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

Angus Cheung

January 2025

Contents

1 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.

2 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.
- **Underlying Indices:** 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)

2.1 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:

$$100\% + (\text{number of observation dates}) \times (\text{coupon}).$$

2.2 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.

3 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 represent t-days annualized interest rate and t-days annualized dividend yield from the trading day. The spot rate r_t will be used for discounting payoffs.

3.1 Interest Rate Interpolation

The interest rate r_t is interpolated using cubic spline interpolation with the data provided below:

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

Table 1: Interest rate data for interpolation

3.2 Dividend Yield Interpolation

The dividend yield d_t is interpolated using cubic spline interpolation with dividend yield data from the OVDV function on the Bloomberg Terminal. The interpolation function used is:

```
interp1d(a, b, kind='cubic', fill_value=(first_value, last_value), bounds_error=False)
```

3.3 Interpolated Curves

The graphs below show the interpolated curves for the three indices: 1 = HSCEI, 2 = KOSPI 200, 3 = S&P 500.

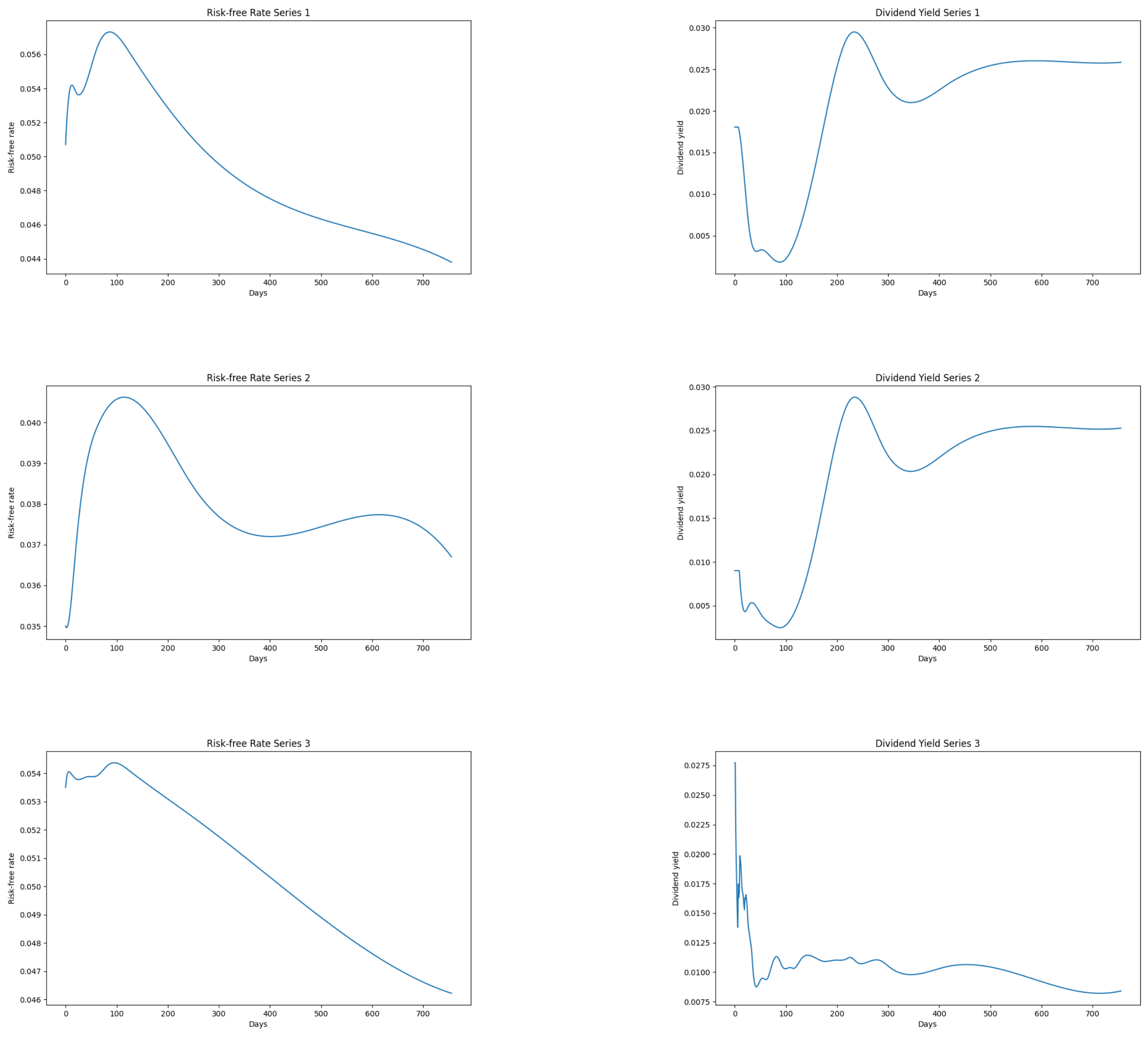

Figure 1: Interpolated curves for HSCEI, KOSPI 200, and S&P 500.

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:

$$r_f^{t+1} = \frac{(1 + r_{t+1} \cdot dt)^{t+1}}{(1 + r_t \cdot dt)^t} - 1$$

$$d_f^{t+1} = \frac{(1 + d_{t+1} \cdot dt)^{t+1}}{(1 + d_t \cdot dt)^t} - 1$$

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

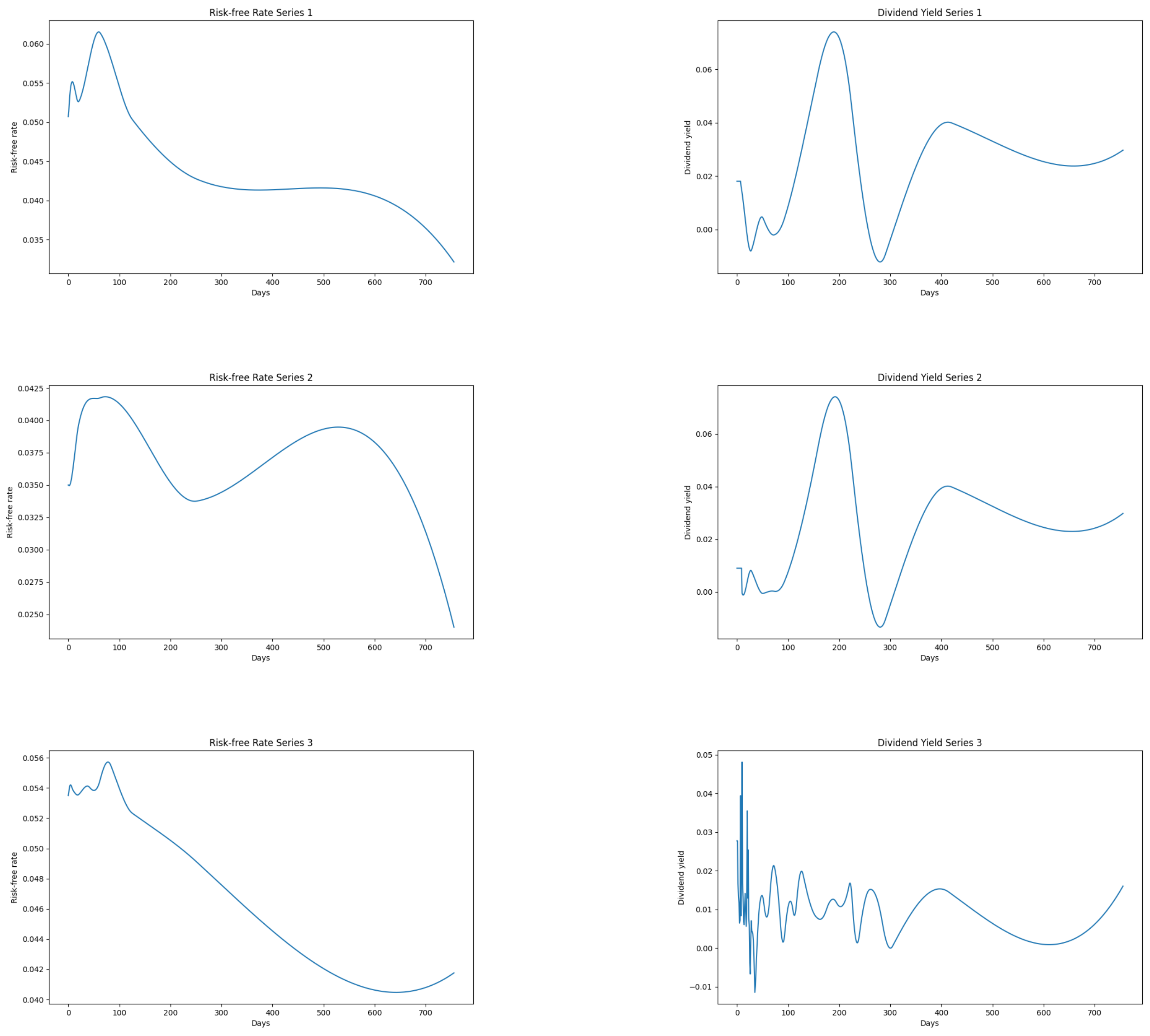

Figure 2: Interpolated curves for HSCEI, KOSPI 200, and S&P 500.

4 Implied Volatility Data

From the Bloomberg OMON function, we first obtain the implied volatility data for the three indices concerned. The collected data includes implied volatilities at various strike levels and exercise dates. If both call and put volatilities are available, we take their average. The consolidated data is then saved in an Excel file for further processing.

5 Implied Volatility Curve Fitting

The next step 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 is to create a smooth, continuous curve that represents the implied volatility surface. This fitted function will then be used to calculate local volatilities.

5.1 The Implied Volatility Function

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 the following parameters:

- σ_{atm}: At-the-money (ATM) volatility,
- δ: Skew, which controls the slope of the curve,
- γ: Convexity, which controls the curvature of the curve,
- κ: Celerity, which controls the steepness of the tails.

The function is expressed as:

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

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 $\sigma_{\text{BS}}^2(z) > 0 \ \forall x \in \mathbb{R}$,

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

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

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.

6 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:

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

where:

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

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

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

$$y = x - \int_0^T (r_t - d_t)\, dt$$

This implies that:

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

Hence, we can write:

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

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

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

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

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.

7 Interpolating Local Volatility

The method calculates the local volatility value at a specific day for a particular underlying. This is achieved 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, a type of piecewise polynomial interpolation. This method is particularly suitable as it:
- Provides a smooth curve that passes exactly through the given data points, - Ensures the second derivative is continuous, making it differentiable everywhere.

7.1 Process Overview

1. Iterating over exercise dates: The process begins by iterating over each exercise date for the given underlying. For each exercise date, the method computes the local volatility at that date.
2. Handling boundary values: The method stores the first and last local volatility values to use as fill values for interpolation outside the range of exercise days.
3. Constructing the cubic spline: Once local volatilities are calculated for all exercise dates, the method constructs a cubic spline interpolation function.
4. Evaluating the interpolation: The interpolation function is evaluated at the specified day, resulting in the interpolated local volatility.

7.2 Output

The interpolated local volatility is returned as the output of the method. The volatility surfaces for the three indices are shown below:

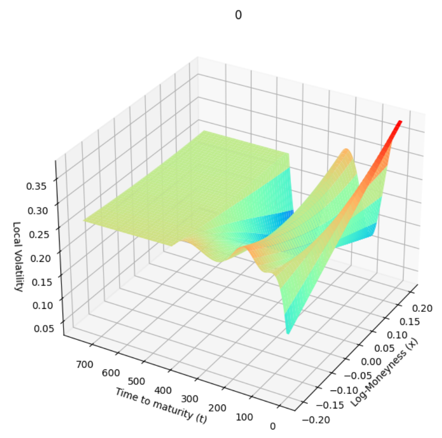

Figure 3: Local Volatility Surface for HSCEI

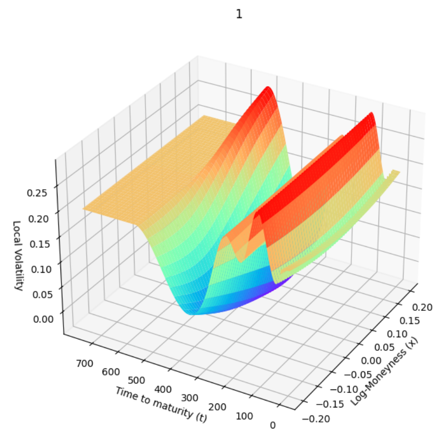

Figure 4: Local Volatility Surface for KOSPI 200

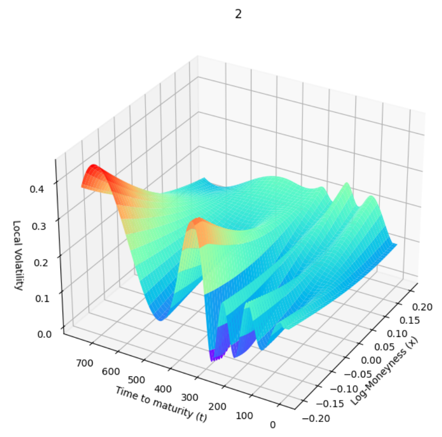

Figure 5: Local Volatility Surface for S&P 500

8 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:

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

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.

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

10 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))
```

11 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%.

12 Appendix: Python Classes

12.1 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.

12.1.1 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.

12.1.2 Instance Methods

The `Div_Rf` class contains the following instance methods:

- `trading_days_in_between(self, y)`: Returns the number of trading days between the trading date and a given date `y`.
- `CubicSplineInterpolationDYRFR(self, time_step, a, b)`: 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)`: Reads 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)`: Returns interpolated dividend yields for a given index at each time step.
- `get_all_dividend(self)`: Returns a list of interpolated dividend yields for three indices: "KOSPID", "SPXD", and "HSCEID."
- `get_r_rdate(self, country)`: Reads 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)`: Returns interpolated risk-free rates for a given country at each time step.
- `get_all_rf(self)`: Returns a list of interpolated risk-free rates for three countries: "KR", "US", and "HK."
- `get_forward_rates(self, r, dt)`: Converts a list of annualized rates into forward implied rates.

12.2 Vol_Calculation Class

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

12.2.1 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.

12.2.2 Instance Methods

The `Vol_Calculation` class contains the following instance methods:

- `load_data(self, excel_file, indices_names)`: Loads implied volatilities from the Excel file and stores them in the class instance.
- `excel_to_implied_vol_dfs(self, excel_file, indices_names)`: Reads implied volatilities from an Excel file. The method returns a list of data frames, one for each index.
- `implied_vol_curve_func(self, x, sigma_atm, delta, gamma, kappa)`: Defines the functional form of the implied volatility curve. It takes log-moneyness `x` and the parameters `sigma_atm`, `delta`, `gamma`, and `kappa`.
- `implied_vol_curve_fitting(self, implied_vol_dfs)`: Performs curve fitting for the implied volatility curve. It takes a data frame of implied volatilities and returns a data frame of fitted parameters for each column.
- `daysinbetween3(self, y)`: Returns the number of trading days between the trading date and a given date `y`.

- `implied_to_local_vol(self, x_0, index_no, params_dfs, exercise_date)`: Converts implied volatilities to local volatilities. It takes log-moneyness `x_0`, the `index_no`, the fitted parameters `params_dfs`, and the `exercise_date`.

- `interpolated_local_vol(self, x_0, index_no, day, params_dfs)`: Interpolates local volatilities. It takes log-moneyness `x_0`, the `index_no`, the `day`, and the fitted parameters `params_dfs`.

12.3 MonteCarlo Class

The `MonteCarlo` class simulates the prices of the three underlying equity indices using the Monte Carlo simulation method.

12.3.1 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.

The class initializes the `Vol_Calculation` object and precomputes the volatility values for the range of log price levels of the underlying assets.

12.3.2 Class Methods

The `MonteCarlo` class includes the following methods:

- `precompute_sigma(self)`: Precomputes and returns sigma (volatility) values for a range of log price levels. It saves the values to a file if not previously saved.

- `price(self, coupon)`: Simulates the price of the financial note over time. It calculates the payoff for each simulation and returns the average payoff.

- `bisection(self, price, tol=1)`: Uses the bisection method to find the coupon rate that results in a given price for the financial note. It iteratively adjusts the coupon rate until the error is within the specified tolerance.

- `get_sigma(self, price, day, index_no)`: Retrieves the precomputed sigma value for a given price level, day, and index number.