=\; -a\, x(t)\, dt \;+\; \sigma\, dW_t^{(1)},\\ dy(t) \;&=\; -b\, y(t)\, dt \;+\; \eta\, dW_t^{(2)},\\ \mathbb{E}_{\mathbb{Q}}\,[\, dW_t^{(1)}\, dW_t^{(2)}\,] \;&=\; \rho\, dt \end{aligned}$$

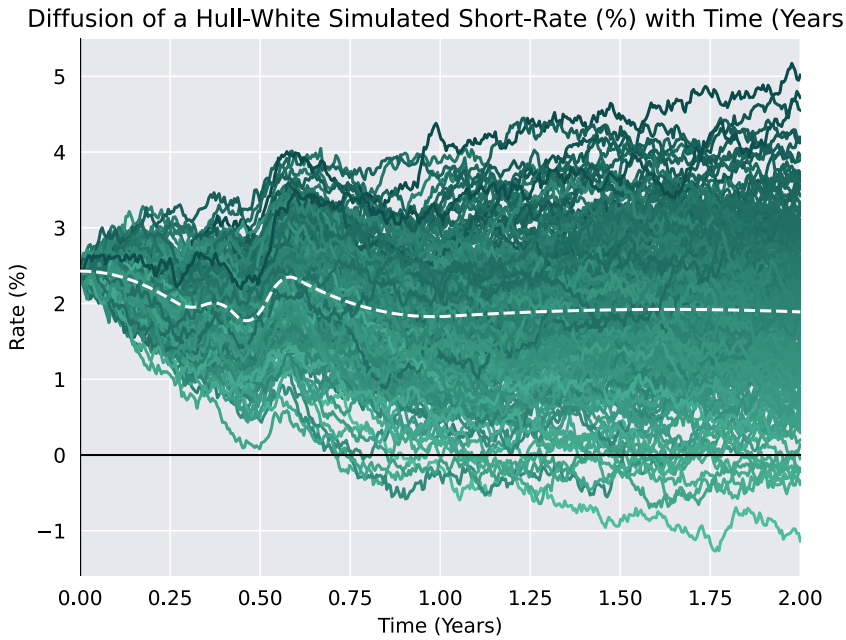


Figure 3.1: HW Diffused Euribor Short Rate

One OU factor accounts for short term movements and the other OU factor accounts for longer term movements. The interpretations of the parameters for G2 are exactly the same as HW but for different time horizons. These two movements are generally negatively correlated to simulate the 'Volatility Hump', which indicates that the vol peaks around the medium term.

G2++ is also log-affine and we will use Simpson numerical integration along the Gaussian random walks of $x(t)$ and $y(t)$ to price swaptions.

3.1.3 Libor Market Model (LMM)

The Libor Market Model diffuses the entire forward Libor curves instead of diffusing the hypothetical short rates like the models above. Unlike its name, the LMM can be used to diffuse any Ibor interest rate asset and even some RFRs with jump extensions (REF). It is a complex multi-curve model that allows us to directly diffuse market observed rates and is the favoured model when pricing most exotic derivatives with the SABR stochastic volatility

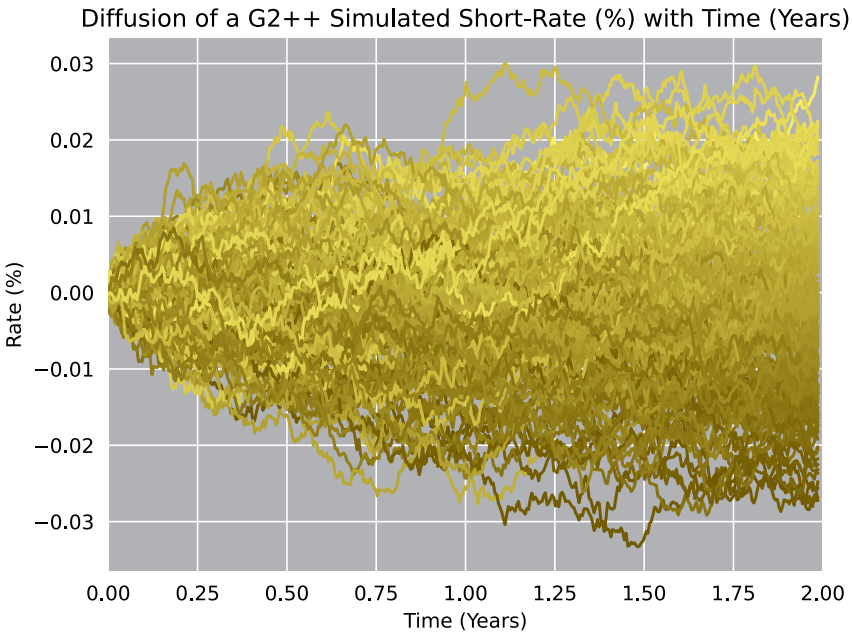


Figure 3.2: G2 Diffused Euribor Short Rate

extension.

Under its own risk measure (bond expiring with the Libor), the 'i'th Libor diffuses log normally without drift:

$$d\,L_i(t) \;= L_i(t)\,\sigma(t,T)\,dW^{\mathbb{Q}_i} \tag{1}$$

By changing the risk measure to a bond expiring with the last simulated Libor, $'k'$, and applying Girsanov's theorem, we can obtain the general diffusion equation for the $'i'th$ Libor as follows:

$$\frac{d\,L_i(t)}{L_i(t)} \;=\; -\,\sigma(t,T_i)\;dt\sum_{j=i+1}^{k}\rho_{i,j}\,\sigma(t,T_j)\,\frac{\delta\,L_j}{1+\delta\,L_j}\;+\sigma(t,T_i)\,dW_i \tag{2}$$

There is no standard python library implementation for the LMM, so we code and deploy

our own model and extensions. We run Monte Carlo Simulations till the option expiry based on the model Stochastic Differential Equations. We then Calculate the swap price based on the Libors we have simulated and the fixed rate. We then obtain the swaption payoff for each path and aggregate across paths to calculate the fair price of the swaption.

We will use a Rebonato vol parametrisation to capture the "vol hump", which has economic interpretability as mid tenor assets have the highest vol (18). The Rebonato/Hump vol is parametrised as the following:

$$\sigma(t) = \big(\alpha + \beta\, t\big)\, e^{-\gamma\, t} \;+\; d$$

Where t is the time to expiry

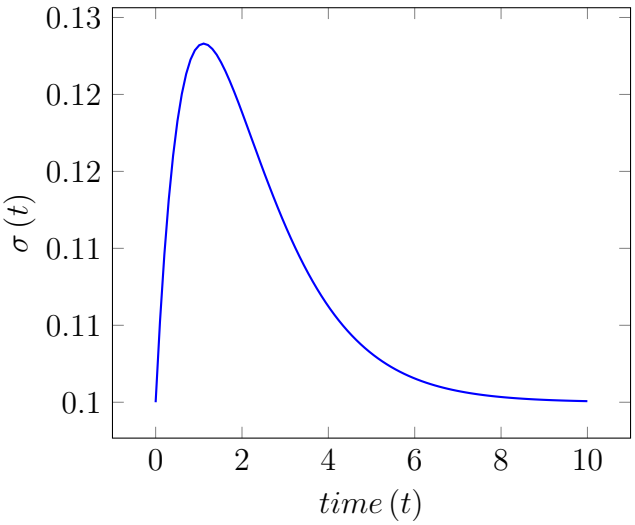


Figure 3.3: Vol Hump plot

There are two main methods of addressing negative rates in the LMM in the literature, which we will explore below.

Displaced LMM

The LMM assumes Log-Normal Diffusion and cannot take negative values. We assume that the Libor actually can go negative until a predetermined lower bound, and the value of the rate is Log-Normal relative to the lower bound instead (11). In effect, a displaced process, $F_i = L_i + d$, is Log-Normal, Where L_i is the ith Libor and d is the predetermined displacement.

$$\frac{d\,F_i(t)}{F_i(t)} \;=\; -\,\sigma(t,T_i)\;dt\;\sum_{j=i+1}^{k}\;\rho_{i,j}\,\sigma(t,T_j)\,\frac{\delta\,F_j}{1+\delta\,L_j}\;+\sigma(t,T_i)\,dW^{\mathbb{Q}_k} \tag{2}$$

At the end of our diffusion, we retrieve our original Libors by subtracting the displacement from F.

CEV LMM

The Constant Elasticity of Volatility (CEV) Local Vol model parametrizes the volatility of the Geometric Brownian Motion of the LMM as:

$$\sigma_i\big(\,t,L\,\big) = \sigma_i\big(\,t\,\big)\;L_i\big(\,t\,\big)^{\beta-1}, \qquad 0 \le \beta \le 1$$

$$dL_i\big(t\big) \;=\; \mu_i\big(t,L\big)\,dt + \sigma_i\big(t\big)\;L_i\big(t\big)^{\beta}\,dW$$

$\beta = 0$ represents a Normal diffusion, and $\beta = 1$ represents a regular Log-Normal diffusion. In it's default state, the LMM cannot be accurately fit to the vol smile as the volatility is independent of the level of the rate. CEV allows us to accurately adjust the level of vol smile based on our choice of β. This is related to the short-rate CEV model published by (5).

The CIR model showed that the rates will remain strictly positive for $\beta = 0.5$ (6). However, as our beta falls under that value, we will have the potential to obtain negative rates. This was considered to be a fault of the model, and industry practitioners resorted to absorb or

Figure 3.4: Low Beta - Volatility Skew

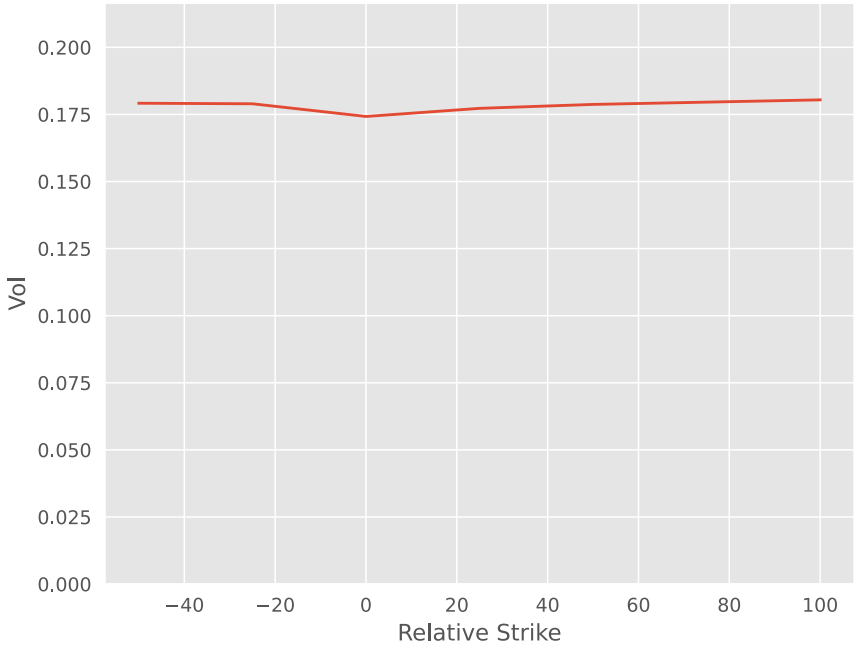


Figure 3.5: High Beta - No Volatility Skew

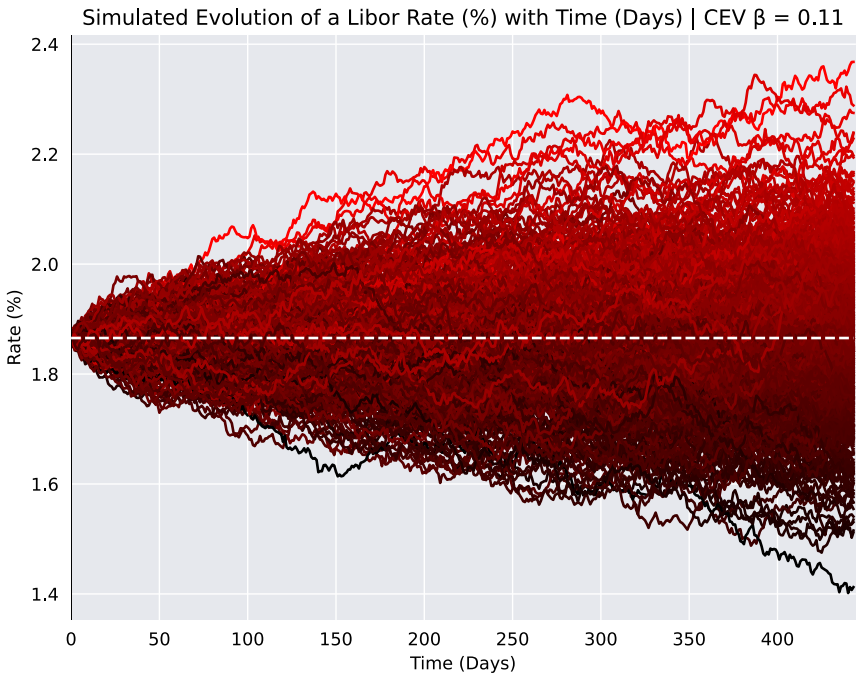


Figure 3.6: Low Beta, Normal Vol-Like

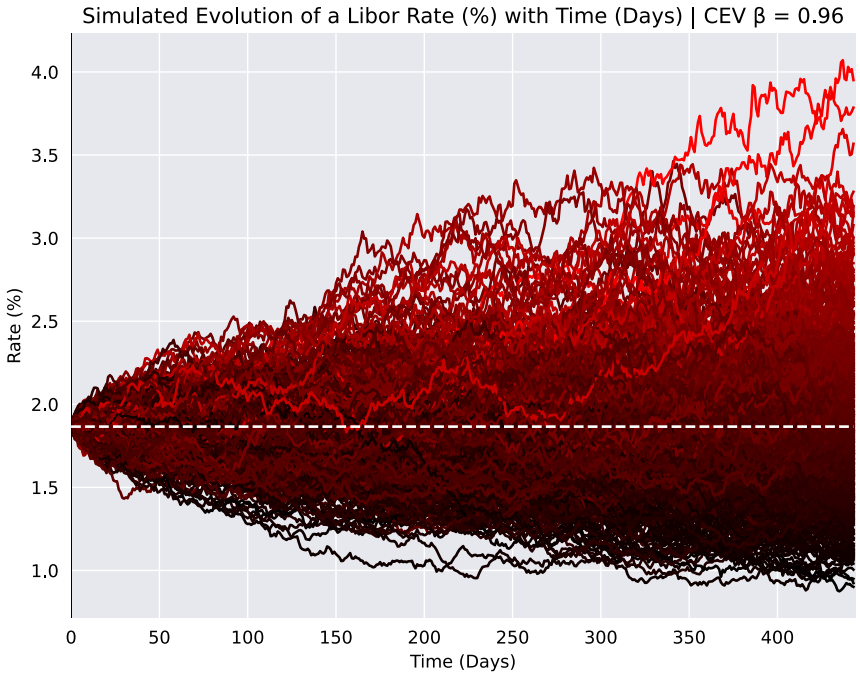


Figure 3.7: High Beta, Log-Normal Vol-Like

reflect (Hagan and Lesniewski (10)) any rates that crossed the zero lower bound. The CEV model in its initial days struggled with negative interest rates, as they were considered to be an impossibility. However, this problem can now be considered to be a feature of the model. We do have to be careful around fractional powers of negative numbers as it can cause errors in simulation.

3.2 Research Design

This section goes over our research methodology design and explanations of why specific modelling choices have been made. The codes for the analyses can be found in the appendix.

Overview: The paper tests for model performance on accuracy (fitted and out of sample), compute speed and parameter fitting ease. We achieve this by fitting the models to Euribor 3M swaption prices from 2020 to mid 2025 across strikes, tenors and expiries. We also check whether model parameters and the underlying economics of the asset diffusion make sense.

3.2.1 Design Specification

There are 2 main types of research conducted for this paper.

- **Market Performance**
- **Individual Model Characteristics**

Market Performance

Training is done on monthly data is from 2020 to May 2025. This includes quotes for Euribor spot rates, FRAs, swap from which the yield curve is constructed. Swaption prices for different relative strikes and expiries are collected. The relative strikes are -200, -100, -50, -25, ATM, 25, 50, 100 and 200 bips and expiries are 3m, 1y, 5y and 10y. The tenor for all

the swaptions is 1y. For every data point, the models are trained, parameter fitting is tested, and predictions are made for out of sample testing. For every data point the following is performed:

1. **Parameter fitting**

   Model Parameters are fit to the day's data. Each model is fit to the following:

   HW: The model is fit to all expiries separately at the ATM strike only

   G2: The model is fit to all expiries separately at the strikes -100, -25, ATM, 25 and 100.

   LMM: The model is fit to all expiries separately at the strikes -100, -25, ATM, 25 and 100. Each day, the initial parameter guess starts with the optimum fit of the previous expiry, allowing us to get quadruple the amount of training iterations versus random parameter initialisation.

   At the end of parameter fitting, we have optimum model parameters for each expiry and training strike (from the list of strikes the model was fit to).

2. **Predictions and Fit Quality**

   From the optimum parameters obtained above, the fit quality is checked and reported. Out of sample testing is done to check whether the model can generalise across strikes. The model predicts the price of the -50, +50 as well as -200, +200 strikes which it has not been trained to, and the accuracy of the prediction is tested. The implication of good out of sample prediction accuracy is that the model can fit properly to the market's expected distribution.

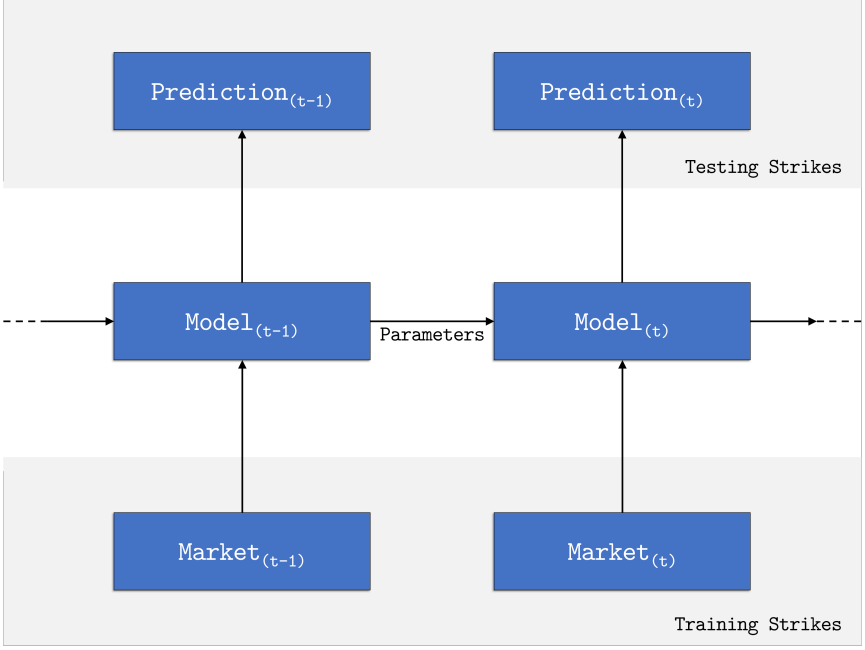


Figure 3.8: Flow Chart of the training process

Individual Model Characteristics

For analysing the underlying economics of the models, we pick a point in time in the market and compute greeks and implied volatilities based on the fitted parameters obtained from the previous section. We perform tests by modifying parameters and market conditions based on the type of analysis.

3.2.2 Data Specification

Choice of Interest Rate

Our paper analyses swaptions on Euribor 3m rates. Our Rate asset choice was must satisfy the following conditions, in order of precedence:

1. **Forward Looking Rates:** After the Libor Fallback, most markets switched to backward looking RFRs. This includes SOFR in the US and SONIA in the UK. These backward looking rates are modelled differently as compared to Ibor rates, which are

forward looking. Due to the backward looking nature of the RFRs, they include jumps in the market. This is modelled as a Poisson Jump Process in addition to the GBM diffusion in the Libor market model (1). However, since the extension fundamentally changes the LMM, we will not consider it to be within our scope. Due to this, our choice for the Rate Asset must be an IBOR rate.

2. **Market Size** Our rate asset must be as widely used as possible.

3. **Minimum Market Idiosyncrasies** The IBOR rates we are modelling must have market dynamics as close to other IBOR rate dynamics as possible. This eliminates the TONAR as the Japanese markets have been stuck around negative or zero rates.

4. **Liquidity and Product Availability** To attribute pricing errors solely to the pricing models, the product availability and liquidity of its derivatives must be high. This minimises market pricing inefficiency, allowing us to isolate model errors.

Due to the above considerations, the Euribor 3M Rate was chosen

Choice of Time Period

We choose the period of 2020 to May 2025 for our analysis. There are negative rates at the early part of the selected period, allowing us to test for our negative rate LMM extensions. After the negative rate period, we have a post crisis market, which is the nature of the current market. This gives us a good balance of data and relevance.

3.2.3 Parameter Fitting

Pricing model parameters are fitted to match market prices. However, we make the choice of fixing some parameters beforehand in order to retain economic interpretability and because it is market practice. The nature of parameter fixings are given below:

HW: We fix the Mean Reversion parameter in the Hull-White Model. This is because it does not make economic sense to adjust it to fit prices everyday. The mean reversion parameter is supposed to represent some fundamental nature of the short rate process, and is assumed to be constant over the duration of the 5 year data.

We calculate the constant mean reversion by fitting our parameters to swaption prices across every month in the 5 year data. We use this optimum mean reversion as our constant mean reversion parameter.

G2: Due to similar reasons as the HW, we fix 1 mean reversion parameter which is fit to market data across time. The other variables are free to move with time. Correlations are not marked as short rate process correlation calculations require knowledge of mean reversions, hence the decision to fix 1 mean reversion parameter instead.

LMM: We fix our correlations for the Libor Market Model. This is a common practice in industry. This is done to retain economic interpretability of the diffused Euribor prices. Correlations are calculated on the daily changes in log prices for every displaced (artificially made positive) Euribor. The displacement parameter is set based on pricing accuracy in the negative interest rate domain.

Methodology:

We use Nelder-Mead (NM) simplex optimisation on the mean squared errors of model prices to calculate the optimum parameters. The NM algorithm is a non Newtonian gradient descent algorithm which works on sampling instead of calculation of exact Jacobians. We cannot use exact gradient descent methodologies (such as BFGS) as we do not have an exact price differential equation for the LMM. This is the main bottleneck for the optimisation of the LMM and a major drawback, which we will discuss in the next section.

3.2.4 GPU Acceleration and Monte Carlo Noise

The LMM is a path dependent model, which forces us to use Monte Carlo simulations for pricing our swaptions (except for an overly simplistic single variable case without Local Volatility). Since Monte Carlo (MC) calculates expectations using sampling instead of numerical integration, the process is inherently noisy. MC variance (uncertainty) is a $O\left(N^{-1}\right)$ process, which forces us to decide our error tolerance and the number of simulated paths jointly.

To maximise our accuracy and use a high number of paths, we will use GPU accelerated pricing. We vectorize our LMM code to ensure all processing is done at the GPU level. This gives us and exponential improvement in compute time versus CPU computation.

Chapter 4

RESULTS

4.1 Accuracy - Price and Greeks

Accuracy is reported in the Root Mean Squared Error (RMSE) and the Mean Absolute Deviation (MAD) of the model predictions versus the market prices in Table 4.1

Table 4.1: ATM Accuracy metrics for HW, G2 and LMM (bips)

Accuracy (ATM)	RMSE	MAD
HW	1.59×10^{-11}	1.35×10^{-11}
G2	1.46×10^{-1}	1.14×10^{-1}
LMM	8.33×10^{-2}	5.96×10^{-2}

RMSE accuracies versus various non ATM strikes of the options are given in Table 4.2.

Table 4.2: Accuracy RMSE for HW, G2 and LMM across strikes (bips)

Accuracy	HW	G2	LMM
-200bps	1.49×10^{0}	9.75×10^{-1}	3.28×10^{-1}
-100bps	2.05×10^{0}	1.31×10^{0}	4.59×10^{-1}
-50bps	1.78×10^{0}	1.26×10^{0}	3.88×10^{-1}
-25bps	1.49×10^{0}	1.28×10^{0}	3.31×10^{-1}
ATM	1.59×10^{-11}	1.46×10^{-1}	8.33×10^{-2}
+25bps	1.66×10^{0}	1.38×10^{0}	3.62×10^{-1}
+50bps	2.20×10^{0}	1.77×10^{0}	4.56×10^{-1}
+100bps	2.80×10^{0}	2.16×10^{0}	5.25×10^{-1}
+200bps	2.74×10^{0}	1.94×10^{0}	5.18×10^{-1}

Accuracy metrics for the negative rates regime are given in Table 4.3. Plots are given in the appendix.

Table 4.3: Accuracy RMSE for CEV and Displaced LMMs for negative rates (bips)

Accuracy	Displaced	CEV
-200bps	1.56×10^{0}	3.95×10^{-1}
-100bps	1.88×10^{0}	6.00×10^{-1}
-50bps	9.39×10^{-1}	3.59×10^{-1}
-25bps	1.19×10^{0}	2.28×10^{-1}
ATM	2.42×10^{-1}	7.74×10^{-2}
+25bps	1.12×10^{0}	2.16×10^{-1}
+50bps	9.85×10^{-1}	2.52×10^{-1}
+100bps	1.00×10^{0}	1.35×10^{-1}
+200bps	7.80×10^{-1}	1.13×10^{-1}

4.1.1 Hull White:

The Hull white model performs poorly on fitting the price curve (vs Strike) of the swaptions. As we have only 1 free parameter to adjust, we cannot make the model fit the dynamics of the entire price curve. This is a notable limitation of the HW model as we cannot extrapolate prices

The HW model has reasonable predictions for ATM Delta and conditionally on Gamma. Due to this, it can be used for quick calculations for delta hedging a portfolio. As the model does not fit the overall curve well, and that curve shifts are not perfectly parallel in reality, it does not provide consistent estimates for Vega and Gamma. Only situationally, when the model calibrates well to the market, can the model Vega and Gamma be used.

4.1.2 G2++:

The G2 model fits the curve well. The fit gets worse as we approach deep in the money and out the money strikes. The gains from slight increase in complexity from the HW model is massive. Due to this, the G2 is the superior model for most scenarios if stable fitting can be

guaranteed. We can see from the price plots that the G2 model fit can sometimes be erratic and unstable.

4.1.3 Libor Market Model:

The Libor market model consistently provides the best fit to the price curve, delta and vega of the swaptions.

We found that CEV outclasses displaced libor even in non negative cases. Although negative CEV does not have economic interpretability, using CEV and displaced libor together would not make sense either. We don't run or need displaced Libor for pricing. Displaced LMM with CEV can work together when beta is above 0.5 and Libor gets restricted to positive values.

The Libor Market Model also has the most consistent calculations for Greeks, in particular, Delta and Vega computation. However, this requires many more path simulations as differentiating is a high pass filter and accumulates MC noise. This makes the model unsuitable for continuous hedging, as MC noise will give us temporal fluctuations, and because MC Greeks are more computationally expensive to compute. This effect is pronounced in Second Order Greek (Example: Gamma and Vomma) calculation as the Signal to MC noise ratio becomes poor. The only way to get consistent second order greeks is to increase step size for calculations or to increase the number of paths (by an order of magnitude). This makes the LMM useful for final end of day hedging for exotics instead of continuous hedging required for vol and one-delta derivatives.

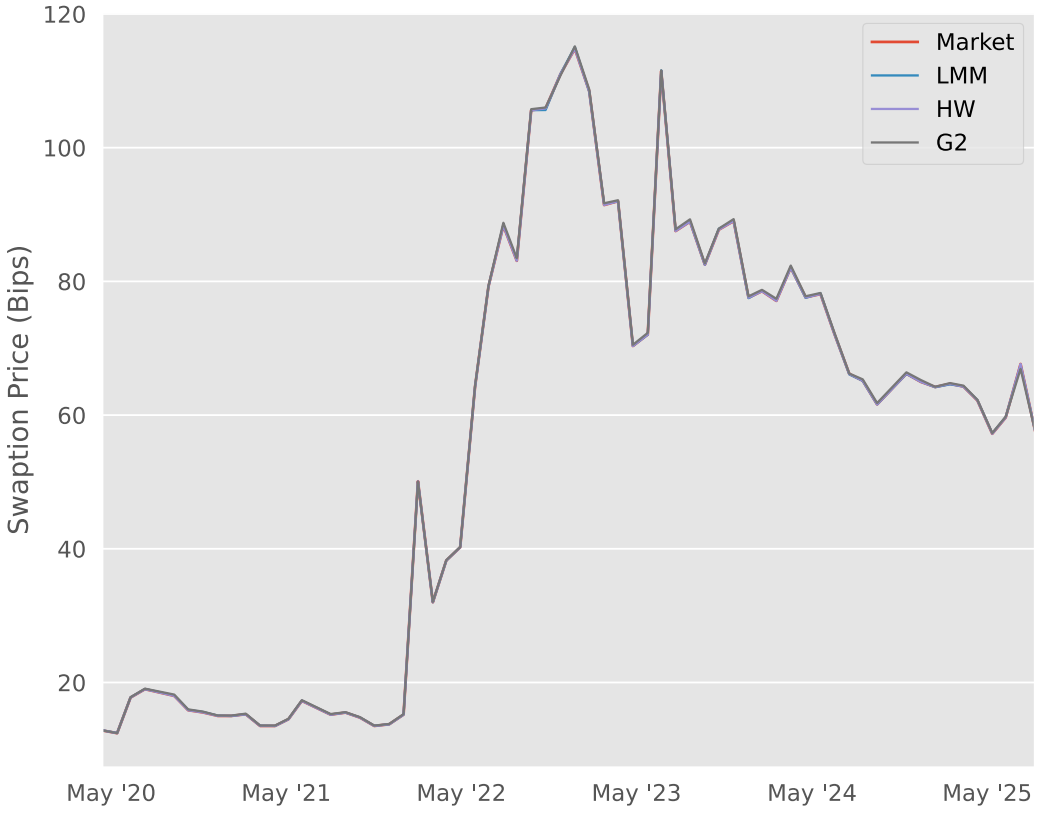
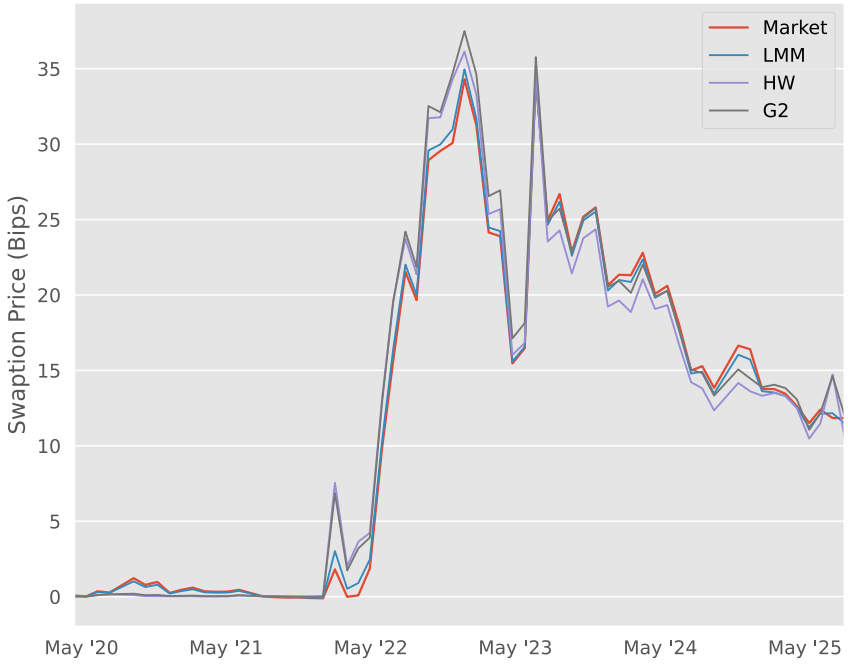
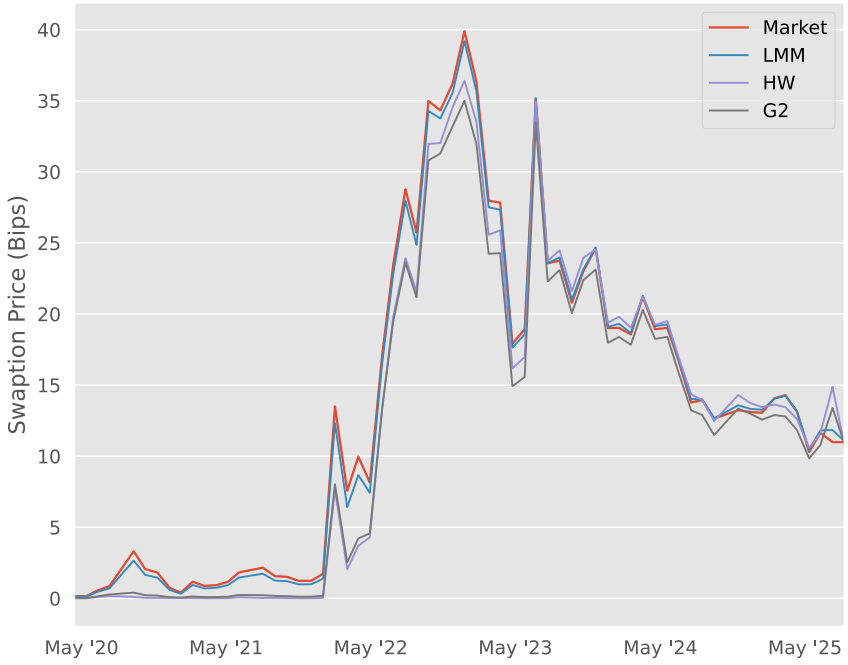


Figure 4.1: 1Y1Y ATM Swaption Price Accuracy - Almost a perfect fit

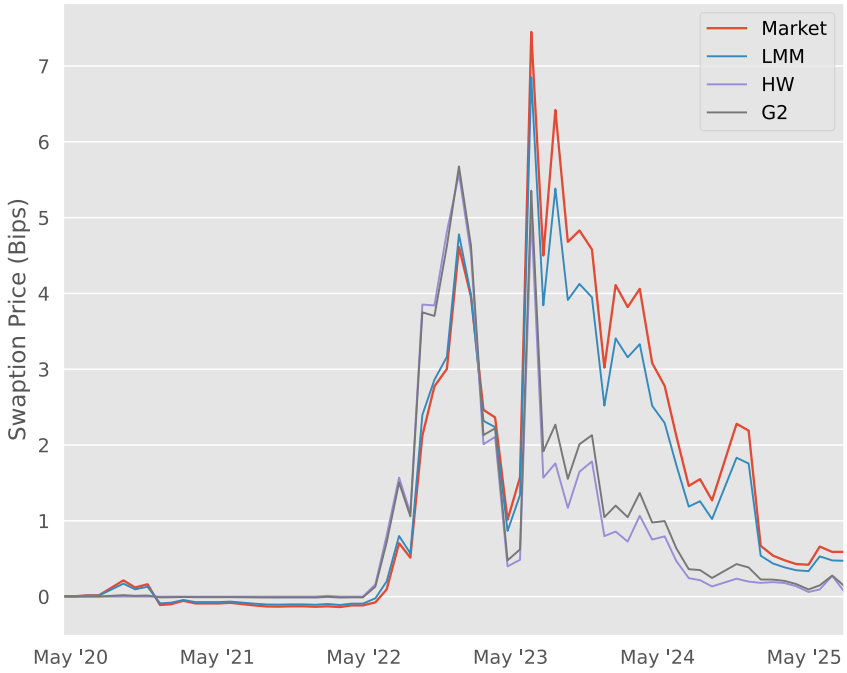


(a) 1Y1Y -50 Strike Swaption Out of Sample Price Prediction

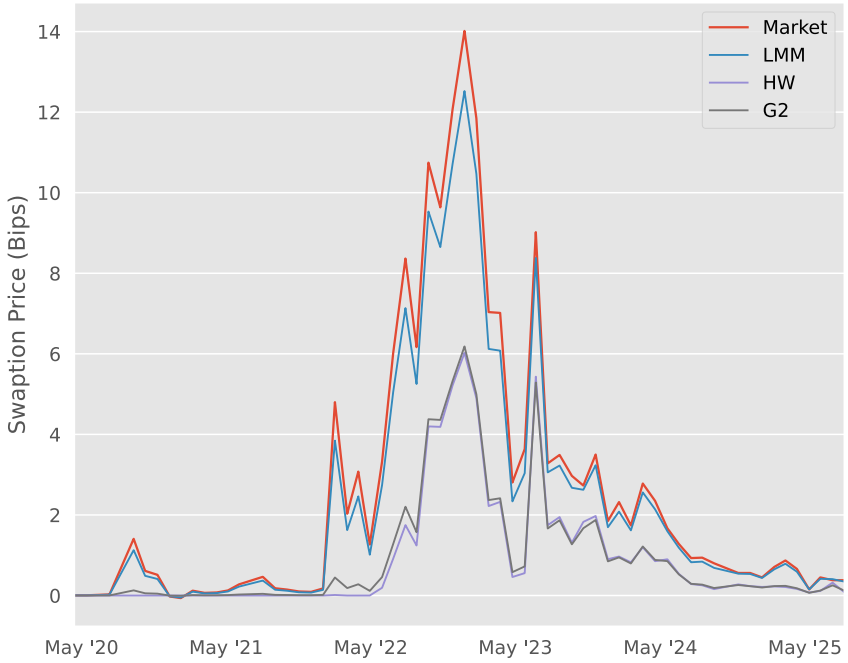


(b) 1Y1Y +50 Strike Swaption Out of Sample Price Prediction

Figure 4.2: Notice the asymmetry caused by incorrect skew fitting



(a) 1Y1Y -200 Strike Swaption Out of Sample Price Prediction



(b) 1Y1Y +200 Strike Swaption Out of Sample Price Prediction

Figure 4.3: HW and G2 get inaccurate towards the wings

4.1.4 Greeks

Delta

Delta is calculated by parallel yield curve shifts. All three models give good prediction for delta, with G2 giving the best performance on accuracy, and HW being the fastest. These two models lose their quality as we move towards the extremities of the vol smile. The LMM gives good estimates of delta throughout the vol smile, but is slower and prone to noise.

Vega

The HW and G2 model have constant vol, which causes them to give poor Vega estimates. The LMM gives good estimates of vega throughout the vol smile. The plot below gives the evolution of ATM vega through the dataset.

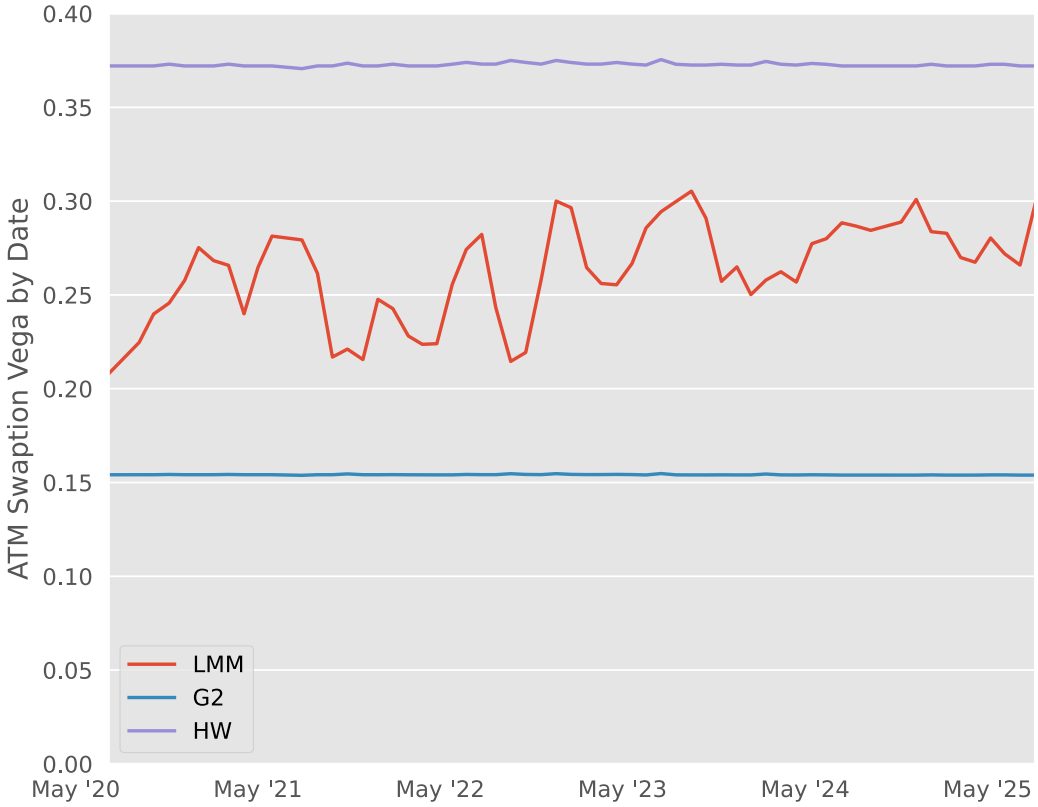


Figure 4.4: ATM Vega vs Time

The LMM this paper works with is a Local Vol CEV LMM, which does not diffuse vol. This causes our LMM vega estimates to be inconsistent. Local Vol LMM vega can still be used if the derivative does not have forward vol dependence, has low vol exposure and is calculated around the strike (near ATM).

Gamma

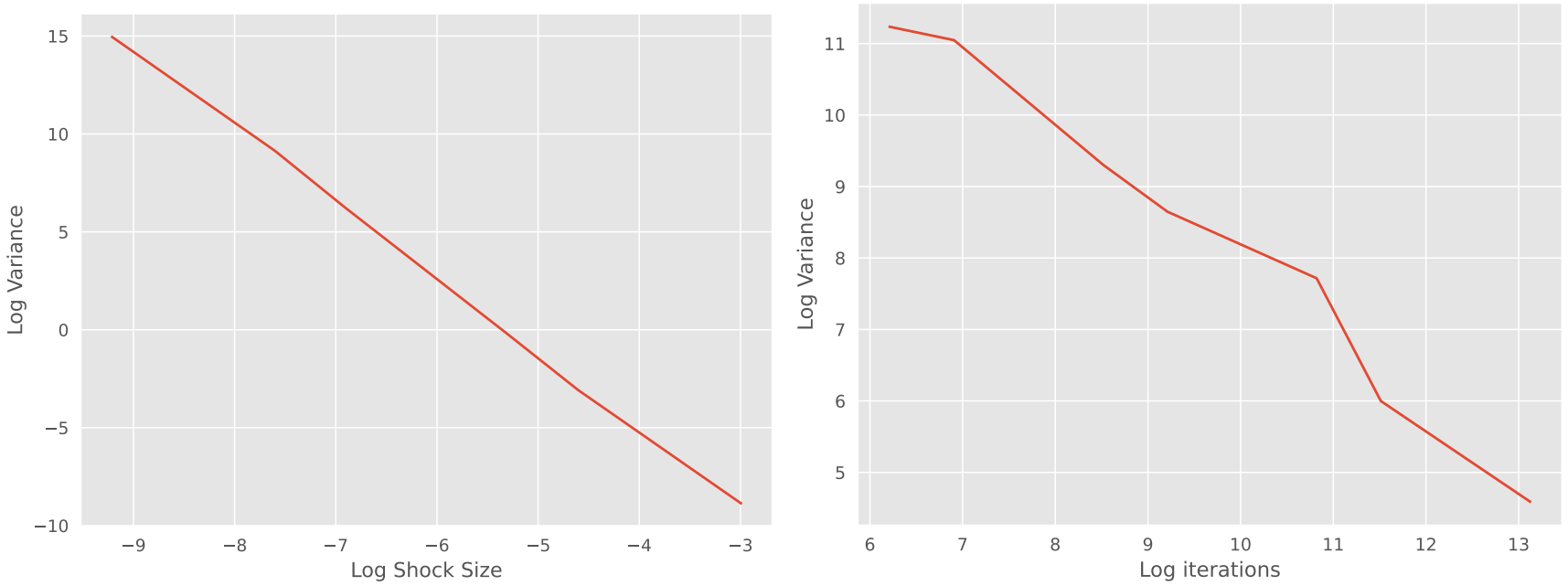
The gamma performance of all models is subpar. HW and G2 give stable gamma estimates, but cannot be trusted with consistency as their dynamics are not rich enough to capture higher order market movements. The LMM has very poor gamma performance. If the number of simulated paths is kept the same as the number in the pricing analysis (50,000), LMM gamma gives a standard error that is 50% that of the expected gamma (approximated). Differentiating can be thought of as a high pass filter. MC noise causes LMM gamma estimates to be too noisy.

The two ways of addressing noisy gamma are increasing the number of sample paths and to increase the shock size. The effect of the former is low because even considering 1 Million path for gamma computation give unsatisfactory results. The problem with increasing the step size is that stable estimates are only reached at very high shock sizes (of around 1%), which makes the calculated gamma a bad representation of the second order spot shock effect. The plot of LMM estimated gamma with step size is given below.

4.2 Computation Speed

The mean, μ, and standard deviation, σ, for computation speed of the models are presented below. Speeds are computed from an average of 50 runs of the model.

Computation speeds are noted to not be statistically different for different strikes or different pricing dates.



(a) Log Variance in Gamma calculation vs Log Bump Size (b) Log Variance in Gamma calculation vs Log Number of Iterations

Figure 4.5: LMM Gamma Noise (Variance)

Table 4.4: Computation Speed metrics for HW, G2 and LMM

Time (s)	μ	σ
HW	1.1×10^{-3}	3.5×10^{-4}
G2	1.5×10^{-3}	5.6×10^{-4}
LMM	78	3.9

The HW model is the fastest and least resource intensive model in the list. The G2 model is fast and not very resource intensive. The LMM is orders of magnitude slower than both the other models. This is expected as paths are being simulated through every time-step for computing expectations on the payoff.

4.3 Ease of Parameter Fitting

The mean, μ, and standard deviation, σ, for optimisation time of the models are presented below. Speeds are computed from an average of 63 runs of the model across time.

Time(s)	μ	σ
HW	0.014	0.0022
G2	0.022	0.0024

Table 4.5: Optimisation Speed for HW and G2

Time(s)	μ	σ
3M	9.23	1.06
1Y	35.9	4.01
5Y	167	19.5
10Y	360	36.5

Table 4.6: Optimisation Speed for LMM vs Expiry

The HW model is the easiest to train, with just 1 free parameter and fast pricing. HW can be calibrated on a CPU easily.

The G2 model takes 1 order of magnitude more compute than the HW model to train due to having 4 free parameters. However, it is still easy to train. G2 can be calibrated on a

CPU comfortably.

The LMM is orders of magnitude slower and more resource intensive than the other two models. The LMM also takes GPU compute to fit the market as parameter fitting solely on CPU would not be feasible. This is due to the decision of fitting vol parameters directly to market prices. Rebonato vol parametrisation fits LMM vols to Black vols, making the process faster. However, the paper decided to not go ahead with that optimisation decision as it would require picking displaced diffusion parameters for implying Black vol which our LMM does not use.

4.4 Vol Curve/Cube Fitting

The LMM is able to fit a limited part of the Vol Cube. The G2 model is able to fit a limited part of the vol smile. The HW model only fits to a single price on the curve/cube.

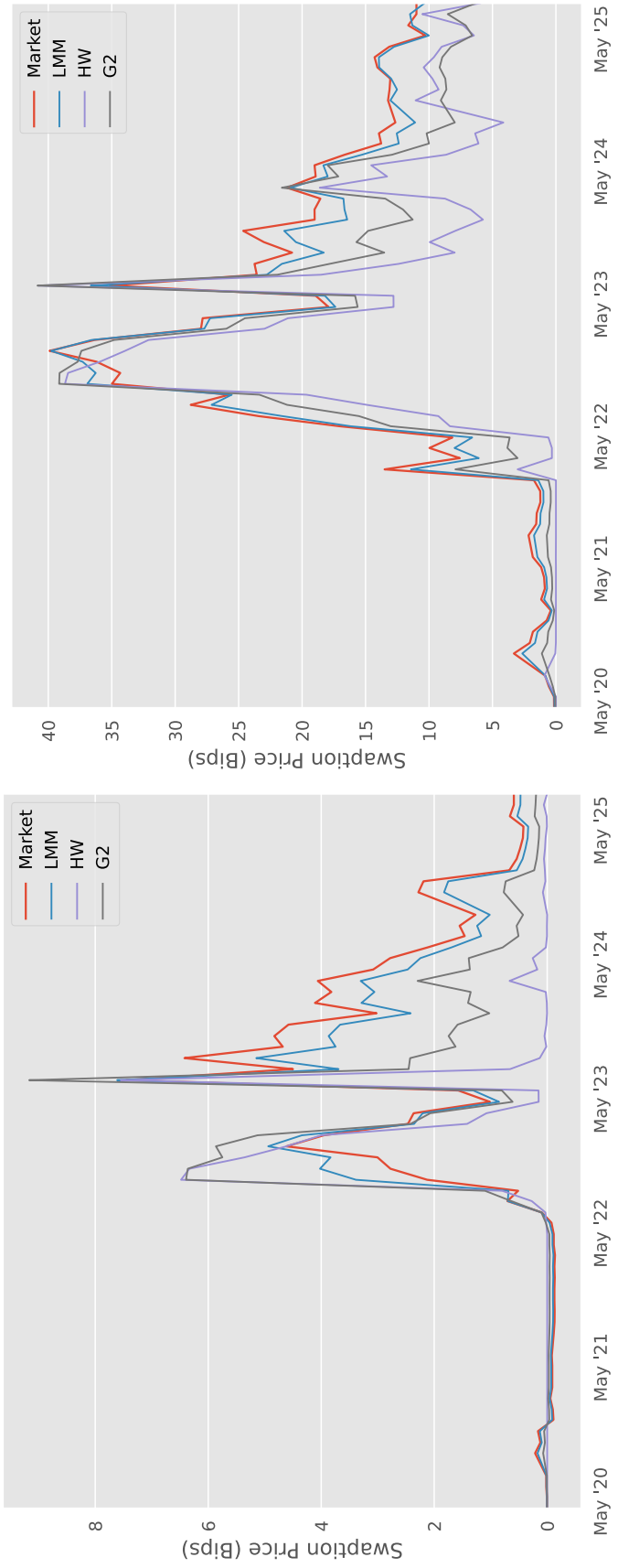
4.4.1 Forward Vol Fitting

Out of all the models, only the LMM can simultaneously fit the vol cube. Even this breaks down for forward vol smiles as they flatten out.

4.5 LMM - CEV versus Displaced

We can find some equivalence between the CEV and displaced diffusion LMMs as shown by (16). However, this equivalence does not extend to the distribution of the simulated ibor forwards.

The displaced diffusion cannot fit the vol smile of the asset by itself. This means that the displaced LMM only generalises across ATM expiries, not strikes. This limits its practical applicability.



(a) 1Y1Y Fitting to Forward Vol, -200 Strike

(b) 1Y1Y Fitting to Forward Vol, +50 Strike

Figure 4.6: Models - Forward Vol Fitting. LMM comes out on top.

The CEV Beta is able to directly adjust the LMM smile across strikes. However, as it is a known limitation of Local Vol, the non stochastic CEV flattens out forward smiles. This causes our forward smiles to become inaccurate with increasing time. Hence, the CEV LMM fits across the strike space and expiry space, and to a shorter strike space.

Chapter 5

DISCUSSION

Given our analyses, the ranking of the models are given in Table 5.1

Ranking	HW	G2	LMM
Accuracy	3	2	1
Speed	1	2	3
Ease of Use	1	2	3

Table 5.1: Model Ranking

Due to the vast differences in the model performance, each one fills its own niche use case.

5.1 Implications and Interpretations

5.1.1 Pricing

HW:

The HW model provides accurate estimates for prices only when fit to the specific strike, expiry and tenor.

G2:

The G2 model gives us a good balance of price curve fitting and calculation speed. The G2 model loses out to the LMM in price curve fitting accuracy. However, the G2 model

is numerically integrated, which gives it noise-free estimates of both first and second order Greeks which makes it the ideal choice for hedging. The gamma estimates will only be consistent for simple derivatives with low deltas.

LMM:

The Libor Market Model provides consistent estimates for the price curve and the Greeks across the Vol Cube (strike, tenor, expiry). It wins out to HW and G2 in most measures of accuracy. The LMM loses out in hedging and in computation of second order Greeks due to MC noise.

In short, the paper finds that the performance of the models is unchanged relative to past literature.

5.1.2 Greeks

The models can estimate delta effectively. However, higher order Greeks might suffer from the following model inaccuracies.

Gamma

G2 gamma can be used under certain cases where gamma sensitivity isn't too high and when the market fit is sufficient. However, G2 is not able to capture true higher order effects of spot movements.

For calculating gamma consistently, we must use the LMM, However, due to Monte Carlo pricing, the LMM is not able to calculate gamma effectively without noise.

Vega

Given the limitations of our models, vega is the most difficult to calculate as only the LMM can somewhat consistently calculate it. Ignoring the stochastic component of volatility, the local vol LMM can still not capture the complete effect of local vol on prices. The CEV parameterisation of vol only allows for specific shapes of the vol curve, which need not give consistent vega. Moreover, the CEV and Vol hump parameterisation do not consider cross effects of time and spot, reducing our confidence in vega.

Implications:

The hedging quality of all the models is not sufficient. This is one of the main findings of this paper. Delta can be hedged effectively by the G2 and LMM. Vega can be estimated by the LMM under situations where vol sensitivity of the asset is low. However, this is as far as the models will go with greeks. The models are incapable of providing consistent estimates for any higher order greeks due to under-specification or Monte Carlo noise. This is a core limitation of the current state of IR modelling literature. The implications of this finding are enumerated below:

1. For exotic derivatives that have non-Markovian payoff dependences, there exists no consistent way of estimating higher order greeks. Even considering the local vol LMM's limitation of not diffusing vol, consistent estimates for gamma are only obtained stably at a spot bump of 1% - which is too big to accurately capture 2nd order risks. The same can be said for Volga (second order vol) and Vanna (second order vol and spot) risks as well.

2. Due to the inconsistency of higher order greek estimation, the hedging of exotic derivatives will be inconsistent. This is a concern as they have greater higher-order risk than vol or 1 delta derivatives. For example, a sudden spot shock (what gamma represents)

is likely to affect the price of a callable swap more compared to a swaption. This uncertainty is exacerbated considering higher-order risks exist along the spot and vol space, and that cross term effects can cause underestimation of risk.

3. The industry handles this problem by applying bigger spreads to exotic products and by approximating the cumulative effects of higher order market movements through model reserves. The latter works on the assumption of the Fundamental Theorem of Asset Pricing, where the price of hedging should equal the price of the asset. However, since the risks and model reserves become unclear, there is no solid counterfactual evidence to compare prices against. This may result in vast underestimation of risk.

4. This also implies that the only class of these products that can be effectively priced and hedged must rely on Monte Carlo noise reduction methods such as using Control Variates from related vol derivatives.

In summary, these models only excel at calculating delta, and cannot be used for high quality gamma and vega calculations in all scenarios.

5.1.3 Negative Rate Pricing - LMM

As CEV is shown to be necessary for accurate fitting to the vol cube, the displaced LMM might seem redundant. Purely for pricing, using both together seems parametrically over-identifying a problem that can be fixed by picking any one modelling choice. That does make sense if we only wish to deploy a stand-alone LMM with CEV, and use an extra parameter for specifying vol that will improve accuracy instead of displacement.

However, in practical applications, the LMM is linked to other pricing models that require displacement. For example, the Rebonato vols are fitted to Black implied vols, which are log normal and need displacements. In such scenarios, it makes sense to use displaced diffusion

and CEV together.

5.2 Limitations

We define the items outside the scope of this paper below:

1. **No Stochastic Volatility**

   We don't be considering the stochastic volatility modelling as part out our LMM. Although the LMM can fit the current vol smile, the forward vol smile tends to flatten out with time. This is the main scope limitation of the paper as most exotics use some variant of the LMM with Stochastic Vol.

2. **No OIS Discounting**

   Derivatives traded in the market are discounted using the OIS rate, which would be ESTR in our case. We discount our prices using the same Ibor rates. The effects of single curve pricing are minimal due to the following considerations:

   - **Forward Premiums:** Forward premiums for pricing negate the effects of OIS discounting as the discounting starts only at the expiration of the option.

   - **Internal Consistency:** Since all pricing models simulate and use the same Ibor rates, they are all internally consistent and comparable. The relative characteristics of the pricing models will not change.

   This however, this still remains as a blind spot of the paper as there can be cross effects not captured here.

3. **Limited Tenors:**

This paper restricts its scope of (swap) tenors to 1y. However, this effect is not major as we can estimate the accumulated time inaccuracy directly by using earlier swaption model parameters to price swaptions with later expiries.

Chapter 6

CONCLUSION

This paper has compared the performances of the Hull White, G2++ and the Libor Market Model on various metrics which are found to be consistent with the literature. The LMM dominates in accuracy and market fitting. The G2 is a good balance between speed and accuracy for most situations. The HW model can be used for fast delta hedging for simple derivatives.

The paper has found limitations in the applicability of the three models in hedging and risk management. The three models can provide reasonable estimates for swaption delta. Only the Libor Market Model provides consistent estimates for vega as the other two models have constant vols. However, the Local Vol LMM does not take into account of the stochasticity of vol, restricting Local Vol LMM vega to simple payoffs that do not have high vol exposure. The main concern found by the paper relates to the calculation of second order greeks. The short rate models are under-specified to capture higher order effects and the LMM suffers from compounding MC noise. The paper assesses that there currently exist no general models that can estimate higher order greeks effectively, which leaves a literature gap in the pricing and hedging of exotic derivatives.

In the negative rates regime, the log normal assumption of the LMM does not restrict its extensions in their usability. In the general case, the LMM is better off with CEV vol

parametrisation than its original log normal formulation. The paper has also found that displaced LMM cannot replace the CEV model, and is best used as a support for economic interpretability.

Bibliography

[1] Andrei Lyashenko1, F. M. (2019). Looking forward to backward-looking rates: A modeling framework for term rates replacing libor. *SSRN*.

[2] Brigo, D. (2006). *An Empirically Efficient Analytical Cascade Calibration of the Libor Market Model Based Only on Directly Quoted Swaptions Data* -. SSRN.

[3] Brigo, D. and Mercurio, F. (2001). *Interest Rate Models Theory and Practice*. Springer Science Business Media.

[4] Brigo, D., Mercurio, F., and Morini, M. (2005). The libor model dynamics: Approximations, calibration and diagnostics. *European Journal of Operational Research*, 163.

[5] Cox, J. (1975). Notes on option pricing i: Constant elasticity of diffusions, stanford university, 1975.

[6] Cox, J. C., Ingersoll, J. E., and Ross, S. A. (1985). A theory of the term structure of interest rates. *Econometrica*.

[7] Dariusz Gatarek, P. B. and Maksymiuk, R. (2006). *The LIBOR Market Model in Practice*. Wiley.

[8] Frank de Jong, J. D. and Pelsser, A. (2004). On the information in the interest rate term structure and option prices. *Springer*.

[9] Gupta, A. and Subrahmanyam, M. (2008). An examination of the static and dynamic performance of interest rate option pricing models in the dollar cap-floor markets. *SSRN*.

[10] Hagan, P. and Lesniewski, A. (2019). Libor market model with sabr style stochastic volatility. *SSRN*.

[11] Henrard, M. (2012). Libor market model with displaced diffusion: Implementation. *OpenGamma Documentation*.

[12] Hull, J. and White, A. (2015). Pricing interest-rate-derivative securities. *The Review of Financial Studies*.

[13] Jamshidian, F. (1989). An exact bond option formula. *Wiley*.

[14] K. C. Chan, G. Andrew Karolyi, F. A. L. A. S. (1992). An empirical comparison of alternative models of the short-term interest rate. *SSRN*.

[Lesniewski] Lesniewski, A. Interest rate and credit models - libor market model. Baruch MFE Course.

[16] Marris, D. (1999). Financial option pricing and skewed volatility. Master's thesis, University of Cambridge.

[17] Mikkelsen, L. B. (2010). Pricing and hedging interest rate caps: With the libor, hull-white, and g2++ interest rate models - evidence from the danish market. Master's thesis, Copenhagen Business School.

[18] Rebonato, R. (2002). *Modern Pricing of Interest-Rate Derivatives : The LIBOR Market Model and Beyond*. Princeton University Press.

[19] Rebonato, R. (2003). Term-structure models: a review. *QUARC - Royal Bank of Scotland*.

[20] Sebastien Gurrieri, M. N. and Wong, T. (2010). Calibration methods of hull-white model. *SSRN*.

Appendix A

The Fundamentals

A.1 The Market

The primary assets that are traded in FICC markets are the interest rates and interest rate spreads. Both of these are modelled similarly. Interest rates belong to two types:

- **OIS Rates (Overnight Indexed Swap):**
  These are overnight rates and are primarily used for discounting. Examples are SONIA, SOFR and ESTR.

- **IBOR Rates (Interbank Offer Rates):**
  These are the rates at which banks lend to each other for predetermined tenors. These were the main type of rates that were used for floating rates before the Libor Fallback. Example: LIBOR, EURIBOR, TONAR, etc.

Ibor rates are quoted in Annual Percentage Rates. This means that they are quoted in simple interest terms relative to the year, but are compounded normally. For example, for a

sequence of Ibors of the same tenor, the discount rate at the current time till T_F is given by:

$$D(0,T_F) = \prod_{i=1}^{F} \frac{1}{1+\delta L(0,T_i)}$$

Where $L(0,T_i)$ is the Libor maturing at T_i with a tenor year fraction of δ.

A.2 Swaps

Swaps are derivatives which allow two parties to swap between a fixed rate and a floating rate. A payer swap is a swap where the buyer pays the decided fixed rate and is paid the floating rate. A receiver swap is the opposite. Payments are made at a predetermined schedule, and can be different between the fixed and floating legs. Swaps are traded at par rate or zero price, where the Net Present Value (NPV) of the fixed leg is equal to the NPV of the floating leg.

For a swap that is discounted at the floating rate it uses, the NPV of a unit notional is calculated as follows:

$$NPV_{Float} = 1 - D(0,T)$$

$$NPV_{Fixed} = r_{Fixed} \sum_{i=1}^{F} D(0,T_i)$$

$$P = NPV_{Float} - NPV_{Fixed}$$

Where T_i is the 'i'th fixed rate coupon payment time.

The fixed rate is chosen to set the price, $NPV_{Float} - NPV_{Fixed}$ to 0.

A.3 Swaptions

Swaptions are the right, but not the obligation, to enter into a predetermined swap at the swaption expiration date. An At The Money (ATM) swaption is a swaption that has its underlying swap set at the par rate. The expiration time of the swaption is called the expiry and the tenor of the underlying swap is called the tenor of the swaption. Swaptions can either have a European, Bermudan or American exercises. Our model will be restricted to European exercises for the sake of comparability. The fair price of a payer swaption is calculated as the following:

$$P = \mathbb{E}_{\mathbb{Q}}\left[(\, NPV(\, S\,) - NPV(\, K\,))^{+}\right]$$

Where S and K represent the floating and fixed legs respectively.

The payoff of a simple swaption can be visualised as follows: In the market, most ATM

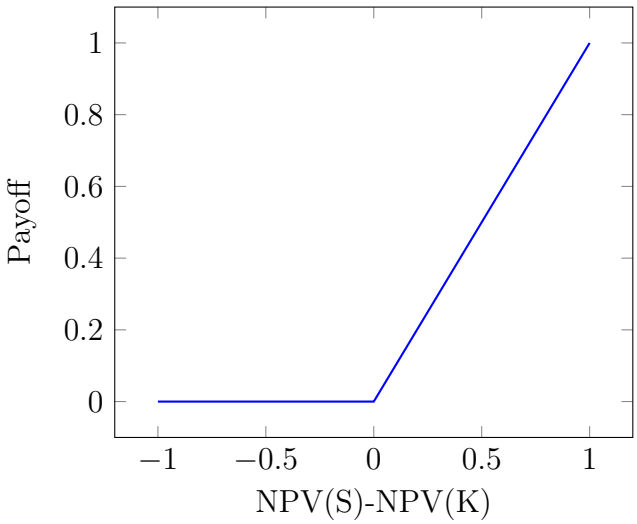


Figure A.1: Single Swaption Payoff

swaptions are traded as straddles instead of regular single swaptions. A straddle swaption is a combination of an ATM payer swaption and an ATM receiver swaption. A receiver swaption

is a swaption where the underlying swap is a receiver swap. All the ATM swaptions we have analysed in the paper are straddle swaptions. The payoff of a straddle swaption is shown below.

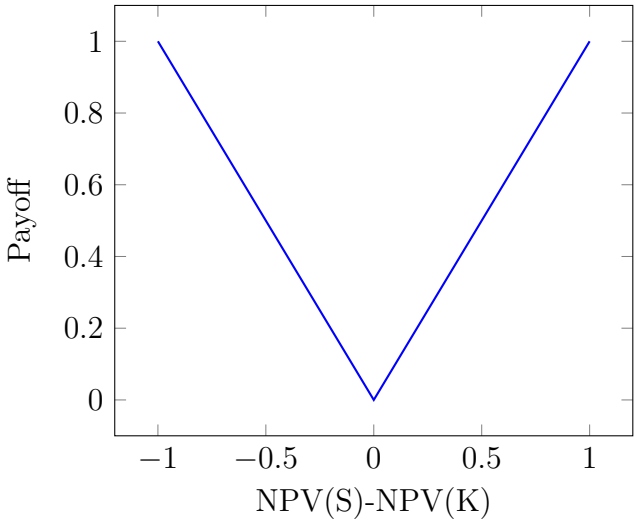


Figure A.2: Straddle Swaption Payoff

For our analysis, we will use Forward Prices/Premiums, which only discount the swaption till the option expiry as that is when the payment is supposed to be made for this payment type.

Appendix B

GPU Acceleration

The choice to use GPU acceleration was necessary to fit the LMM. The metric for accuracy was chosen through the following method:

- **Resource Availability:** How much computational/time ability do we have available to run and train the model.

- **Precision Requirements** What is the necessary standard for accuracy we need to complete the objective.

For this project, we define our standard in error for our model as the following - The probability of the predicted price being greater than 1% away from the actual price must be lesser than 1 in 1000. Numerically, this threshold is around 10,000 simulated paths.

We can see from the calculation speed plot above that the computation time for GPU processed pricing is roughly $O\big(1\big)$. This is because the bottleneck in pricing is data transfer (I/O) between the GPU and the CPU instead of the computation itself. Due to this, we have very little downside to using the maximum number of paths our GPU RAM can stably allow. The GPU RAM usage is directly proportional to the $Simulation\ Time\ \times\ Number\ of\ Paths$. We run our code on an NVIDIA RTX 3080 with 10GB of VRAM which allows us to run

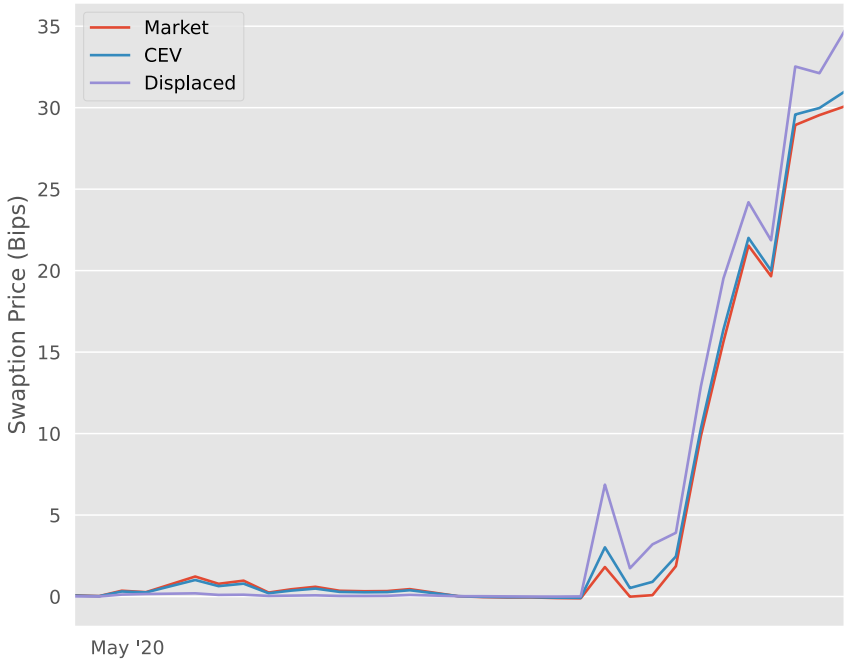
300,000 paths simultaneously in around just a second.

A big downside to the noisy estimates of price is its effect on numerical optimization. As we use a simplex search method to compute the function optimum instead of gradient calculation methods, noise in the pricing process can drastically increase optimisation load and time (and might be unstable for a very low number of paths due to differing local minima). Since our MC constraint is the GPU RAM, simultaneously pricing many securities across strikes and expiries restrict the maximum amount of paths simulated. Due to this fact, we use a mix of 50,000 and 30,000 paths for both, parameter fitting as well as pricing (for internal consistency). Since both of these are well above the required threshold of 10,000, we consider our MC pricer to be sufficiently accurate for our task.

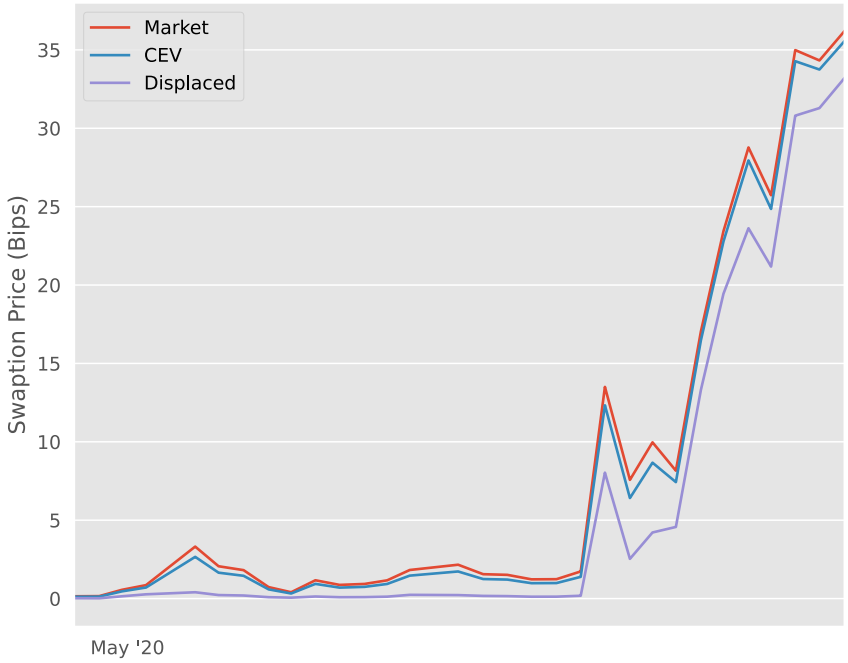
Appendix C

Extra Plots

C.1 Negative Rates Regime



(a) Negative Rates, -50 Strike



(b) Negative Rates, +50 Strike

Figure C.1: LMM Negative Rates Fitting

Appendix D

Python Code

Main LMM file:

```
import numpy as np
import cupy as cp
import scipy as sc
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
import QuantLib as ql
import yieldcurve as yc
import time
import funcs as f
import swaption as swp

#plt.rcParams["figure.facecolor"] = "grey"
mpl.rcParams['mathtext.fontset'] = 'custom'
mpl.rcParams['mathtext.rm'] = 'Bitstream Vera Sans'
mpl.rcParams['mathtext.it'] = 'Bitstream Vera Sans:italic'
```

```
17 mpl.rcParams['mathtext.bf'] = 'Bitstream Vera Sans:bold'
18
19 plt.style.use('ggplot')
20
21 correlations = np.ones((4, 4)) * 0.75
22 for i in range(len(correlations)):
23     correlations[i, i] = 1
24
25 default_lmm_options = {'model':{'correlations' : correlations,
26                                 'lmm_displacement': 0/100, # % to fraction
                                    ↪  earlier: 0.8
27                                 'lmm_skew' : 0.30227599,
28                                 'vol_alpha': 0.0081749, 'vol_beta': 0.00546891,
29                                 'vol_gamma' : 0.15334623, 'vol_d': 0.00587788,
30                                 }, # Earlier 0.01, 0.005, 1
31                           'sim':{'paths':50000, 'lowMem':0}}
32
33 # '''
34
35 class lmm:
36
37     def __init__(self, swaption_wrapper, options=None):
38
39         if type(options) == type(None):
40             options = default_lmm_options
41
42         sec = swaption_wrapper.swaption
```

```
43          swaption_type = swaption_wrapper.type
44          self.swaption_wrapper = swaption_wrapper
45
46          self.swaption_type = swaption_type
47          self.options = options
48          self.strike = swaption_wrapper.strike
49          self.expiry = swaption_wrapper.expiry
50          self.tenor = swaption_wrapper.tenor
51
52          self.swaption = sec
53          self.swap = sec.underlyingSwap()
54          self.idx = self.swap.iborIndex()
55          self.yieldcurve = self.idx.forwardingTermStructure()
56          self.current_date = swaption_wrapper.date
57
58          # Value eg: self.swap.fixedRate() * self.swap.nominal() *
            ↪  self.swap.endDiscounts(0)
59          fixed_cash = 0
60          for i, cf in enumerate(self.swap.leg(0)):
61              fixed_cash += cf.amount()    # cf.date() not needed
62          self.fixed_cash = fixed_cash / self.swap.nominal()
63
64          self.daycount = self.swap.floatingDayCount()
65          schedule = self.swap.floatingSchedule()
66          self.calendar = schedule.calendar()
67          self.expiries = schedule.dates()
```

```
68          #self.tenortime = self.daycount.yearFraction(self.expiries[0],
            ↪  self.expiries[-1]) # This gives too high a value. 1.0138
69          self.tenortime = (self.expiries[-1] - self.expiries[0]) /
            ↪  ((self.current_date)+ql.Period('1Y') - self.current_date)
70          self.starts = self.expiries[:-1]
71          self.expiries = self.expiries[1:]
72          self.days_in_year = self.daycount.dayCount(self.current_date,
            ↪  self.current_date + ql.Period('1Y'))
73
74          self.option_expiry = sec.exercise().dates()[0] # Swaption start
75          # option_expiry = self.swap.startDate() # Same val
76          self.swap_maturity = self.swap.floatingSchedule().dates()[-1] #
            ↪  Check if 2 bus day payment delay
77
78          self.delta = 0.5 * ('6M' in self.idx.name() ) + 0.25 * ('3M' in
            ↪  self.idx.name() )
79          self.yearcount =
            ↪  self.daycount.yearFraction(self.option_expiry,self.swap_maturity)
80
81          self.n_libors = len(self.expiries)
82          #self.n_libors = round(self.yearcount / self.delta ) # N of
            ↪  simulated libors. # CHeck what to do with rounding dates off
83
84          self.paths = options['sim']['paths']
85          self.displacement = options['model']['lmm_displacement'] # F = L +
            ↪  displacement. We diffuse F with LMM formula for - rates
86
```

```
87          # Rebonato Vol params
88          self.vol_alpha = options['model']['vol_alpha']
89          self.vol_beta = self.options['model']['vol_beta']
90          self.vol_gamma = self.options['model']['vol_gamma']
91          self.vol_d = self.options['model']['vol_d']
92
93          # Correlation loading
94          self.corr = options['model']['correlations']
95
96          # Skew Loading
97          # Skew is the power of L(t,T) in vol(t,T,L) in GBM diffusion
            ↪  (BARUCH LMM)
98          # dL => vol(t,T,L) = u * dt + vol(t,T) * L(t,T) ^ skew * dW
99          # Skew = 1 => Regular GBM, Skew = 0 => Normal diff
100         if 'lmm_skew' in options['model']:
101             self.lmm_skew = options['model']['lmm_skew']
102         else:
103             self.lmm_skew = 1
104
105         self.cudaoverride = cp.cuda.runtime.getDeviceCount() > 0
106         self.lowMem = options['sim']['lowMem']
107         self.diffusion = None
108         self.calculatedprice = None
109
110     def change_model_params(self, params):
111         # Supply new params as [alpha, beta, gamma]
```

```
112         self.vol_alpha, self.vol_beta, self.vol_gamma, self.vol_d,
            ↪  self.lmm_skew = params
113         self.diffusion = None   # Reset Diffusions
114         self.extracted_diffusions = None
115
116     def vol(self,current_time, expiration_times):
117         if cp.cuda.runtime.getDeviceCount() > 0 and self.cudaoverride != 0:
118             xp = cp
119         else:
120             xp = np
121         vol = self.vol_d + xp.exp(-self.vol_gamma * (expiration_times -
            ↪  current_time)) * (
122                     self.vol_alpha + self.vol_beta * (expiration_times -
                        ↪  current_time))
123         return(vol)
124
125
126     def expected_payoff(self, spot_bump = 0): # Does not work for fixed leg
        ↪  periods > 1 as assumes 1 cashflow
127         # Expected forward NPV
128
129         if type(self.diffusion) == type(None):
130             self.diffuse(spot_bump = spot_bump)
131
132         if cp.cuda.runtime.getDeviceCount() > 0 and self.cudaoverride != 0:
133             xp = cp
134             if type(self.extracted_diffusions) == type(np.array([1])):
```

```
135             self.extracted_diffusions =
                ↪  cp.asarray(self.extracted_diffusions)
136         else:
137             xp = np
138 
139         rates = self.delta * self.extracted_diffusions
140         rates = rates.transpose()
141 
142         discounts = 1/(1+rates)
143         discounts = xp.cumprod(discounts, axis = 1)
144 
145         npv_float = xp.sum(discounts * rates, axis=1).transpose() # Same as
            ↪  1 - D(t0, tf), probably faster. Shift.
146         npv_fixed = discounts[:,-1] * self.fixed_cash
147 
148         difference = npv_float.reshape(-1,1) - npv_fixed.reshape(-1,1)  #
            ↪  Payer swaption payoff
149         difference = difference * ( 1 -  2 *
            ↪  int(self.swaption_wrapper.isShort)) # Reverse diff, if swap is
            ↪  short, reciever
150 
151         if self.swaption_type == 'Single':
152             difference[difference <= 0] = 0
153             price = xp.average(difference)
154 
155         elif self.swaption_type == 'Straddle':
156             difference = xp.abs(difference)
```

```
157             price = xp.average(difference)
158
159         if cp.cuda.runtime.getDeviceCount() > 0 and self.cudaoverride != 0:
160             price = cp.asnumpy(price)
161
162             if self.lowMem:
163                 del self.extracted_diffusions
164             else:
165                 self.extracted_diffusions =
                    ↪  cp.asnumpy(self.extracted_diffusions)
166             cp._default_memory_pool.free_all_blocks()
167
168         if self.lowMem:
169             del self.diffusion
170             self.diffusion = None
171         return float(price)
172
173
174     def diffuse(self, get_full_diffusion = 0, spot_bump = 0):
175         # Spot bump for delta and gamma
176
177         if type(self.diffusion) != type(None):# and
            ↪  ql.Settings.instance().evaluationDate == self.current_date:
178             if get_full_diffusion:
179                 return self.diffusion
180             else:
181                 return self.extracted_diffusions
```

```
182
183         corr = self.corr
184         if cp.cuda.runtime.getDeviceCount() > 0 and self.cudaoverride != 0:
185             corr = cp.asarray(self.corr)
186             self.useCuda(1)
187             xp = cp
188
189         else:
190             self.useCuda(0)
191             xp = np
192         #print(xp)
193
194         ql.Settings.instance().evaluationDate = self.current_date
195
196         timesteps = self.calendar.businessDaysBetween(self.current_date,
            ↪  self.starts[0]) # FIX TO OPTION END!
197
198         diffusion = xp.zeros((self.n_libors, self.paths, timesteps+1))#
            ↪  n_libor x Path x Timestep matrix will be created
199
200         bus_days_in_year =
            ↪  self.calendar.businessDaysBetween(self.current_date,
            ↪  self.current_date+ql.Period('2Y')) / 2
201         expiration_times = []
202         for expiry in self.expiries:
```

```
203
            #expiration_times.append(self.daycount.yearFraction(self.current_date,
            expiry))
204
            expiration_times.append(self.calendar.businessDaysBetween(self.current_d
            expiry)/bus_days_in_year)
205         expiration_times = xp.array(expiration_times).reshape( (-1,1) ) #
            Column vec
206
207         f0 = self.fwdRatesFromCurve() + self.displacement + spot_bump #
            Override fwds
208         f0 = xp.array(f0).reshape((-1,1))
209         f0 = xp.repeat(f0, self.paths, 1)
210         diffusion[:,:,0] = f0
211
212         extracted_diffusions = xp.zeros((self.n_libors, self.paths))
213         extracted_diffusions[0,:] = diffusion[0,:,0]
214
215         time_start = 0 # Start from today
216
217         # Sim till option must be exercised. Then if Market NPV Float >
            NPV Fixed, exercise.
218         time_end = self.calendar.businessDaysBetween(self.current_date,
            self.option_expiry) / bus_days_in_year
219
220         sim_date = self.current_date
221         time = time_start
```

```
222         dt = (time_end-time_start) / timesteps
223
224         for timestep in range(timesteps):
225
226             sim_date = self.calendar.advance(sim_date, ql.Period('1D'))
227             time += dt
228
229             # n x paths
230             dw = xp.sqrt(dt) *
                ↪  xp.random.multivariate_normal(xp.zeros((self.n_libors)),
                ↪  corr, self.paths ).transpose()
231
232             #'''
233             if self.lmm_skew != 0:
234                 # Drifts is (n x paths) matrix
235                 drifts = xp.zeros((self.n_libors, self.paths))
236                 for i in range(self.n_libors):
237                     # not using Frozen drift approx
238                     corrs = corr[i,i+1:]
239
240                     corr_vol_i = corrs *
                        ↪  self.vol(time,expiration_times[i+1:]).transpose()
241                     rate_return_i = xp.abs(diffusion[i+1:,:,timestep]) **
                        ↪  self.lmm_skew / (1 / self.delta + diffusion[i+1:,:,
                        ↪  timestep])
242
```

```
243                 # drift = (1 x k ) X ( k x Paths) = (1 x Paths) - for
                    ↪  each libor
244                 drift = - corr_vol_i @ rate_return_i # Dotting corr_i *
                    ↪  vol_i with delta L /(1+delta L) # With skew
245                 drifts[i, :] = drift # Drifts is 1 x paths array?
246             #'''
247
248             #print(self.vol(time, expiration_times ))
249             if self.lmm_skew == 1:
250                 # Lognormal
251                 diffusion[:,:,timestep+1] = (1 + self.vol(time,
                    ↪  expiration_times ) * (drifts * dt + dw )) *
                    ↪  diffusion[:,:,timestep]
252
253             if self.lmm_skew == 0:
254                 #Normal Diffusion, not theoretically correct, needs drift
                    ↪  term. Only use for reference.
255                 diffusion[:, :, timestep + 1] = diffusion[:,:, timestep] +
                    ↪  self.vol(time, expiration_times) * dw
256
257             else:
258                 # General skew
259                 diffusion[:, :, timestep + 1] = diffusion[:,:,timestep] +
                    ↪  (self.vol(time,
260                     expiration_times ) * (drifts * dt + dw ) *
                        ↪  xp.abs(diffusion[:,:,timestep])**self.lmm_skew)
261
```

```
262             1
263
264         del corr # Loaded into GPU, need to clear
265         diffusion = diffusion - self.displacement
266         self.extracted_diffusions = diffusion[:,:,-1] # Libors at option
            ↪  expiry
267
268         if cp.cuda.runtime.getDeviceCount() > 0 and self.cudaoverride != 0:
269             if self.lowMem == 0:
270                 self.diffusion = cp.asnumpy(diffusion) # Eats too much mem
271         if self.lowMem:
272             del diffusion
273
274         if get_full_diffusion:
275             if self.lowMem:
276                 print("Diffusion not stored in low mem version. Please
                    ↪  switch to hi mem and rerun")
277                 return None
278             return diffusion
279         else:
280             return extracted_diffusions
281
282     def reset_diffusion(self):
283         self.diffusion = None
284         cp._default_memory_pool.free_all_blocks()
285
286     def price(self, paramoverrides = None, spot_bump = 0):
```

```
287         if type(paramoverrides) != type(None):
288             self.change_model_params(paramoverrides)
289         #return(self.expected_payoff(self.diffuse()))
290         price = self.expected_payoff(spot_bump = spot_bump)
291         self.calculatedprice = price
292         return(price)
293 
294     def plotLibor(self, L, save = 0):
295         # '''Plot L'th libor's time paths:
296 
297         if type(self.diffusion) == type(None):
298             self.price()
299 
300         M = np.max(self.diffusion[L,:,-1])
301         m = np.min(self.diffusion[L,:,-1])
302 
303         color_upper = np.array([1, 0, 0])
304         color_lower = np.array([0, 0, 0])
305         fig, ax = plt.subplots()
306         ax.set_facecolor(np.array([230, 232, 237])/255)
307 
308         if save:
309             1
310             #fig.set_dpi(1200)
311             #fig.set_size_inches(8,6, forward=True)
312 
```

```
313         plt.title(f'Diffusion of a LMM Simulated Libor (%) with Time (Days)
            ↪  | CEV   = {self.lmm_skew}', loc = 'center')
314         plt.xlabel('Time (Days)')
315         plt.ylabel('Rate (%)')
316         plt.grid(True)
317         plt.grid(color='white')
318
319         plt.xlim(0,self.diffusion.shape[-1])
320
321         # Ensure that the axis ticks only show up on the bottom and left
            ↪  of the plot.
322         # Ticks on the right and top of the plot are generally unnecessary
            ↪  chartjunk.
323         ax.get_xaxis().tick_bottom()
324         ax.get_yaxis().tick_left()
325
326         # Remove bounding box
327         ax.spines["top"].set_visible(False)
328         ax.spines["bottom"].set_visible(False)
329         ax.spines["right"].set_visible(False)
330         ax.spines["left"].set_visible(False)
331
332         # Remove tick marks
333         #plt.tick_params(axis="both", which="both", bottom="off",
            ↪  top="off", labelbottom="on", left="off", right="off",
            ↪  labelleft="on") # No work
334         ax.xaxis.set_ticks_position('none')
```

```
335         ax.yaxis.set_ticks_position('none')
336 
337         for j in range(self.diffusion.shape[-2]):
338             x = self.diffusion[L,j,-1]
339             scale = (x-m)/(M-m)
340             plt.plot(100*self.diffusion[L, j, :],
                ↪  color=scale*color_upper+(1-scale)*color_lower)
341 
342         if m < 0:
343             # Zero line
344             plt.axhline(y=0, lw=1, color='black')
345         plt.axvline(x=0, lw=1, color = 'black')
346 
347         # Initial Rate
348         plt.plot(100*self.diffusion[L, 0, 0] *
                ↪  np.ones(self.diffusion.shape[-1]), '--', color='white', )
349 
350         if save:
351 
                ↪  plt.savefig('./plots/simexample/beta-'+str(self.lmm_skew)+'_'+str(time.t
                ↪  format='svg', dpi=600, bbox_inches="tight")
352         #plt.show()
353         #plt.savefig('destination_path.eps', format='eps', dpi=1000,
                ↪  bbox_inches="tight")
354 
355     def volSmile(self):
356 
```

```
357         self.paths = 10000
358         f = self.fixed_cash
359 
360         strikelist = [-100, -50, -25, 0, 25, 50, 100]
361         #strikelist = [0]
362         vollist = []
363 
364         t = self.tenortime
365         for strike in strikelist:
366             if strike < 0:
367                 self.swaption_wrapper.isShort = 1
368             else:
369                 self.swaption_wrapper.isShort = 0
370 
371             k = self.fixed_cash+ strike / 1e4
372             self.fixed_cash = k
373             price = self.price()
374 
375             #print(price)
376             def optim(vol):
377                 #print(vol)
378                 d1 = (np.log(f/k) + (t*vol**2)/2)/vol/np.sqrt(t)
379                 d2 = d1 - vol * np.sqrt(t)
380                 if strike >=0 :
381                     c = f * sc.stats.norm.cdf(d1) - k *
                        ↪  sc.stats.norm.cdf(d2) # Forward prem, no
                        ↪  discounting
```

```
382                     return (price-c)**2
383                 elif strike < 0:
384                     p = k * sc.stats.norm.cdf(-d2) - f *
                        ↪  sc.stats.norm.cdf(-d1)
385                     return (price-p)**2
386
387             optimised = sc.optimize.minimize(optim, 0.1,
                ↪  method='Nelder-Mead', bounds=((0.0001,1),),
388                                                 tol=1e-28,
                                                    ↪  options={'maxiter':1000,
                                                    ↪  'disp':True})
389             vollist.append(optimised.x[0])
390
391             self.fixed_cash = f
392         return vollist
393
394     def fwdRatesFromCurve(self):
395         val = []
396         for i, cf in enumerate(self.swap.leg(1)):
397             val.append(cf.amount() / self.delta)
398         return np.array(val)
399
400     def distribution(self):
401         # Dist of last libor
402         plt.hist(self.diffusion[3,:,-2])
403         plt.show()
404
```

```
405     def monteCarloNoise(self, paths, maxiter = 50, plot = 0):
406         self.paths = paths
407         plist = []
408         for i in range(maxiter):
409             plist.append(self.price())
410             self.diffusion = None
411         if plot:
412             plt.hist(plist)
413             plt.show()
414         return np.array(plist)
415 
416     def useCuda(self, choice = 1):
417         self.cudaoverride = choice
418         if choice == 0:
419             if type(self.diffusion) != type(None):
420                 self.diffusion = cp.asnumpy(self.diffusion)
421             cp._default_memory_pool.free_all_blocks()
422         elif choice == 1:
423             if type(self.diffusion) != type(None):
424                 self.diffusion = cp.asarray(self.diffusion)
425 
426     def averageTime(self, paths, iters=10):
427         self.paths = paths
428         tlist = []
429         self.diffusion = None
430         for i in range(iters):
431             t1 = time.time()
```

```
432             price=self.price()
433             t2 = time.time()
434             tlist.append(t2-t1)
435             self.diffusion = None
436         return tlist
437 
438     def greekdelta(self, spot_bump=10e-4):
439         # 10 bips default
440         self.paths = 100000
441         self.lowMem = 1
442         return (self.price(spot_bump=spot_bump) -
        ↪  self.price(spot_bump=-spot_bump)) / 2 / spot_bump
443 
444     def greekgamma(self, spot_bump=10e-4, paths_override = 100000):
445         self.paths = paths_override
446         self.lowMem = 1
447         p  = self.price()
448         return (self.price(spot_bump=spot_bump) +
        ↪  self.price(spot_bump=-spot_bump) - 2*p) / spot_bump ** 2
449 
450     def gammadistpaths(self, N=10):
451         iterlist = [500, 1000, 5000, 10000, 50000, 100000, 500000]
452         gammalist = np.zeros((N, len(iterlist)))
453 
454         for i, iters in enumerate(iterlist):
455             for j in range(N):
456                 gammalist[j,i] = self.greekgamma(paths_override=iters)
```

```
457             print(f'Iteration {iters} done')
458         vars = np.var(gammalist, axis = 0)
459         fig, ax = plt.subplots()
460         fig.tight_layout()
461         plt.plot( np.log(iterlist), np.log(vars))
462         plt.ylabel('Log Variance')
463         plt.xlabel('Log Iterations')
464         ax.xaxis.set_ticks_position('none')
465         ax.yaxis.set_ticks_position('none')
466         plt.savefig(f'./plots/gammavariter' + '_' + str(time.time()) +
        ↪  '.svg', format='svg',
467                     dpi=600, bbox_inches="tight")
468         plt.show()
469 
470         return gammalist
471 
472     def gammadistbump(self, N=10):
473         bumplist = [1e-4, 5e-4, 10e-4, 50e-4, 100e-4, 500e-4]
474         gammalist = np.zeros((N, len(bumplist)))
475 
476         for i, bump in enumerate(bumplist):
477             for j in range(N):
478                 gammalist[j,i] = self.greekgamma(spot_bump=bump)
479             print(f'Iteration {bump} done')
480         vars = np.var(gammalist, axis = 0)
481         fig, ax = plt.subplots()
482         fig.tight_layout()
```

```
483         plt.plot( np.log(bumplist), np.log(vars))
484         plt.ylabel('Log Variance')
485         plt.xlabel('Log Shock Size')
486         ax.xaxis.set_ticks_position('none')
487         ax.yaxis.set_ticks_position('none')
488         plt.savefig(f'./plots/gammavarbump' + '_' + str(time.time()) +
            ↪  '.svg', format='svg',
489                     dpi=600, bbox_inches="tight")
490         plt.show()
491         return gammalist
492
493 def plotsmile(hibeta = 1):
494     strikes = [-100, -50, -25, 0, 25, 50, 100]
495
496     l1 = [0.17916903070938994, 0.1789677048426475, 0.17424173092753706,
        ↪  0.17726529218533552,
497           0.17872857426647643,
498             0.18042922617086588]
499
500     l2 = [0.24469230844650175, 0.21231252300583264, 0.20008583263190224,
501           0.17854023908064598, 0.17326524429070506, 0.1658328033404589,
              ↪  0.1458119651350089]
502
503     if hibeta:
504         data = l1
505         strikes = strikes[1:]
506     else:
```

```
507         data = l2
508
509     fig, ax = plt.subplots()
510     ax.set_ylim(0, max(data)*1.2)
511     fig.tight_layout()
512     plt.plot(strikes, data)
513     plt.ylabel('Vol')
514     plt.xlabel('Relative Strike')
515     ax.xaxis.set_ticks_position('none')
516     ax.yaxis.set_ticks_position('none')
517     plt.savefig(f'./plots/vol_smile_{hibeta}' + '_' + str(time.time()) +
        ↪  '.svg', format='svg',
518                 dpi=600, bbox_inches="tight")
519     plt.show()
520
521 if __name__ == '__main__':
522     rundate = yc.testpd
523     default_lmm_options['sim']['lowMem'] = 0
524     ycurve, ychandle, ind = yc.gen_yield_curve(rundate, 1)
525     print(f'Running on {rundate}')
526     single = swp.create_single_swaption(yc.testqldate, 0, '1y', '1y',
        ↪  ychandle)
527
528     mod = lmm(single)
529     mod.price()
530
531     plotsmile(hibeta=0)
```

Swaption Wrapper used to wrap all derivative data:

```
import funcs
import yieldcurve as yc
import QuantLib as ql

# '''
class swaptionWrapper():
    # Container class only
    # Do not put pricing fn here. THis fn feeds to pricers
    def __init__(self, swaption, type, swaption2=None, strike = 0, expiry =
    ↪  '1y', tenor = '1y', params = None ):
        self.swaption = swaption
        self.type = type
        if type == 'Straddle':
            self.swaption2 = swaption2
        self.params = params
        self.strike = strike
        self.expiry = expiry
        self.tenor = tenor
        self.isShort = strike < 0 # Reciever Swap - You receive fixed, pay
        ↪  float
        self.date = ql.Settings.instance().evaluationDate
# '''
```

```
21
22
23 # Generate Straddle Swaption: payer + receiver
24 def make_zero_fixed_swap(pricing_date: ql.Date,
25                         expiry: str,
26                         tenor: str,
27                         yieldcurvehandle: ql.YieldTermStructureHandle,
28                         ) -> tuple:
29     """
30     Builds the forward-starting swap (payer) that begins in 'expiry'
       ↪ months/years
31     and runs 'tenor' yrs; returns the swap.
32     """
33     swap_start = yc.swapcal.advance(pricing_date, ql.Period(expiry),
       ↪ ql.ModifiedFollowing, False)
34     swap_maturity = yc.swapcal.advance(swap_start, ql.Period(tenor),
       ↪ ql.ModifiedFollowing, False)
35
36     fixed_sched = ql.Schedule(swap_start, swap_maturity,
       ↪ ql.Period(ql.Annual),      yc.swapcal,
37                               ql.ModifiedFollowing, ql.ModifiedFollowing,
38                               ql.DateGeneration.Forward, False)
39     float_sched = ql.Schedule(swap_start, swap_maturity, yc.ind.tenor(),
       ↪ yc.swapcal,
40                               ql.ModifiedFollowing, ql.ModifiedFollowing,
41                               ql.DateGeneration.Forward, False)
42
```

```
43      # 10000 notional as quotes are in bips. Doesnt change strike
44      swap = ql.VanillaSwap(ql.VanillaSwap.Payer, 10000,
45                            fixed_sched, 0.0, yc.swapdcconv,
46                            float_sched, yc.ind, 0.0, yc.ind.dayCounter())
47
48      swap.setPricingEngine(ql.DiscountingSwapEngine(yieldcurvehandle))
49      strike = swap.fairRate()
50      return swap, strike
51
52  def create_straddle_swaption(pricing_date: ql.Date,
53                               expiry_yrs: str,
54                               tenor_yrs: str,
55                               yieldcurvehandle: ql.YieldTermStructureHandle,
56                              ) -> float:
57      """
58      Analytic Jamshidian price of an ATM straddle (payer+receiver),
59      discounted to spot. Always ATM.
60      """
61      swap, strike = make_zero_fixed_swap(pricing_date, expiry_yrs, tenor_yrs,
        ↪  yieldcurvehandle)
62
63      exercise = ql.EuropeanExercise(swap.startDate())
64      ind = yc.ind
65      swapcal = yc.swapcal
66      swapdcconv = yc.swapdcconv
67      # Payer leg ----------------------------------------------------
68      payer_swap = ql.VanillaSwap(ql.VanillaSwap.Payer, 1,
```

```
69                                  swap.fixedSchedule(), strike, swapdcconv,
70                                  swap.floatingSchedule(), ind, 0.0,
71                                  ind.dayCounter())
72      payer_swap.setPricingEngine(ql.DiscountingSwapEngine(yieldcurvehandle))
73      payer = ql.Swaption(payer_swap, exercise)
74
75
76      # Receiver leg ----------------------------------------------------
77      recv_swap = ql.VanillaSwap(ql.VanillaSwap.Receiver, 1,
78                                  swap.fixedSchedule(), strike, swapdcconv,
79                                  swap.floatingSchedule(), ind, 0.0,
80                                  ind.dayCounter())
81      recv_swap.setPricingEngine(ql.DiscountingSwapEngine(yieldcurvehandle))
82      receiver = ql.Swaption(recv_swap, exercise)
83      wrapped_swaption = swaptionWrapper(payer, 'Straddle', receiver, 0,
        ↪  expiry_yrs )
84
85      return wrapped_swaption
86
87  def create_single_swaption(pricing_date: ql.Date,
88                             strike_shift: float, # Relative, bips
89                             expiry_yrs: str,    # Can input string eg '1y' and
                               ↪  int eg. 1
90                             tenor_yrs: str,
91                             yieldcurvehandle: ql.YieldTermStructureHandle,
92                             ) -> ql.Swaption:
93      # Leg 0 is fixed leg.
```

```
94      swap, strike = make_zero_fixed_swap(pricing_date, expiry_yrs, tenor_yrs,
        ↪  yieldcurvehandle)
95
96      exercise = ql.EuropeanExercise(swap.startDate())
97
98      # Since all non ATM swaps are OTM
99      if strike_shift >= 0:
100         type = ql.VanillaSwap.Payer # pay fixed, get float
101     elif strike_shift < 0:
102         type = ql.VanillaSwap.Receiver
103     # Payment leg ----------------------------------------------------------
104     swap = ql.VanillaSwap(type, 1,
105                               swap.fixedSchedule(), strike +
                                  ↪  strike_shift/10000, yc.swapdcconv,
106                               swap.floatingSchedule(), yc.ind, 0.0,
107                               yc.ind.dayCounter())
108     swap.setPricingEngine(ql.DiscountingSwapEngine(yieldcurvehandle))
109     swaption = ql.Swaption(swap, exercise)
110     wrapped_swaption = swaptionWrapper(swaption, 'Single', None,
        ↪  strike_shift, expiry_yrs)
111     return wrapped_swaption
112
113
114 if __name__ == '__main__':
115     import yieldcurve as yc
116     #'''
117     pd = '2025-04-14'
```

```
118     qldate = ql.Date(pd, yc.dateformat)
119     ycurve = yc.gen_yield_curve(pd)
120     mr_test = 0.03
121     vol_test = 0.01
122     swap, strike = make_zero_fixed_swap(qldate, '1y','10y', yc.ychandle)
123     straddle = create_straddle_swaption(qldate, '1y','10y', yc.ychandle )
124     single = create_single_swaption(qldate, 0, '1y','1y', yc.ychandle )
125     singlehigh = create_single_swaption(qldate, 50, '1y','1y', yc.ychandle )
126     singlelow = create_single_swaption(qldate, -50, '1y','1y', yc.ychandle )
127     #'''
128
```

Calibration File:

```
1 import QuantLib as ql
2 import lmm
3 import hw
4 import g2
5 import yieldcurve as yc
6 import swaption as swp
7 import time
8 import datetime as dt
9 import pandas as pd
10 import numpy as np
11 import funcs
12 import scipy
```

```
import matplotlib.pyplot as plt
import pickle
import multiprocessing as mp
import warnings
warnings.filterwarnings("ignore",
    message="cupy.random.RandomState.multivariate_normal is")
warnings.filterwarnings("ignore", message="cupy.random.multivariate_normal
    is")

def loaddata(name):
    with open('./saved/'+name, 'rb') as f:
        x = pickle.load(f)
        return x

def model_prices(pricer_matrix,calib_params = None):

    model_prices = np.zeros(pricer_matrix.shape)

    for i in range(pricer_matrix.shape[0]):
        for j in range(pricer_matrix.shape[1]):
            for k in range(pricer_matrix.shape[2]):
                model_prices[i,j,k] =
                    pricer_matrix[i,j,k].price(calib_params)

    return model_prices

def mse_cost_functiom(target_prices, pricer_matrix, calib_params ):
```

```
37
38      calculated_prices = model_prices(pricer_matrix, calib_params)
39      weights = calculated_prices == np.max(calculated_prices) # More weight
        ↪  to A
40      scale = np.prod(weights.shape) - 1 # could be 1 also
41      weights = weights.astype(np.float32) * scale
42      weights += 1
43      cost = np.average( ( target_prices - calculated_prices) ** 2 ,
        ↪  weights=weights)
44
45      return cost
46
47  def get_target_prices(calib_dates, calib_strikes, calib_expiries):
48
49      calib_strikes_transform_dict = { -200:0, -100:1, -50:2, -25:3, 0:4,
        ↪  25:5, 50:6, 100:7, 200:8}
50      calib_expiries_transform_dict = {'3m':0, '1y':1, '5y':2, '10y':3}
51
52      all_tickers = pd.read_excel( yc.path_to_data + 'swaption_details.xlsx',
        ↪  index_col = 0, sheet_name='Forward Premium')
53      all_tickers = all_tickers.to_numpy()[:,:]
54      tickers = np.zeros((len(calib_strikes), len(calib_expiries)), dtype =
        ↪  '|S19')   # '|S18' works but not for 10y
55
56      for i,strike in enumerate(calib_strikes):
57          for j, expiry in enumerate(calib_expiries):
```

```
58              ticker = all_tickers[calib_expiries_transform_dict[expiry],
                ↪  calib_strikes_transform_dict[strike]]
59              #print(strike, expiry, ticker)
60              tickers[i,j] = str.encode(ticker) # Correct, checked
61
62      target_prices = np.zeros((len(calib_dates), tickers.shape[0],
        ↪  tickers.shape[1]))
63
64      for col in range(len(calib_expiries)):
65          for row in range(len(calib_strikes)):
66              ticker = tickers[row,col]
67              df = pd.read_excel( yc.path_to_data + 'swaption_details.xlsx',
                ↪  index_col = 0, sheet_name=ticker.decode())
68              df = df['Last Price']
69              target_prices[:,row,col] = df[calib_dates].to_numpy() / 10000 #
                ↪  Price is in bips
70
71      #print('Prices Loaded')
72      return(target_prices)
73
74  def get_pricer_matrix(calib_dates, calib_strikes, calib_expiries, model,
    ↪  options = {}):
75      # Returns a matrix of pricers that are later used to calculate
        ↪  price(vol)
76
77      #seclist = []
78      pmlist = [] # Date x Strike x Expiry - Same as target price array
```

```
79      for i, current_date in enumerate(calib_dates):
80
81          qldate = ql.Date(current_date, yc.dateformat)
82          ql.Settings.instance().evaluationDate = qldate
83          ycurve = yc.gen_yield_curve(current_date)
84          ychandle = ql.YieldTermStructureHandle(ycurve)
85          ind = ql.Euribor3M(ychandle)
86
87          #sec_array_strike = []
88          pm_array_strike = []
89          for s, strike in enumerate(calib_strikes):
90              #sec_array_exp = []
91              pm_array_exp = []
92              for e, expiry in enumerate(calib_expiries):
93                  # time_to_expiry = funcs.to_real_time(expiry)
94                  if strike == 0:
95                      sec = swp.create_straddle_swaption(qldate, expiry, '1y',
                            ↪  ychandle)
96                  else:
97                      sec = swp.create_single_swaption(qldate, strike, expiry,
                            ↪  '1y', ychandle)
98                  #sec_array_exp.append(sec)
99                  if model == 'lmm':
100                     if options == {}:
101                         pm_array_exp.append(lmm.lmm(sec))
102                     else:
103                         pm_array_exp.append(lmm.lmm(sec, options))
```

```
104                 elif model == 'g2':
105                     pm_array_exp.append(g2.g2(sec))
106                 elif model == 'hw':
107                     pm_array_exp.append(hw.hw(sec))
108             #sec_array_strike.append(sec_array_exp)
109             pm_array_strike.append(pm_array_exp)
110         #seclist.append(sec_array_strike)
111         pmlist.append(pm_array_strike)
112 
113     return np.array(pmlist)
114 
115 
116 def optimise_model_oneshot(calib_dates,calib_strikes,calib_expiries, model,
    ↪  full_return = 0, x0 = None, options_override= None, plot = 0):
117 
118     target_prices = get_target_prices(calib_dates, calib_strikes,
        ↪  calib_expiries)
119 
120     model_list = ['hw', 'g2', 'lmm']
121     if type(options_override) != type(None):
122         for key in options_override:
123             if key == model:
124                 model_options = options_override[key]
125         else:
126             model_options = None
127     pricer_matrix = get_pricer_matrix(calib_dates, calib_strikes,
        ↪  calib_expiries, model, model_options)
```

```
128
129
130     def cost_wrapper(calib_params):
131         return mse_cost_functiom(target_prices, pricer_matrix,
            ↪  calib_params)
132
133     if model == 'hw':
134         #bounds = ((0.00001,0.6),(0.000001,0.5))
135         bounds = ((0.000001,0.3),)
136         tol = 1e-14
137         maxiter = 500
138     elif model == 'g2':
139         # Alpha, Sigma, Beta, Eta, rho
140         bounds = ((0.0001,1), (0.0001,1), (0.0001,1), (0.0001,1), (-0.85,
            ↪  0))   # Rho negative for vol hump
141         tol = 1e-12
142         maxiter = 200
143     elif model == 'lmm':
144         # Alpha, Beta, Gamma, D, Skew
145         bounds = ((0.0001, 0.5), (0.00000001, 1), (0.0001, 4), (0.0001,
            ↪  0.05), (0.01, 0.99))
146         tol = 1e-10
147         maxiter = 1
148
149     if type(x0) == type(None):
150         if model == 'hw':
```

```
151             #x0 = [hw.default_hw_options['model']['mr'],
                ↪  hw.default_hw_options['model']['vol']]
152             x0 = [hw.default_hw_options['model']['vol']]
153         elif model == 'g2':
154             x0 =
                ↪  [g2.default_g2_options['model']['g2_alpha'],g2.default_g2_options['model
155                   g2.default_g2_options['model']['g2_beta'],
                      ↪  g2.default_g2_options['model']['g2_eta'],
156                   g2.default_g2_options['model']['g2_rho']]
157             #x0 = [0.04,0.04,0.03,0.03,-0.5] # Eta gives problems
158         elif model == 'lmm':
159             temp = lmm.default_lmm_options['model']
160             x0 = [temp['vol_alpha'], temp['vol_beta'], temp['vol_gamma'],
                ↪  temp['vol_d'], temp['lmm_skew']]
161             x0 = [0.00857742, 0.00580225, 0.14930935, 0.00534377,
                ↪  0.30187107] # 500 iters, good fit??
162
163
164     options = {'maxiter':maxiter, 'disp':True}
165
166     if type(options_override) != type(None):
167         for key in options_override:
168             if key not in model_list:
169                 options[key] = options_override[key]
170
171     methods = ['Nelder-Mead', 'Powell', 'L-BFGS-B' ]
172     method = methods[0]
```

```
173
174     if method == 'L-BFGS-B':
175         options['eps'] =  0.001
176
177     t1 = time.time()
178     optimised = scipy.optimize.minimize(cost_wrapper, x0, method = method,
        ↪  bounds = bounds,
179                                         tol=tol, options = options )
180
181     t2 = time.time()
182     time_taken = (t2-t1) /
        ↪  (len(calib_strikes)*len(calib_expiries)*len(calib_dates)) # Time
        ↪  taken per sec to match
183     # Fair to use for LMM and G2 this as HW is calibrated for every K, T,
        ↪  t
184
185     if not full_return:
186         if plot:
187             # ''' For plotting fit
188             xopt = optimised.x
189             print(f'New X: {xopt}\nOld: {x0}')
190             est = model_prices(pricer_matrix)
191             est2 = model_prices(pricer_matrix, xopt)
192
193             plt.plot(est[0, :, 0], marker='o', label = 'mod')
194             plt.plot(est2[0, :, 0], marker='o', label = 'est')
195             plt.plot(target_prices[0, :, 0], marker='o', label = 'target')
```

```
196             plt.legend()
197             plt.show()
198             # '''
199 
200         print('Optimization Finished')
201         return optimised
202     else:
203         return (optimised, target_prices, pricer_matrix, time_taken)
204 
205 
206 def optimise_model_sequential( start_date, end_date, frequency = '1w'):
207 
208     qldate = ql.Date(start_date, yc.dateformat)
209     rundate = yc.swapcal.advance(qldate, ql.Period('0d'))
210     endqldate = ql.Date(end_date, yc.dateformat)
211     strdate = str(rundate.to_date())
212 
213     all_expiries = ['3m', '1y', '5y', '10y']
214 
215     all_strikes = [-200, -100, -50, -25, 0, 25, 50, 100, 200]
216 
217 
218     calib_strikes = {'hw':[0], 'g2':[-100, -25, 0, 25, 100], 'lmm':[-100,
        ↪  -25, 0, 25, 100]}
219 
220     iterlist = {'hw':200, 'g2':200, 'lmm' : 25 } # Put half for lmm as it
        ↪  will iterate twice
```

```
221
222     model_list = ['hw', 'g2', 'lmm']
223
224
225     data = []
226
227     first_row = {} # Contains gen info
228     #first_row.append({'Calib Strikes': calib_strikes})
229     first_row['Strikes'] = all_strikes
230     first_row['Expiries'] = all_expiries
231     first_row['Model List'] = model_list
232     #first_row.append({'Calib Expiries': calib_expiries})
233
234     count = 0
235
236     data.append(first_row)
237     while rundate > endqldate:
238         print(f'Executing Date {strdate}')
239         data_row = {'Date': strdate}
240         skip = 0
241         for model in model_list:
242
243             model_dict = {}
244             print(f'    Running {model}')
245             for i, expiry in enumerate(all_expiries):
246                 # Calibrate
247                 expiry_dict = {}
```

```
248
249                 # Find previous optimal as starting value, should make
                    ↪  optim faster
250                 if i > 0:
251                     # Smaller expiry's x0
252                     x0 = model_dict[all_expiries[i-1]]['fit']['opt']['x']
253                 elif len(data) >= 2:
254                     # Previous timeperiod's x0
255                     x0 = data[-1][model][all_expiries[0]]['fit']['opt']['x']
256                 else:
257                     x0 = None
258
259                 options_override = {'disp': False,
                    ↪  'maxiter':iterlist[model]}
260                 if model == 'lmm':
261                     try: # Prefitting to x0
262                         model_options = lmm.default_lmm_options
263                         model_options['sim']['paths'] = 50000
264                         model_options['sim']['lowMem'] = 1
265                         options_override['lmm'] = model_options
266                         opt = optimise_model_oneshot([strdate], [0],
267
                                                                    ↪  [ex
                                                                    ↪  mod
                                                                    ↪  ful
                                                                    ↪  x0=
```

```
268
                                                                    ↪ opt
269                     x0 = opt.x
270                     model_options['sim']['paths'] = 30000
271                     options_override['lmm'] = model_options
272                 except:
273                     # If no data available on that day
274                     skip = 1
275                     break
276
277             try:
278                 opt, market, fitted, time_taken =
                    ↪ optimise_model_oneshot([strdate],
                    ↪ calib_strikes[model],
279                                          [expiry], model, full_return = 1, x0=x0,
                                             ↪ options_override=options_override)
280             except:
281                 # If no data available on that day
282                 skip = 1
283                 break
284
285             opt['time_taken'] = time_taken
286             expiry_dict['fit'] = {}
287             expiry_dict['fit']['market'] = market
288             expiry_dict['fit']['estimated'] = model_prices(fitted)
289             expiry_dict['fit']['opt'] = opt
290
```

```
291                 del market, fitted, time_taken
292
293                 '''
294                 if 'Market' not in data_row:
295                     data_row['Market'] = market
296                 #'''
297
298                 # Predict across all strikes and expirires
299                 try:
300                     target_prices = get_target_prices([strdate],
                        ↪ all_strikes, all_expiries)
301                 except: # If no data
302                     skip = 1
303                     break
304                 pricer_matrix = get_pricer_matrix([strdate], all_strikes,
                    ↪ all_expiries, model)
305                 t0 = time.time()
306                 estimates = model_prices(pricer_matrix, opt.x)
307                 t1 = time.time()
308
309                 expiry_dict['pred'] = {}
310                 expiry_dict['pred']['market'] = target_prices
311                 expiry_dict['pred']['model'] = estimates
312                 expiry_dict['pred']['time_taken'] = t1-t0
313
314                 model_dict[expiry] = expiry_dict
315             data_row[model]=model_dict
```

```
316             if skip: # At model level
317                 break
318         current_day = strdate
319         qldate = yc.swapcal.advance(qldate, -ql.Period(frequency))
320         rundate = yc.swapcal.advance(qldate, ql.Period('0d'))
321         strdate = str(rundate.to_date())
322         if skip:
323             print(f'Error on {current_day}, moving on to:\n')
324             continue
325         print('-----------------------\n')
326 
327         data.append(data_row)
328         count +=1
329         if count % 10 == 0:
330             now = dt.datetime.now()
331             savestring = './saved/' +'_'.join(model_list)+'_' +
                ↪  str(now.year) + '-' + str(now.month) + '-' + str(now.day)
332             savestring += '_' + str(now.hour) + "-" + str(now.minute)
333             savestring += f'_from_{start_date}_to_{current_day}.pkl'
334             with open(savestring, "wb") as f:
335                 pickle.dump(data, f)
336             print('Data Saved at:', savestring)
337 
338     now = dt.datetime.now()
339     savestring = './saved/'+'_'.join(model_list)+'_'+str(now.year)+'-' +
        ↪  str(now.month) + '-' + str(now.day)
340     savestring += '_' + str(now.hour) + "-" + str(now.minute)
```

```
341     savestring += f'_from_{start_date}_to_{current_day}.pkl'
342     with open(savestring, "wb") as f:
343         pickle.dump(data, f)
344     print('Data Saved at:', savestring)
345     return 1
346 
347 if __name__ == '__main__':
348     calib_options_hw = {'model': 'hw'}
349     calib_options_lmm = {'model': 'lmm'}
350 
351     calib_dates = ['2025-04-09']  # Create from calendar later # Ideally
        ↪  calibrate everyday
352 
353 
354     calib_strikes = [-100, -25, 0, 25, 100]
355 
356     calib_expiries = ['10y']
357 
358     model = 'lmm'
359     temp1 = optimise_model_oneshot(calib_dates,calib_strikes,calib_expiries,
        ↪  model)
360 
361     temp2 = optimise_model_sequential('2025-05-07', '2020-01-01', '1m')
362 
```