A Futures Market Based Short Term Interest Rate Model

Atanas G. Iliev Ameer Talha Yasser

Contents

Abstract

In this paper, we propose an alternative way to calibrate one-factor interest rate models. Instead of using the interest rate yield curve, we suggest that they are anchored to the interest rate futures curve. We implement the Ho-Lee model due to its simplicity and tractability. We fitted this model to the 3 month SOFR futures curve and used it to predict zero-coupon bond prices. Finally, we suggest future steps to expand on a more sophisticated volatility function and using r^{*1} as the mean reverting term in the drift function.

1 Introduction

One-factor interest rate models are fundamental tools in financial mathematics and quantitative finance, playing a crucial role in understanding and modeling the evolution of interest rates over time. These models, including seminal frameworks such as the Ho-Lee, Vasicek, and Hull-White models, focus on the dynamics of a single factor: the short-term interest rate. One-factor models can capture key characteristics of interest rate behavior, such as mean reversion and volatility, which are critical for accurately reflecting market conditions. These models are described using an SDE of the form:

$$dr = Adt + BdW$$

where dW is the stochastic term, A is the drift function, and B is the volatility function.

These models are heavily used for pricing various fixed-income securities, pricing interest rate derivatives, and enabling financial professionals to develop and implement sophisticated strategies for managing interest rate risk. To be able to 'correctly' price fixed income instruments, the drift functions of these models are calibrated to the yield curve[4]. Yield curve fitting has some textbook problems. One fundamental problem with yield curve fitting is the assumption that if market prices of simple bonds are accurately represented by a model at a certain time time, then recalibrating the model after a short period (such as one week) should yield minimal changes in the fitted function[7]. However, empirical evidence shows that this is rarely the case. When revisited after a week, the function often changes significantly,

[1]Neutral Interest Rate

indicating that the model does not hold consistently over time. This discrepancy suggests a fundamental flaw in the model's assumptions or its ability to capture the true dynamics of the yield curve.

We focus on other problems that the use of the yield curve presents. The yield curve can not be considered a reliable estimate of future interest rates, especially in the short term. Both ends of the yield curve are heavily influenced by the Fed's monetary policy regime - in the short end, the Fed controls (the effective FFR and therefore) short term rates through open market operations and in the long end, the Fed intervenes through 'large scale asset purchases' (quantitative easing).

The Fed consistently intervened to keep short term rates near zero from 2008-2023 and swelled their balance sheet to $9 trillion after buying almost 1-in-3 bonds in existence. We argue that the yield curve is not an indicator for future interest rate expectations but a snapshot on any given day that represents the Fed's monetary policy.

The blip in the 20Y treasury yield presents another problem in using the yield curve as a predictor for interest rates. The 20Y treasury, since its reintroduction, trades at higher yields than its 10Y and 30Y versions due to poor liquidity/demand for that specific tenor. We also argue that the yield curve is influenced by the supply/demand and liquidity dynamics of bonds and may not be able to cleanly capture future rate expectations.

We will use the Ho-Lee interest rate model and will calibrate it to the interest rate futures market (3-mo SOFR). The Ho-Lee model provides us with an easy to calibrate model. Additionally, interest rate futures are highly liquid and are a good proxy for market expectation of interest rates since they are by definition priced as an even risk neutral bet on future interest rates[6]. Therefore, they can act as a good proxy for interest rate expectations.

We chose to model the SOFR rate given its paramount importance in the repo market (an essential mechanism in the financial market with $3.3T in daily outstanding repos on average) and its use in pricing interest rate swaps, which have a total notional value of $100T in 1Q2024.

The paper now shifts to the methodology we employed to implement the model.

2 Methodology

In this section we discuss the methodology we used to achieve the aims of the paper. In particular, we discuss the specifics about the implementation of the Ho-Lee model for our purposes. We go over both the algorithmic methodology and the technical particularities around the implementation (necessitated by and specific to Python 3) Along the way, we make note of all the assumptions necessitated by the scale and scope of the project.

We begin by discussing all the variables needed for a successful implementation of the interest-rate model.

2.1 Initial Variables

The first few variables we need are an initial interest rate r_0, time difference dt, volatility σ, initial drift μ, number of steps T, an array of rates r, and an array of prices A.

Let us briefly mention the precise meanings of dt and T. The former denotes the time difference between two levels of our tree and is measured out of 1 year. For example, $dt = 0.25$ tells us that any two levels in our tree take place exactly $12 \cdot 0.25 = 3$ months apart. In contrast, T is simply the number of steps in the calculation of the interest rates / prices, and hence the number of levels in the tree. In particular, the total time covered by a given tree simulation is calculated as $(T-1) \cdot dt$. Thus, if $T = 3$ and dt is as before, the simulation goes only 6 and not 9 months in to the future.

Our initial aim was to come up with some function to model volatility better than keeping it constant. Due to time constraints, we decided to keep it fixed in the end, and have implemented our algorithm in a way that allows for an immediate modification of the volatility should the reader prefer to use a non-constant function. Initially, we used $\sigma = 0.0025 = 0.25\%$[7]. This is based on the usual change caused by the interference of the Fed (25 bps). The reader is welcome to adjust this to their preference. Additionally, $\mu = [0.0]$. Notice that the drift is an array and not just a single number. This is so because of the fitting we describe later. Essentially, at each step in the tree, we want to use such drift that the difference between our predicted rate/price and the actual one is minimized.

Next, we have r, a list of all the rates we want to use. This is also the data we have collected. More on that in another section. Notice that $r_0 = r[0]$. The reason for it's separation lies in an earlier version of our algorithm that

only simulated a fixed low number of steps. We have decided to keep it separate for time sake and the simplicity it brings to the write up of the actual paper. To get from r to A, our list of prices, we perform a very simple calculation. For each i between 0 and $len(r)$, denoting the length of r, we take $r[i]$ and multiply that by $\frac{100.0}{e^{r[i]\cdot(i+1)}} = 100.0 \cdot e^{-r[i]\cdot(i+1)}$. Hence, our prices in A are in fact the prices of zero-coupon bonds, that follow from the rates we have collected in r.

From here, it is somewhat easy to see that T will eventually be exactly $len(r) + 1 = len(A) + 1$. This is because we want our final tree to have computed $len(r) = len(A)$ levels. Notice, that T will also be equal to $len(\mu)$. In our case, T ended up equal to 25, because we used quarterly data from June 2024 to March 2030[1], inclusive[2].

In particular the data we downloaded and subsequently used from Chicago Mercantile Exchange website was the prices for SOFR futures that we used to obtain quarterly interest rates from June 2024 to March 2030 as described in the previous sentence[2].

Now, we are done with the review of our *initial* variables and we can move to the discussion of the implementation of the actual algorithm.

2.2 The Implementation of the Ho-Lee Model

In this subsection we discuss the specific Python implementation of the Ho-Lee Model we developed. We first describe the main algorithm in general terms. Then we look at each subroutine (method) in detail.

The general structure of our algorithm is quite simple. Our aim is to produce $2(T-1)$ trees. Trees come in pairs. For each different number of time steps, we want to give a tree simulating the possible evolution of the interest rates. Then we want to apply that tree to calculate the prices of zero-coupon bonds given by those very rates. This results in the second tree for a given number of time steps.

Naturally, we begin by looping over a range of values for number of time steps up to $T-1$. For each such value we then have several major jobs. First, we compute *a tree*[3] for rates. Second, we compute another tree for prices of zero-coupon bonds. Then we calculate an error term using that latter tree

[2]The rates are published in March, June, September, and December every year. Thus, we have $3 + 4 \cdot 5 + 1 = 24$ rates in total.

[3]Notice that neither this tree nor the one from the next sentence are actually returned.

and the values from A. Next, we minimize the error term at each step in the calculation. Doing this we obtain a value which we append to μ. We then use the updated list μ to calculate new versions of the first 2 trees we computed. Finally, we return the fitted trees. We then increment the number of time steps[4].

In total, there are 4 helper functions. One for the construction of a rates tree. One for the construction of a prices tree. Another for finding the error term (or gap) between our predicted rates/prices and the ones we get from r/A. A final one for minimizing that gap. We now discuss each of those subroutines in detail. The first three are separated into their own sub-subsections. The last one is a built-in Python method so we discuss it briefly here before proceeding with the rest.

In order to minimize the error term, we use a method called *minimize* from scipy (specifically scipy.optimize in Python 3). The method takes 4 arguments. First, it takes another method to be minimized. Next, it take an argument taken by the other function (the one to be minimized) to return such that the function in question is minimized. Third it takes as a list, any arguments that method to be minimized needs or may take. Finally, as an optional parameter, it takes a method (given as a string) to use when minimizing[4]. Admittedly, for our project's sake this choice did not matter that much given that the 'hardest' of our to be minimized functions are just exponential. Regardless, we used the *L-BFGS-B* method[5].

Notice that *minimize* is returning a value such that it method it takes is literally minimized. Hence, this has the potential to cause problems. Notice, that we called our error term a gap. If we were to take a literal difference between our predicted values and the actual values given by r/A, that would be disastrous. Given that this function is basically a difference, it is probably minimized in a way that results in a value that is negative and not exactly 0 which we want if it is indeed a gap. Hence, whatever function we are minimizing, has to have a theoretical minimum at 0. We explain how we deal with this in the subsection about the last helper function.

Last, we mention that an alternative way to deal with this issue would be to use another built-in method called *fsolve*. As one might guess given

[4]The actual implementation works a bit different, but it does not affect the procedure. In our actual code only the error-finding function is called in the main loop, with the first two subroutines getting called only in the error-finding function. Afterwards, the process continues as described.

[5]It might be important to note that it tends to overestimate rather than underestimate.

the name, it returns a root of the function it takes, instead of a value that minimizes it. We opted not to do this for several reasons. Issues sometimes arrise with *fsolve* because of the need to take the square root of negative numbers or due to the additional check required to see whether we have found a minimum or a maximum[6]. Now that we have discussed *minimize* we proceed to review the other supplementary functions.

2.2.1 Building the Rates Tree

The first helper function we discuss is responsible for the computation of the tree giving the interest rates over the examined time period. We call that function *tree*. It takes 4 self-explanatory arguments: μ, r_0, σ, dt. First, we get ourselves a simple $t = len(\mu)$ for the number of time steps. Next, we make ourselves a $t \times t$ matrix consisting of zeros using *np.zeros* from numpy. We call this matrix *rates*.

As one might expect with 2-dimensional matrices, what follows is a nested loop. First, we loop a variable j from 0 to t. Then we loop another variable i from j[6] to t. If both variables are 0 we are at the very beginning and we just set $rates[t-1][i]$ to our initial rate (r_0). If only j is 0 (notice that the opposite is impossible), we set it to the previous element ($rates[t-1][i-1]$) plus $\mu[i]dt - \sigma\sqrt{dt}$. The last follows from the Ho-Lee model and reveals that we have an arithmetic Brownian motion. Last, if neither variable is 0, we are finished with $rates[t-1]$ and we now set $rates[t-j-1][i]$ to $rates[t-1][i]$ plus $2.0 \cdot j\sigma\sqrt{dt}$. A quick check reveal we have now filled exactly half of the tree (the upper half remains filled with zeros and we do not use it). We conclude by returning *rates*.

We can now move to the function responsible for the computation and production of the prices tree.

2.2.2 Building the Prices Tree

As mentioned above, the second helper function builds the tree responsible for the prices (of zero-coupon bonds such as STRIPS). We call that function *zc*. This method takes two arguments. First, a tree *rt* produced by the *tree* method[7]. Second, it takes *dt* needed for the actual computation.

[6]Those are usually reversed, but we did not notice that until we were late enough into the code that it wasn't worth replacing everything.

[7]Technically it takes any 2-dimensional list.

First, as before, we get our number of time steps from the tree we already have (last time we used the drift list μ). Now, we get $t = rt.shape[0]$ using *shape* in Python. We then create our regular empty $t \times t$ matrix just as before and we call it zct (from zero-coupon tree). Next, we have the exact same nested loop, but the counter variables i, j are no longer swapped.

First, if $i = 0$ we simply set $zct[j][t-1-i]$ to $100.0 \cdot e^{-rt[j][t-1-i]dt}$ Once we are done with those and i is no longer 0, we continue to set the same $zct[j][t-1-i]$ to $0.5(zct[j][t-i] + zct[j-1][t-i]) \cdot e^{-rt[j][t-1-i]dt}$. The first part is just the expected value. We assume that the up and down movements have equal probability of occurring. Finally, we return zct having again filled the lower half of the matrix.

We now move to the last helper function that we have not yet reviewed.

2.2.3 Calculating the Error Term

Our last helper function is called zc_err. It takes four parameters: μ, A and then $param, drift$. You may notice we have both μ and $drift$. The latter is going to be constantly appended at each step in the calculation whereas the former is only used in the beginning. This is purely a stylistic choice. Meanwhile, $param$ is just a collection of variables in a list. It contains r_0, σ, dt. We got that idea from *minimize* as discussed above. We get our t from $len(drift)$ but we must not forget to add 1 for the final step. We then construct rt and zct by calling $tree$ and zc respectfully using the exact arguments described above. Finally, calculate and return the squared difference of each element of zct with each element of A[8]. Notice that in this way, we have something isomorphic, but not equal to the gap and we remove the possibility of *minimize* malfunctioning as describe at the beginning of this subsection. The output we return is a list containing all those squared differences as elements.

Now we plug all that back into our main function and construct our final trees appending μ after each step with a drift that minimizes the square difference. This produces our final 48 trees (24 pairs) which we print tabulated. This conclude our methodology section and allows us to move to the results section.

[8]Alternatively, we could skip computing zct and just compare the elements of rt to those of r provided we have r instead of A as an argument of zc_err

3 Results

In this section we carefully examine all of our results. First, we go over the data we have computed. Then in the Conclusion subsection we draw takeaways.

3.1 Data Results

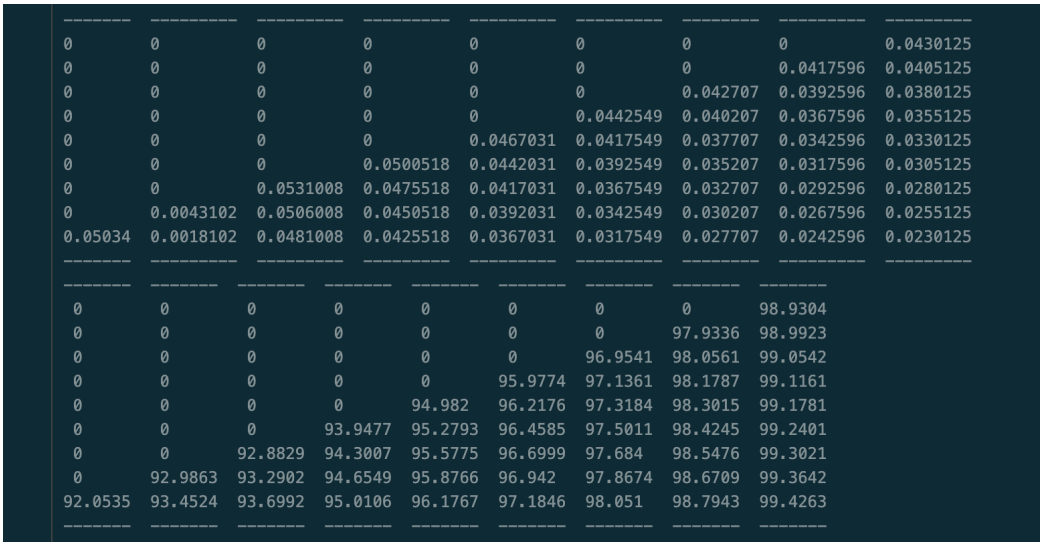
In total, we have built 96 trees. Of them, 48 are interest-rate trees, and 48 are zero-coupon bonds price trees. Of each of those, exactly 24 are fitted by modifying the drift so as to minimize the squared difference between the rates/prices predicted in the first 48 trees and the actual rates/prices given by r/A respectively.

Of the 48 fitted trees, we have 24 pairs of interest rates and zero-coupon bond prices trees. Each pair has a unique number of time steps ranging from 1 to 24. Thus, our first two fitted trees have 2 levels each, and our last two fitted trees have 25 levels each.

Each tree is a $t \times t$ matrix where t is the number of time steps[9] for that tree. The upper-left half is empty (filled with zeros), and the other one is the actual tree. Only the fitted trees are kept. Here is a screenshot for the pair of fitted trees at $t = 10$[10].

[9] Note that t and T are distinct. Whereas T goes to 25, t only goes to 24.

[10] All fitted trees can be studied in the Jupyter Notebook we are attaching to our submission.



0	0	0	0	0	0	0	0	0.0430125
0	0	0	0	0	0	0	0.0417596	0.0405125
0	0	0	0	0	0	0.042707	0.0392596	0.0380125
0	0	0	0	0	0.0442549	0.040207	0.0367596	0.0355125
0	0	0	0	0.0467031	0.0417549	0.037707	0.0342596	0.0330125
0	0	0	0.0500518	0.0442031	0.0392549	0.035207	0.0317596	0.0305125
0	0	0.0531008	0.0475518	0.0417031	0.0367549	0.032707	0.0292596	0.0280125
0	0.0043102	0.0506008	0.0450518	0.0392031	0.0342549	0.030207	0.0267596	0.0255125
0.05034	0.0018102	0.0481008	0.0425518	0.0367031	0.0317549	0.027707	0.0242596	0.0230125

0	0	0	0	0	0	0	0	98.9304
0	0	0	0	0	0	0	97.9336	98.9923
0	0	0	0	0	0	96.9541	98.0561	99.0542
0	0	0	0	0	95.9774	97.1361	98.1787	99.1161
0	0	0	0	94.982	96.2176	97.3184	98.3015	99.1781
0	0	0	93.9477	95.2793	96.4585	97.5011	98.4245	99.2401
0	0	92.8829	94.3007	95.5775	96.6999	97.684	98.5476	99.3021
0	92.9863	93.2902	94.6549	95.8766	96.942	97.8674	98.6709	99.3642
92.0535	93.4524	93.6992	95.0106	96.1767	97.1846	98.051	98.7943	99.4263

Figure 1: Fitted rates/prices tree pair at $t = 10$

In our model, the price of a \$1 zero coupon bond would be $e^{-(r_1 dt_1 + r_2 dt_2 + ... r_n dt_n)}$ where the sum of all dt_i equals the tenor of the zero, and r_i represents the 3-mo SOFR at the start of the period dt_i. Pricing a floating rate payment schedule based on the 3-mo SOFR can be done by considering each payment as a zero, and adding the price of all zeros. If there is a credit spread above SOFR, then the credit spread should be added to each r_i.

Notice that the first level of every zero-coupon price tree is our prediction for the current price of a zero-coupon bond with the respective maturity right now. To see why this is the case, let us recall the way in which that value is obtained. In particular, we start with two lists r and A for the rates and zero-coupon prices right now respectively. So given that the prices in A are computed from r and dt it makes sense to ask why we compute the fitted tree for the zero-coupon bond prices and don't just use the values given by A. This is particularly important given that it is true that we don't need to produce the whole tree to find those values. This is the case because we are aware of the specific path the futures rates are taking[11]. However, using that approach, the initial computation becomes cumbersome because the discounting period between any two rates can vary. We can't rely on a more general approximation, as we do in A (described in Methodology). Instead, we first fit the tree and apply the optimal drifts at each step. Then

[11]This is because we have them in advance.

once we have the whole initial tree, we fit it going backwards to produce the fitted one. At the very end we get to the correct value of the given element of A and we use that as our prediction for the value of the specific zero-coupon bond right now.

Here is a screen shot of our prediction of for the current price of zero-coupon bonds with 24 different maturities at quarterly intervals from June, 2024 to March, 2030.

```
[98.6738716  97.43350896 96.31221808 95.31337871 94.41218904 93.58500672
 92.81610648 92.05352359 91.26982106 90.4837418  89.73134817 88.93183576
 88.15266897 87.3322828  86.52385757 85.69291808 84.86995858 84.01708626
 83.12913034 82.20122347 81.31416121 80.3844047  79.45336025 78.56844738]
```

Figure 2: Predicted Prices for 24 Zero-Coupon Bonds

These values are stored in a list called simply *bonds*. This is an important result on its own because we use exactly those values to determine how successful our model is at pricing zeros. Now that we surveyed all the results we obtained, we are about to discuss exactly that.

3.2 Conclusion

To determine the success of our model, we compared our predicted prices for the different maturity zero-coupon bonds right now to the actual prices of STRIPS with the closest maturities. Unfortunately, it wasn't possible to do a comparison of the exact same maturities because our zero-coupon bonds expire at the end of every third month and the real life actual STRIPS expire two and a half months later[3]. Despite that, our results signal the model is performing quite well. Here is a picture detailing the actual prices and our prediction for the price of a zero-coupon bond with the respective corresponding maturity.

Market Prices of Treasury Strips		Model Predicted Prices	
Maturity Date	Price of a 100 Zero	Maturity Date	Price of a 100 Zero
8/15/24	98.883599	6/30/24	98.6738716
11/15/24	97.615901	9/30/24	97.43350896
2/15/25	96.473635	12/30/24	96.31221808
5/15/25	95.31534	3/30/25	95.31337871
8/15/25	94.183485	6/30/25	94.41218904
11/15/25	93.077935	9/30/25	93.58500672
2/15/26	91.933088	12/30/25	92.81610648
5/15/26	90.99196	3/30/26	92.05352359
8/15/26	90.05698	6/30/26	91.26982106
11/15/26	89.159765	9/30/26	90.4837418
2/15/27	88.22141	12/30/26	89.73134817
5/15/27	87.24726	3/30/27	88.93183576
8/15/27	86.269775	6/30/27	88.15266897
11/15/27	85.334977	9/30/27	87.3322828

Figure 3: Actual STRIPS Prices vs Predicted Zero-Coupon Bonds Prices

As one can see our prediction is within \$1 of the actual strip price for exactly half of all computed pairs.

Last, we present some comparisons between the actual and predicted prices with the respective days to maturity of each pair of strip and zero-coupon bond. Again, our predictions are quite close. The results are presented in the following two graphics.

Days to Maturity	Market Price of a \$100 Zero	Days to Maturity	Model Price of a \$100 Zero
80.00	\$98.88	34.00	\$98.67
172.00	\$97.62	126.00	\$97.43
264.00	\$96.47	217.00	\$96.31
353.00	\$95.32	307.00	\$95.31
445.00	\$94.18	399.00	\$94.41
537.00	\$93.08	491.00	\$93.59
629.00	\$91.93	582.00	\$92.82
718.00	\$90.99	672.00	\$92.05
810.00	\$90.06	764.00	\$91.27
902.00	\$89.16	856.00	\$90.48
994.00	\$88.22	947.00	\$89.73
1083.00	\$87.25	1037.00	\$88.93
1175.00	\$86.27	1129.00	\$88.15
1267.00	\$85.33	1221.00	\$87.33

Figure 4: Actual STRIPS Prices vs Predicted Zero-Coupon Bonds Prices

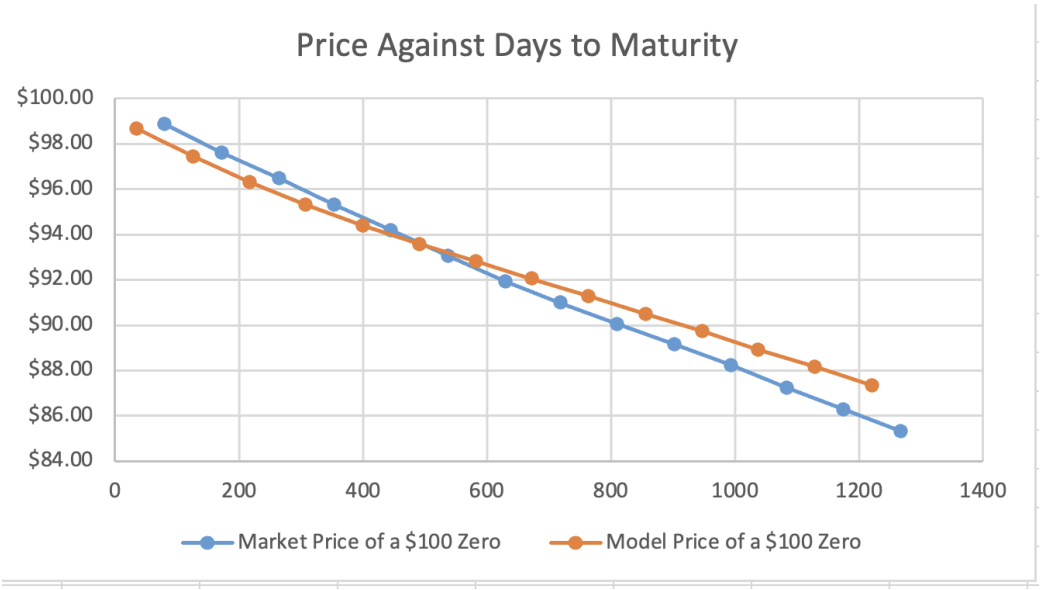


Figure 5: Actual STRIPS Prices vs Predicted Zero-Coupon Bonds Prices

Given this, we believe that our model, given some further changes suggested below, could be suited to pricing floating rate payment obligations based on the SOFR and be used as a short term interest rate forecasting tool.

3.2.1 Further Possible Changes

Many interest rate models address the existence of credit cycles by incorporating mean reversion. While our modified Ho-Lee model is a short term forecast (and the average length of a credit cycle is 6 years), adding in a mean reverting term which equals the user's opinion on $r*$ (the neutral interest rate) can allow for more sophistication in the model. Since $r*$ is not known and can only be estimated, we do not explicitly add it to our model.

Our modified Ho-Lee model assumes constant volatility. SOFR volatility is subject to change temporarily depending on certain events that can affect liquidity in the overnight collateralized lending market. These could be debt ceiling crises, financial crises, or cash shortages (as happened in 2019). A more thorough consideration of volatility could try to pick up on changes in liquidity and consider a jump diffusion stochastic process with mean reversion to account for events that can cause a spike in overnight rates.

4 References

[1] Bloomberg. UST STRIPS, 2024.

[2] CME Group. Three-month sofr. https://www.cmegroup.com/markets/interest-rates/stirs/three-month-sofr.contractSpecs.html, 2024.

[3] ISDA. Swapsinfo quarterly review. https://www.isda.org/2024/05/13/swapsinfo-first-quarter-of-2024-review/, 2024.

[4] Alexander Levin. Interest rate model selection. https://www.ad-co.com/system/files?file=adco-articles/AlexLevin%20JPM.pdf, 2004.

[5] SIFMA. Us repo statistics. https://www.sifma.org/resources/research/us-repo-statistics/, 2024.

[6] Sang Lee Thomas Ho. Term structure movements and pricing interest rate contingent claims. https://www.jstor.org/stable/2328161, 1986.

[7] Paul Wilmott. Paul Wilmott Introduces Quantitative Finance.