Optimizing Coupon Rate for a Step-up Autocallable Note Using Monte Carlo Simulation

Group 2

Chun Hin CHEUNG

Hok Sum Clarence HSU

Teng Koon Dexter WOO

Hui Kei AU YEUNG

Introduction

This work explores how to back-solve the coupon rate of a step-up autocallable note for a given target note price using Monte Carlo simulation. The process begins with the calculation of spot rates for interest and dividend yields using cubic spline interpolation on Bloomberg Terminal data. Next, forward rates are computed to simulate underlying index paths in the Monte Carlo framework.

We also gathered and fitted implied volatility data, which allows us to create a local volatility surface crucial for the Monte Carlo simulation. Dupire's local volatility model is used to transition from implied to local volatilities.

To identify the optimal coupon rate, we use the bisection method, an iterative algorithm that converges on the coupon rate matching the simulated note price with the target price.

Note Structure

The note under analysis is a step-up autocallable financial instrument. Key details about its structure are as follows:

- Trade Date and Maturity: The trade date is 17/11/2023. The maturity date is three years after the trade date.
- Issue Price and Note Denomination: The issue price is equal to 100% of the note denomination. Each note has a denomination of USD 10,000.

The note is based on three underlying indices:

1. HSCEI (Spot = 5974.47)
2. Kospi 200 (Spot = 331.63)
3. S&P 500 (Spot = 4513.91)

Coupon Payments and Knock-out Event:

- The note pays a "minimum coupon" of 0.01% per annum every six months unless a knock-out event occurs.
- A knock-out event is triggered if the closing price of the laggard index (the index with the lowest relative value) is equal to or greater than the initial spot price.
- In case of a knock-out event, the note is redeemed and expires. The redemption amount equals the note denomination multiplied by (100% + the number of observation dates times the coupon).

Final Redemption Conditions:

- If no knock-out event occurs before maturity, the final redemption depends on the occurrence of a knock-in event.
- A knock-in event is defined as the laggard index's closing price falling to 50% or less of the initial spot price.
- If a knock-in event has not occurred, each note is redeemed at the denomination value.
- If a knock-in event has occurred, each note is redeemed at the lesser of 100% or the ratio of the laggard index's closing price to its initial spot price, multiplied by the note denomination.

The primary task is to find the coupon rate that makes the price of the note close to 98% of the issue price.

Interest Rate and Dividend Yield

We first compute the spot rate of interest rate r_t and dividend yield d_t, where r_t and d_t means t-days annualized interest rate and t-days annualized dividend yield from the trading day. r_t will be used for discounting payoffs.

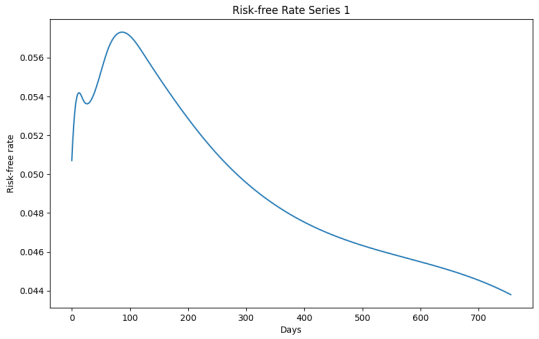
Interest rate r_t is interpolated with cubic spline using data stated below.

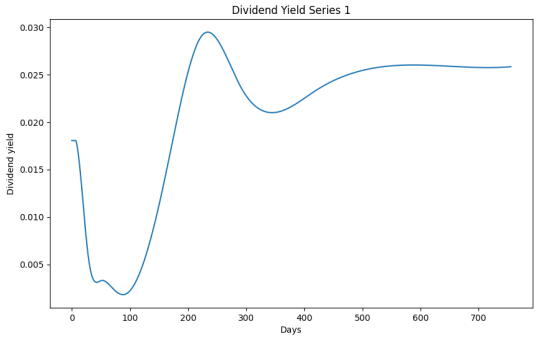
	Bloomberg Terminal Function	Data Used for Interpolation
KOSPI 200	BTMM	KORIBOR, KOFR (1D-3Y)
S&P 500	BTMM	OIS, T-Bills (1D-3Y)
HSCEI	BTMM	HIBOR, OIS (1D-3Y)

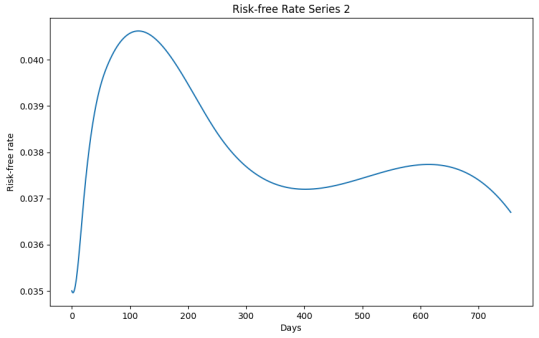
Dividend yield d_t is interpolated with cubic spline using dividend yield data from OVDV function in the Bloomberg Terminal.

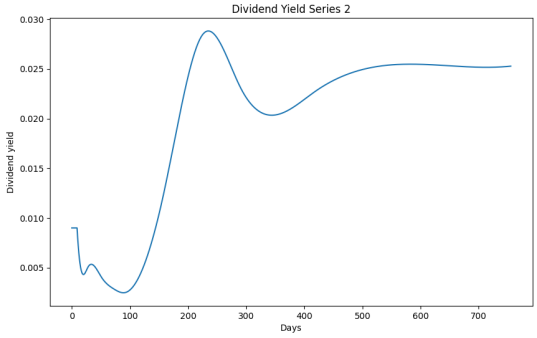
The function we used for interpolation is interp1d(a, b, kind='cubic', fill_value=(first_value, last_value), bounds_error=False)

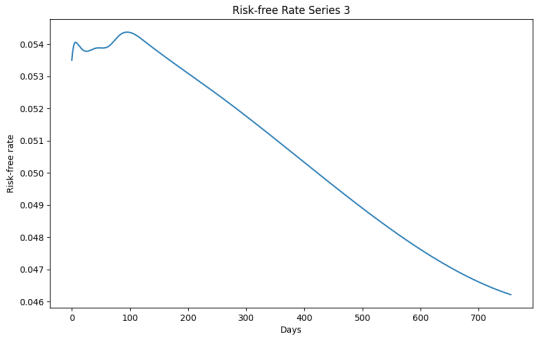
The graphs below show the interpolated curve (1 = HSCEI, 2 = KOSPI 200, 3 = S&P 500):

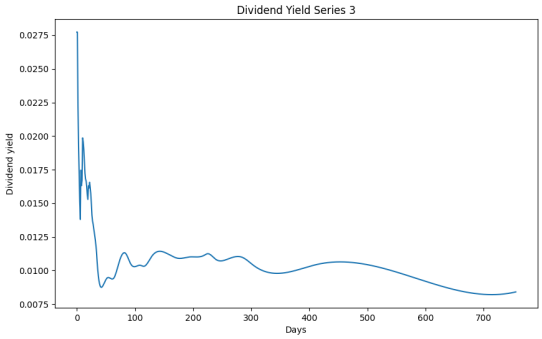










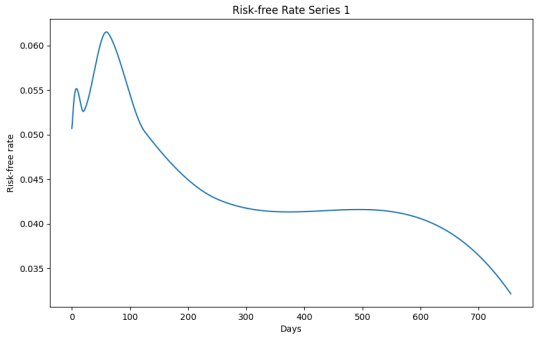


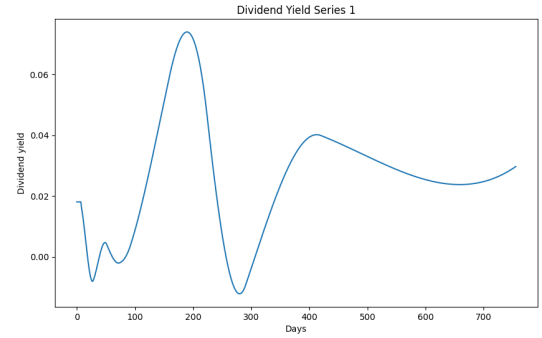
With the above curves, we can obtain the implied forward risk-free rate for each day. These forward rates will be used in each step of the Monte Carlo simulation process:

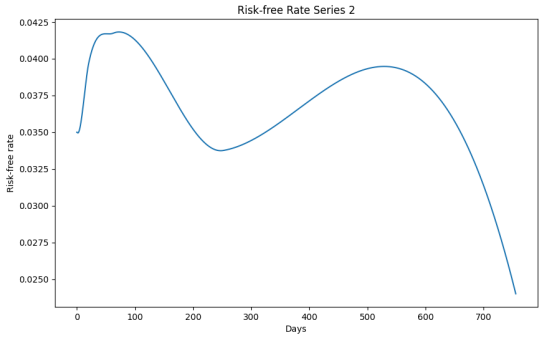
$$r^f_{t+1} = \frac{(1 + r_{t+1}\mathrm{dt})^{t+1}}{(1 + r_t\mathrm{dt})^t} - 1$$

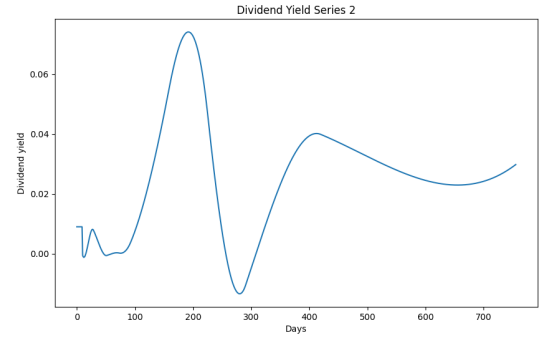
$$d^f_{t+1} = \frac{(1 + d_{t+1}\mathrm{dt})^{t+1}}{(1 + d_t\mathrm{dt})^t} - 1$$

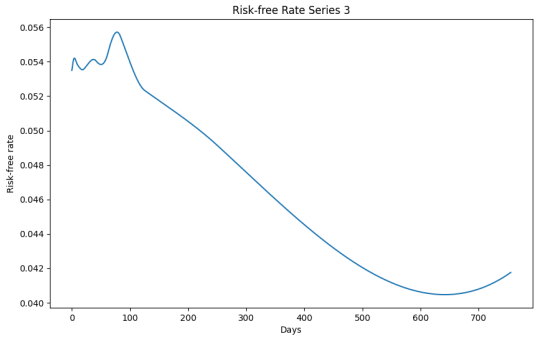
$\mathrm{dt} = 1/252$. Here, we make the assumption that the yield curve pure expectation theory holds true.

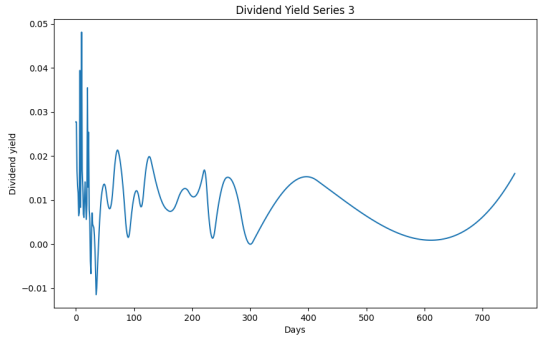












Implied Volatility Data

From OMON, we first get the implied volatility data trading on the market for the three indices concerned. Then, we consolidated the data into a table containing the implied volatilities at each strike level and exercise date. We take the average of the volatilities of the call and the put if both are available. The results are saved in an Excel file.

Implied Volatility Curve Fitting

The next step in our process is to fit a function to the implied volatilities that we collected. This is done for each expiry date across the range of strike prices. The goal here is to create a smooth, continuous curve that represents the implied volatility surface. The function is then used for calculating local volatilities.

In our code, we define a function that models the implied volatility as a function of the log-moneyness of the option, represented as x. The function includes parameters for ATM volatility (σ_{atm}), skew (δ), and convexity (γ), and celerity (κ) to control the steepness of the tails.

$$\sigma_{\text{BS}}^2(x) = \sigma_{\text{atm}}^2 + \delta\left(\frac{\tanh \kappa x}{\kappa}\right) + \frac{\gamma}{2}\left(\frac{\tanh \kappa x}{\kappa}\right)^2$$

We fit this function to the squared implied volatilities from our data, using the method of least squares to minimize the sum of the squared residuals between the function's predictions and the actual data.

We use the minimize function from the scipy.optimize module, with the SLSQP method, to find the parameters that minimize the residuals. We note that $\sigma_{\text{BS}}^2(x)$ cannot be negative. Therefore, we derive restrictions for the parameters as follows:

$$z = \frac{\tanh \kappa x}{\kappa}$$

$$\sigma_{\text{BS}}^2(z) = \sigma_{\text{atm}}^2 + \delta z + \frac{\gamma}{2} z^2$$

Note that $\sigma_{\text{BS}}^2(z)$ is a quadratic function in z. Hence, given that $\sigma_{\text{BS}}^2(z) > 0 \ \forall x \in \mathbb{R}$,

$$\Delta = \delta^2 - 4\left(\frac{\gamma}{2}\right)\sigma_{\text{atm}}^2 < 0 \text{ and} \frac{\gamma}{2} > 0$$

$$\delta^2 - 2\gamma\sigma_{\text{atm}}^2 < 0 \text{ and } \gamma > 0$$

Additionally, we add a constraint for σ_{atm}:

$$0 < \sigma_{\text{atm}} < 0.3$$

The result is that for each expiry date, we obtain a set of optimal parameters for our implied volatility function, which we can then use to estimate implied volatilities for any strike price.

Transforming Implied Volatility to Local Volatility

The local volatility model, proposed by Bruno Dupire, is a technique to convert market-quoted implied volatilities into local volatilities. The relationship is shown below:

$$\sigma_L^2 = \frac{\frac{\partial w}{\partial T}}{1 - \frac{y}{w}\frac{\partial w}{\partial y} + \frac{1}{4}\left(-\frac{1}{4} - \frac{1}{w} + \frac{y^2}{w^2}\right)\left(\frac{\partial w}{\partial y}\right)^2 + \frac{1}{2}\frac{\partial^2 w}{\partial y^2}}$$

$$w = \sigma_{\text{BS}}^2 T$$

$$y = \ln \frac{K}{F_T}$$

The log-moneyness x and log-forward-moneyness y is equated as follows:

$$y = x - \int_0^T (r_t - d_t)\ \text{dt}$$

This implies that:

$$\frac{dx}{dy} = 1,\ \frac{d^2x}{dy^2} = 0$$

Hence, we can write:

$$\frac{\partial w}{\partial y} = \frac{\partial w}{\partial x}\frac{dx}{dy} = \frac{\partial w}{\partial x}$$

$$\frac{\partial^2 w}{\partial y^2} = \frac{\partial^2 w}{\partial x^2}\left(\frac{dx}{dy}\right)^2 + \frac{\partial w}{\partial x}\frac{d^2x}{dy^2} = \frac{\partial^2 w}{\partial x^2}$$

Given that we have expressed w as a function of x,

$$w(x) = \left[\sigma_{\text{atm}}^2 + \delta\left(\frac{\tanh \kappa x}{\kappa}\right) + \frac{\gamma}{2}\left(\frac{\tanh \kappa x}{\kappa}\right)^2\right] T,$$

We can compute σ_L^2 as:

$$\sigma_L^2 = \frac{\frac{\partial w}{\partial T}}{1 - \frac{y}{w}\frac{\partial w}{\partial x} + \frac{1}{4}\left(-\frac{1}{4} - \frac{1}{w} + \frac{y^2}{w^2}\right)\left(\frac{\partial w}{\partial x}\right)^2 + \frac{1}{2}\frac{\partial^2 w}{\partial x^2}}$$

In our process, we utilize the implied volatility curve we fitted in the previous step. By inputting log-moneyness and the date (which must be one of the existing exercise dates), we output the local volatility.

Interpolating Local Volatility

The method calculates the local volatility value at a specific day for a particular underlying. It accomplishes this by interpolating across a series of local volatilities calculated at specific exercise dates and then evaluating the interpolation at the desired day. The interpolation method utilized is cubic spline, which is a type of piecewise polynomial interpolation. This method is particularly suitable for this application as it provides a smooth curve that passes exactly through the given data points, and its second derivative is continuous, making it differentiable everywhere.

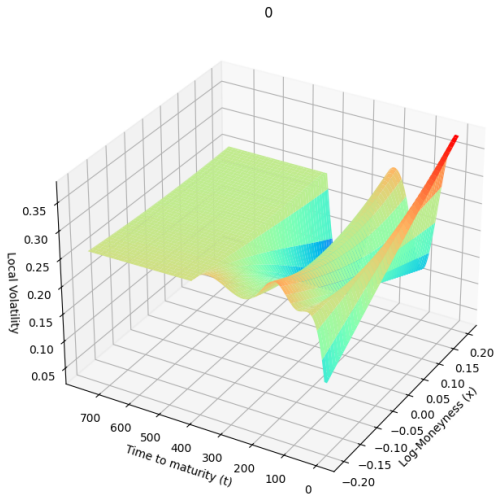
The process begins by iterating over each exercise date for the given underlying. For each of these dates, the method computes the local volatility at the exercise date.

The method also takes note of the first and last local volatility values, to use them later as fill values for the interpolation outside the range of exercise days.

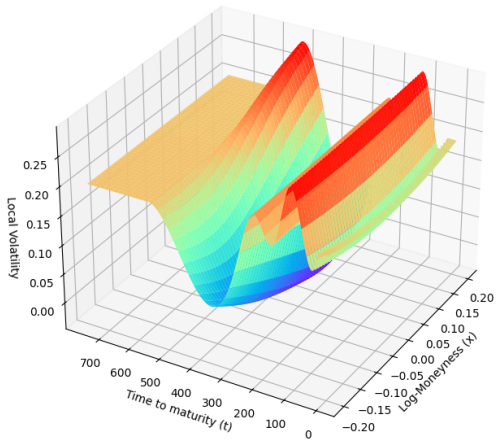
Once it has calculated local volatilities for all the exercise dates, the method constructs the cubic spline interpolation function.

Finally, the method evaluates the interpolation function at the specified day. This results in the interpolated local volatility, which is then returned as the output of the method.

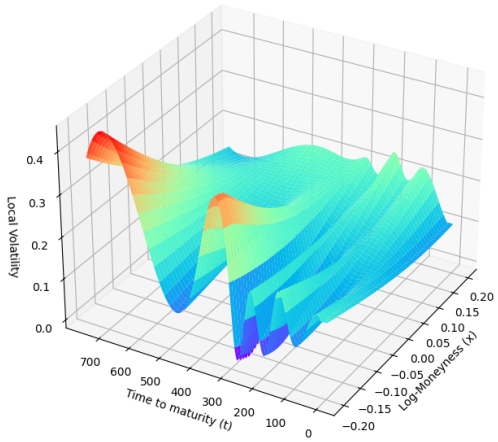
The volatility surfaces are shown below (0 = HSCEI, 1 = KOSPI 200, 2 = S&P 500):



1



2



Monte Carlo Simulation

The method first generates correlated standard normal random variables Z_1, Z_2, and Z_3 using the Cholesky decomposition of the correlation matrix and random normal variables W_1, W_2, and W_3.

From the CORR function in Terminal, the correlation matrix of HSCEI, KOSPI 200, and S&P 500 is:

$$C = \begin{bmatrix} 1 & 0.551 & 0.111 \\ 0.551 & 1 & 0.191 \\ 0.111 & 0.191 & 1 \end{bmatrix}$$

Then, we use np.linalg.cholesky() function to perform Cholesky decomposition to find the lower triangular matrix L where $C = LL^T$.

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0.551 & 0.83450524 & 0 \\ 0.111 & 0.155588 & 0.98156578 \end{bmatrix}$$

We then build 3 correlated processes Z_1, Z_2, Z_3 from 3 independent processes W_1, W_2, W_3 where:

$$\begin{bmatrix} Z_1 \\ Z_2 \\ Z_3 \end{bmatrix} = L \begin{bmatrix} W_1 \\ W_2 \\ W_3 \end{bmatrix}$$

It then concatenates these variables with their negations to form Z_1, Z_2, and Z_3, resulting in the final normal random variables used in the Monte Carlo simulation. That is, we use antithetic variates $\begin{bmatrix} -Z_1 \\ -Z_2 \\ -Z_3 \end{bmatrix}$ for variance reduction in simulation.

Then, we simulate the price path S_t of each index (HSCEI, KOSPI 200, and S&P 500) using a Geometric Brownian Motion, where:

$$S_{t+1} = S_t e^{\left(r_t - d_t - \frac{\sigma_t^2}{2}\right) dt + \sigma_t Z_t \sqrt{\mathrm{dt}}}, S_0 = 1$$

r_t and d_t are the implied forward rates and yields, and σ_t is the local volatility at that time step. dt is set to be $\frac{1}{252}$.

Observations of the price of each index are then made at specified durations (obs_duration = [0.5, 1, 1.5, 2, 2.5, 3]). The worst-performing index (the laggard) is determined at each observation point, and a knock-in event is flagged if the laggard's price falls below 0.5.

The payoff for each simulation is calculated based on the coupon payments and redemption amount of the note. If a knock-out event occurs (i.e., the laggard index's price is above 1), the note is redeemed early and the payoff is the note's denomination plus accrued coupons. If no knock-out event occurs before maturity, the final redemption amount depends on whether a knock-in event has occurred. If a knock-in event has occurred, the final redemption amount is the note's denomination times the laggard index's moneyness (capped at 1). If no knock-in event has occurred, the final redemption amount is the note's denomination.

Finally, the average payoff across all simulations is calculated and returned as the simulated price of the note. In total, we run 1000 simulations to estimate the note price.

Bisection Method to Solve for the Coupon

The method first defines a helper function error(coupon) that calculates the difference between the simulated price of the note (using the price(coupon) method of the MonteCarlo class) and the target price.

Next, it initializes two coupon rates, coupon_l and coupon_r, to 0 and 0.01, respectively. It then increments coupon_r by 0.01 until the simulated price at coupon_r is greater than the target price.

Once coupon_l and coupon_r are set, the method implements the bisection algorithm to find the root of the error(coupon) function, i.e., the coupon rate that makes the simulated price equal to the target price. The algorithm works by repeatedly bisecting the interval [coupon_l, coupon_r] and selecting the subinterval where the error(coupon) function changes sign until the absolute error at the midpoint of the interval is less than the specified tolerance. We set the tolerance to $1.

The method prints the error and the midpoint of the interval at each iteration for debugging purposes.

Finally, the method returns the coupon rate at the midpoint of the final interval, which is the estimated root of the error(coupon) function.

We use bisection method and find out that a note with 98% of issue price has coupon of 13.37%.

Code Workflow

We first define three basic configurations: T = 3, dt = 1 / 252, n_sim = 1000.

We then import the class Div_RF and get the risk-free rate and dividend yield at each step (day) of the Monte Carlo simulation process as follows:

div_rf = Div_Rf(T, dt)

r = div_rf.get_all_rf()

d = div_rf.get_all_dividend()

r_forward = [div_rf.get_forward_rates(index, dt) for index in r]

d_forward = [div_rf.get_forward_rates(index, dt) for index in d]

Afterwards, we import the class MonteCarlo and run the bisection method.

mc = MonteCarlo(r, r_forward, d_forward, T, dt, n_sim)

print(mc.bisection(9800))

Conclusion

In our study, we successfully used Monte Carlo simulation to determine the coupon rate of a step-up autocallable note, targeting a note price of 98% of the issue price. This involved interpolating yields, constructing a local volatility surface, and using a bisection method to find the optimal coupon rate. The determined coupon rate is 13.37%.

Appendix: Python Classes

Div_Rf Class

The Div_Rf class is used to compute the risk-free rate and dividend yield at every step of the Monte Carlo simulation.

Class Initialization

The class is initialized with the following arguments:

T: The total time to maturity.

dt: The time step.

Upon initialization, the class reads in risk-free rate data from an Excel file and saves this data for later use.

Instance Methods

The Div_Rf class contains the following instance methods:

trading_days_in_between(self, y) This method returns the number of trading days between the trading date and a given date y.

CubicSplineInterpolationDYRFR(self, time_step, a, b) This method performs cubic spline interpolation. It takes a time_step and two arrays a and b as input. This method is used internally to interpolate dividend yields and risk-free rates.

get_div_divdate(self, index) This method reads in dividend data for a given index from an Excel file, and calculates the number of trading days from a fixed date to the date of each dividend.

get_dividend(self, index) This method returns interpolated dividend yields for a given index at each time step.

get_all_dividend(self) This method returns a list of interpolated dividend yields for three indices: "KOSPID", "SPXD", and "HSCEID".

get_r_rdate(self, country) This method reads in risk-free rate data for a given country from an Excel file, and calculates the number of trading days from a fixed date to the date of each risk-free rate.

get_rf(self, country) This method returns interpolated risk-free rates for a given country at each time step.

get_all_rf(self) This method returns a list of interpolated risk-free rates for three countries: "KR", "US", and "HK".

get_forward_rates(self, r, dt) This method converts a list of annualized rates into forward implied rates.

Vol_Calculation Class

The Vol_Calculation class is a class designed for calculating local volatilities for each step of the Monte Carlo simulation.

Class Initialization

The class is initialized with the following arguments:

spot_prices: An array of spot prices of the underlying asset.

T: The total time to maturity.

dt: The time step.

today_date: The trade date.

Upon initialization, the class also creates an instance of the Div_Rf class and retrieves risk free rates and dividends.

Instance Methods

The Vol_Calculation class contains the following instance methods:

load_data(self, excel_file, indices_names) This method loads implied volatilities from the Excel file and stores them in the class instance. This is the first instance to be called when using the Vol_Calculation class.

excel_to_implied_vol_dfs(self, excel_file, indices_names) This method reads implied volatilities from an Excel file. The Excel file and the names of the indices are given as inputs. The method returns a list of data frames, one for each index.

implied_vol_curve_func(self, x, sigma_atm, delta, gamma, kappa) This method defines the functional form of the implied volatility curve as previously mentioned. It takes as input the log-moneyness x and the parameters sigma_atm, delta, gamma, and kappa.

implied_vol_curve_fitting(self, implied_vol_dfs) This method performs curve fitting for the implied volatility curve. It takes as input a data frame of implied volatilities and returns a data frame of fitted parameters for each column in the input data frame.

daysinbetween3(self, y) This method returns the number of trading days between the trading date and a given date y. This is identical to trading_days_in_between(self, y) in the class Div_RF.

implied_to_local_vol(self, x_0, index_no, params_dfs, exercise_date) This method converts implied volatilities to local volatilities. It takes as input the log-moneyness x_0, the index number index_no, a data frame of fitted parameters params_dfs, and the exercise date exercise_date. The method returns the local volatility.

interpolated_local_vol(self, x_0, index_no, day, params_dfs) This method interpolates local volatilities. It takes as input the log-moneyness x_0, the index number index_no, the day day, and a data frame of fitted parameters params_dfs. The method returns the interpolated local volatility.

MonteCarlo Class

The MonteCarlo class is a class designed to simulate the prices of the three underlying equity indices using the Monte Carlo simulation method.

Class Initialization

The class MonteCarlo is initialized with the following parameters:

r: A numpy array representing the risk-free interest rates.

d: A numpy array representing the dividends.

T: The total time period for the simulation.

dt: The time step for the simulation.

n_sim: The number of simulations to run.

Upon initialization, the class also calculates the number of steps (n_step), the observation duration (obs_duration), and sets the initial values for the correlation matrix, note denomination, and minimum coupon. The correlation matrix is a pre-determined matrix that represents the correlation between different assets. The note denomination and minimum coupon are attributes of the financial note being modeled.

Additionally, the class initializes the Vol_Calculation object, vol_calc, and the implied volatility parameters data frames (params_dfs). These are used throughout the class to calculate the local volatility and to fit the implied volatility curve.

The class also precomputes and stores the sigma (volatility) values for the range of log price levels of the underlying assets, which are used in the Monte Carlo simulation.

Class Methods

The MonteCarlo class includes the following methods:

precompute_sigma(self) This method precomputes and returns sigma (volatility) values for a range of log price levels of the underlying assets. If a sigma.npy file exists, it attempts to load the sigma values from the file. Otherwise, it computes the sigma values and saves them to the file.

price(self, coupon) This method simulates the price of the financial note over time using the Monte Carlo method. It takes the coupon rate as an input, simulates the price path of three indices (HSCEI, KOSPI 200, and S&P 500), calculates the payoff for each simulation, and then returns the average payoff across all simulations as the simulated price of the note.

The simulation includes modeling the price path of the indices using geometric Brownian motion, with the local volatility model used for the sigma in the GBM. It also includes simulating knock-in and knock-out events, which affect the payoff from the financial note.

bisection(self, price, tol=1) This method uses the bisection method to find the coupon rate that would result in a given price for the financial note. It takes as input the desired price and an optional tolerance parameter, which defaults to 1.

The method iteratively adjusts the coupon rate until the error between the simulated price (using the price(coupon) method) and the input price is within the specified tolerance. It then returns the coupon rate that results in the desired price.

get_sigma(self, price, day, index_no) This method retrieves the precomputed sigma value for a given price level, day, and index number. The price level is used to find the corresponding log price level in the precomputed sigma values array. The method then returns the sigma value at this log price level, day, and index number.