Object Oriented Programming II - Final Project Report
Interest Rate Modeling

Atheesh Krishnan // Rajeev Surve // Sagar Marathe

I. Introduction:

The envisioned project is set to explore the domain of interest rate modeling, with a specific emphasis on the Vasicek, Cox-Ingersoll-Ross (CIR), and Black-Derman-Toy (BDT) models. Our objective is to create Python-based implementations of these models to predict future interest rates which could potentially aid in analysis.

This project involves establishing a python framework that allows us to delve into the complexities of each model, evaluating their forecasting accuracy and responsiveness to changes in market conditions.

By comparing these models, our project seeks to connect theoretical financial concepts with their real-world applications, thus deepening our comprehension of interest rate behaviors and their influence on the global financial landscapes. We aim to simulate interest rates for all three models in a Jupyter Notebook environment with plots designed to visualize the paths of the simulated rates, all of the above implemented in an object oriented framework.

II. Procedure:

2.1 Vasicek Model:

- Initialize Parameters: Set the model parameters including the mean reversion speed ('alpha'), long-term mean level ('beta'), volatility of interest rates ('sigma'), and the initial rate ('r0').
- Simulate Interest Rate Path: Generate a path for future interest rates using specified time intervals and the number of steps.
- Calculate Discount Factors and Yields: Compute discount factors for various times, and derive the yield to maturity for zero-coupon bonds.
- Visualize Results: Plot the interest rate paths and the yield curve to analyze and present the model's output.

2.2 CIR Model:

- Set Initial Conditions: Define the initial parameters such as the speed of mean reversion, the long-term mean rate, the volatility, and the starting interest rate.
- Run Simulations: Simulate multiple paths of interest rate evolution over a given time horizon, using specified time steps.
- Ensure Non-Negativity: Apply correction in the simulation to maintain non-negative interest rates.
- Calculate Yield to Maturity: Determine the yield for zero-coupon bonds maturing at different times.
- Plot Interest Rate Paths: Display the simulated paths to visualize variations and trends in the interest rates.

2.3 BDT Model:

- Load Data: Import volatility and discount factors from specified file formats (.xlsx or .csv).
- Compute Short and Forward Rates: Utilize loaded data to calculate short-term interest rates and forward rates at regular intervals.
- Use Numerical Methods: Employ numerical solving techniques (like 'fsolve') to adjust rates based on the model's equations.
- Visual Representation: Graph the expected short rates and forward rates against time to compare and discuss their behaviors.

III. Detailed Model Implementation:

3.1 Vasicek:

$$\Delta r_{t+\Delta t} = (\alpha - \beta r_t)\Delta t + \sigma\epsilon\sqrt{\Delta t}$$

- $\Delta r_{t+\Delta t}$: The change in rate from t to $t + \Delta t$
- $(\alpha - \beta r_t)$: The trend in rates
- σ: Volatility of rates (in basis points)
- ϵ: Standard normal random variable $\sim N(0, 1)$

We define a class 'VasicekModel' to simulate and analyze the behavior of interest rates using the Vasicek stochastic process model, commonly applied in financial mathematics. This model is initialized with parameters that describe the mean reversion speed ('alpha'), the long-term mean interest rate ('beta'), the volatility of the interest rate ('sigma'), and the initial interest rate ('r0'). The class includes methods for validating input parameters to ensure they meet specific conditions necessary for the model's mathematical integrity.

The class contains several functionalities: simulating the path of interest rates over a given time horizon with a specified number of steps, calculating the discount factor for different times to evaluate present values, and deriving the yield to maturity for zero-coupon bonds for varying maturities. Additionally, it features methods to visually represent the yield curve and the simulated path of interest rates using plots.

3.2 CIR:

$$\Delta r_{t+\Delta t} = (\alpha - \beta r_t)\Delta t + \sigma\sqrt{r_t}\epsilon\sqrt{\Delta t}$$

- $\Delta r_{t+\Delta t}$: The change in rate from t to $t + \Delta t$
- $(\alpha - \beta r_t)$: The trend in rates
- σ: Volatility of rates (in basis points)
- ϵ: Standard normal random variable $\sim N(0, 1)$

We define a class 'CIRModel' to simulate and analyze the behavior of interest rates using the Cox-Ingersoll-Ross (CIR) model, a well-known financial model for describing the evolution of interest rates that ensures they remain non-negative. The model is initialized with parameters alpha, beta, sigma, and r0, which represent the speed of mean reversion, long-term mean rate, volatility of the rate, and the initial short rate, respectively. The parameter values are constrained to be between 0 and 1, ensuring they stay within a normalized range, and input validation is applied to check these conditions.

The 'CIRModel' class includes a 'simulate' method for generating simulated paths of interest rates over a specified time horizon with a certain number of paths and time steps. Additionally, the class provides a 'yield_to_maturity' method that calculates the yield to maturity for a zero-coupon bond with a given maturity T. This involves complex formulas that depend on the model's parameters and include terms for adjusting the calculation based on the CIR dynamics.

3.3 BDT:

$$\Delta \ln(r_{t+\Delta t}) = m(t)\Delta t + \sigma(t)\epsilon\sqrt{\Delta t}$$

- $\Delta \ln r_{t+\Delta t}$: The change in the log rate from t to $t + \Delta t$
- $m(t)$: The trend in rates, but a different value of $a(t)$ at each t
- $\sigma(t)$: The volatility of rates also time varying
- ϵ: Standard normal random variable $\sim N(0, 1)$

The BDT model is implemented a bit differently since we had access to some real world data. The Python code defines a class 'BDTModel' to implement the Black-Derman-Toy (BDT) model, which is used for modeling the evolution of interest rates and constructing a binomial interest rate tree.

The constructor of the class takes a 'volatility_file_path' which is expected to point to either an Excel or CSV file containing the necessary data for volatility and discount factors. The 'load_data' method then loads this data into a pandas DataFrame, managing different file formats and structuring the data for further processing.

The rate calculation function iteratively calculates the short-term interest rates based on previously loaded volatility ('sigmas') and discount factors ('D(T)'). It initializes the first rate using a simple logarithmic decay formula from the first discount factor and then for subsequent time points, it uses the previous rate and the volatility to estimate the new rate using a numerical solver ('fsolve'). We then compute forward rates between consecutive times from the discount factors. These forward rates are indicative of future rates expected in the market, calculated from the ratio of successive discount factors.

Visualization utilizes matplotlib and seaborn to plot both the calculated expected short rates and forward rates over time. This visualization helps in comparing the dynamics of these two types of rates as derived from the model.

IV. Object Oriented Design

We have strived to implement all classes and functions and use them in our analysis notebooks in an object oriented manner. The implementation of the Vasicek, CIR, and BDT models using object-oriented design to model financial systems, encapsulating related properties and behaviors within classes.

4.1 Encapsulation

Each model is encapsulated in a class (**'VasicekModel', 'CIRModel', 'BDTModel'**), which groups together the data (attributes) and methods that operate on the data. This encapsulation provides a clear modular structure, making the code easier to manage, debug, and scale.

Attributes: Each class initializes with specific attributes to store its parameters (like 'alpha', 'beta', 'sigma', 'r0') and data essential for computations (e.g., arrays for rates and discount factors in the BDT model).
Methods: Methods are defined within each class to perform tasks specific to that model, such as simulating interest rate paths, calculating yields, or plotting data. This bundling of data and methods within a class prevents data from being manipulated by unrelated parts of the program, thus safeguarding the integrity of the calculations.

4.2 Polymorphism

While explicit polymorphism isn't demonstrated through method overloading or overriding in the provided examples, the design allows for flexible implementation of model-specific calculations under a unified interface. For instance, each model has methods tailored to compute rates and yields differently based on its theoretical framework, yet they could be called in a similar manner from an external control structure.

4.3 Inheritance

Although not explicitly used in the provided scripts, the object-oriented design opens up possibilities for inheritance, where a base class like 'InterestRateModel' could be created to define common interfaces and shared methods for rate calculations, plotting, or data handling. Specific models like Vasicek, CIR, and BDT could inherit from this base class and implement or extend their unique functionalities. This would reduce code redundancy and enhance code reusability.

4.4 Usage of Constructors

Each class uses a constructor method ('__init__') to initialize its attributes. This method is crucial as it sets up the object's state with the necessary parameters to ensure the object is in a valid state before it is used:
- Validation of inputs is performed to ensure that parameters are within expected bounds, which is essential for the robustness and stability of the financial models.

4.5 Method Organization

Data Handling: Methods like 'load_data' in the BDT model manage input/output operations, abstracting these tasks from the core computational logic.
Core Computations: Each model has dedicated methods for its specific computations — for example, simulating paths ('interest_rate_path' or 'simulate'), calculating rates

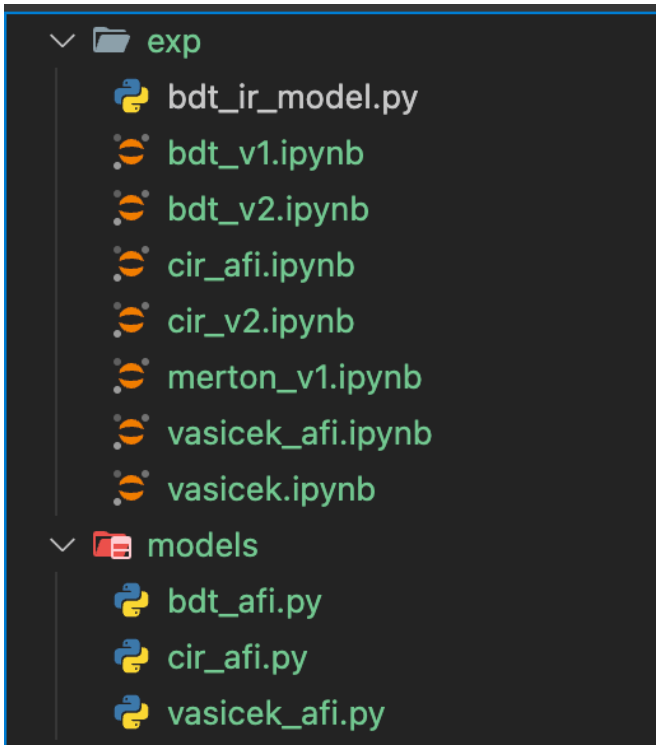
('calculate_rates' or 'yield_to_maturity'), and ensuring constraints like non-negativity of rates in the CIR model.
Visualization: Methods to visualize data help in interpreting the results and are included within the class that generates the data, keeping the display logic closely tied to the data it represents.

V. Experiments

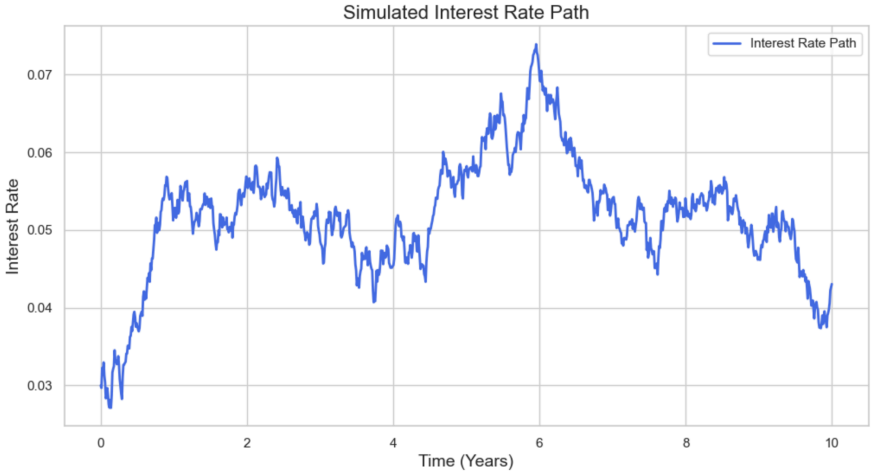
Experiments were created in a Jupyter notebook environment to test and execute code which would be calling the created classes by importing from the source directory. For example:

```
from models.vasicek_afi import VasicekModel
from models.bdt_afi import BDTModel
from models.cir_afi import CIRModel
✓ 1.4s
```

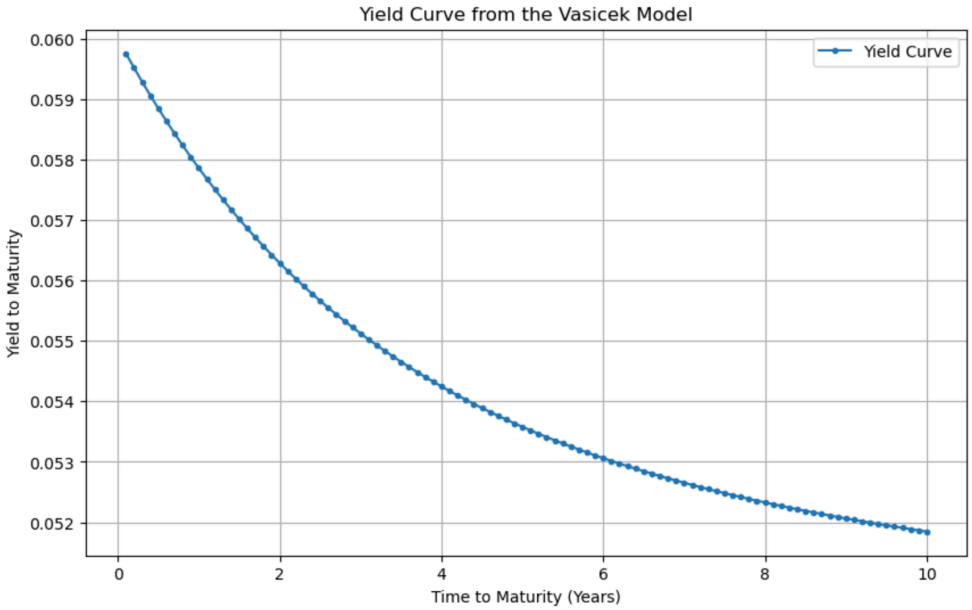
Based on the following directory structure:



5.1 VASICEK:



An occurrence of an inverted yield curve based on our input params:

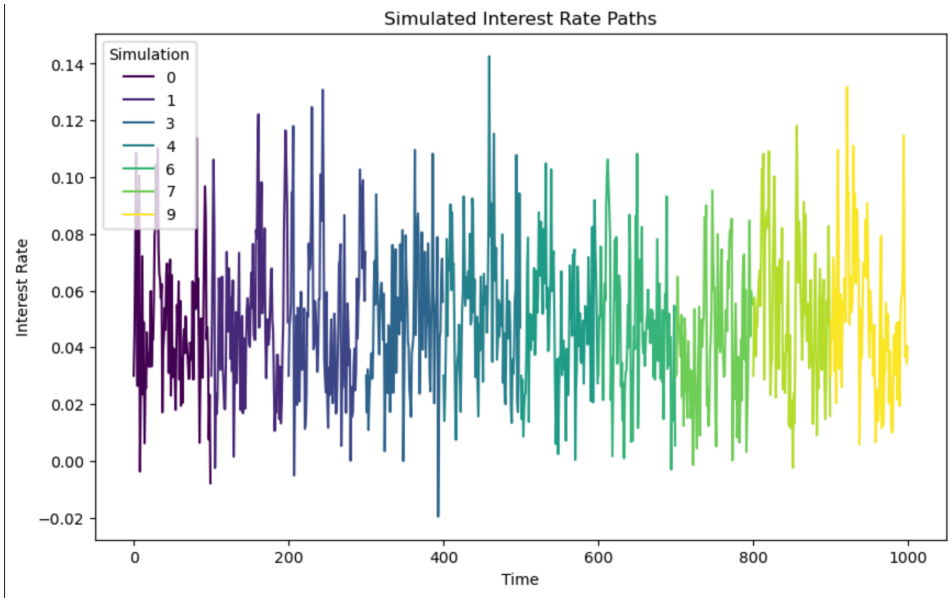


5.2 CIR:

Model for the following params:

```
reversion_rate=0.5, long_term_mean=0.05, volatility=0.1, initial_rate=0.03
```

Multiple intertemporal simulations:



5.3 BDT:

Current data for BDT model:

	Time	Sigmas	D(T)
0	0.5	NaN	0.972476
1	1.0	0.100	0.944569
2	1.5	0.120	0.916324
3	2.0	0.135	0.887934
4	2.5	0.150	0.859741
5	3.0	0.160	0.832044

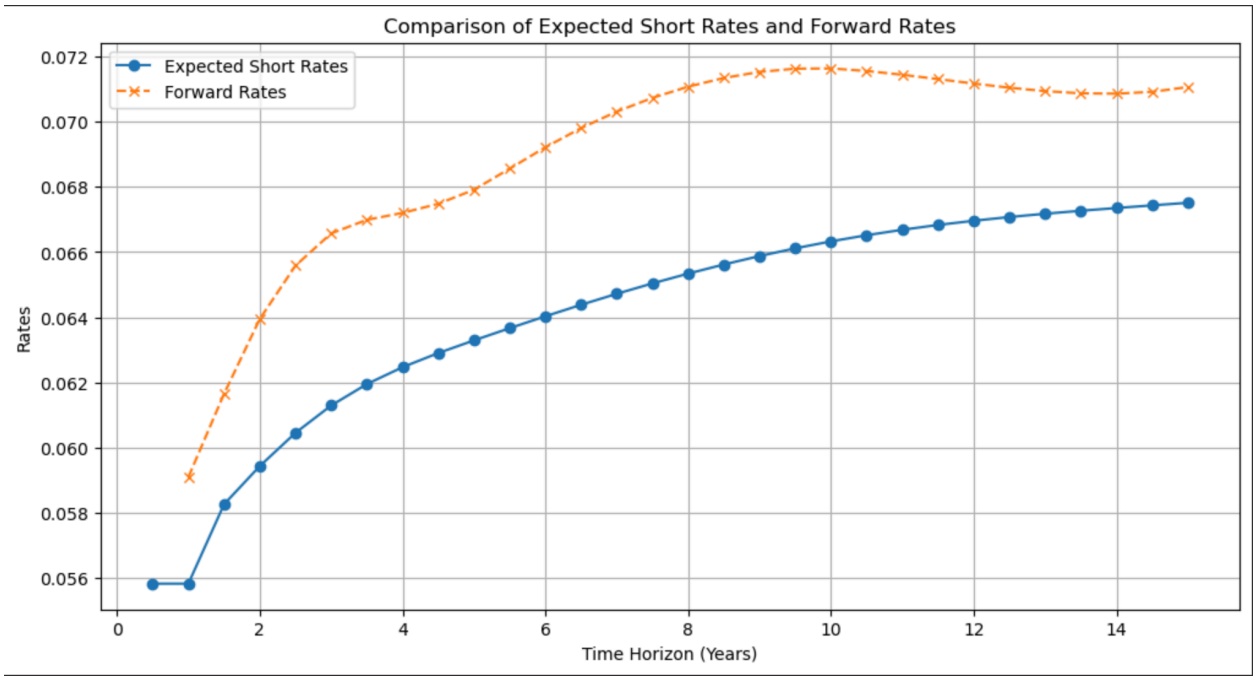
```
for T, rate in zip(times, rates):
    print(f"For Time: {T} years, Rate: {rate:.4%}")
```

```
For Time: 0.5 years, Rate: 5.5819%
For Time: 1.0 years, Rate: 5.5819%
For Time: 1.5 years, Rate: 5.8257%
For Time: 2.0 years, Rate: 5.9429%
For Time: 2.5 years, Rate: 6.0450%
For Time: 3.0 years, Rate: 6.1290%
For Time: 3.5 years, Rate: 6.1948%
For Time: 4.0 years, Rate: 6.2467%
For Time: 4.5 years, Rate: 6.2901%
For Time: 5.0 years, Rate: 6.3289%
For Time: 5.5 years, Rate: 6.3664%
For Time: 6.0 years, Rate: 6.4030%
For Time: 6.5 years, Rate: 6.4382%
For Time: 7.0 years, Rate: 6.4720%
For Time: 7.5 years, Rate: 6.5040%
For Time: 8.0 years, Rate: 6.5340%
For Time: 8.5 years, Rate: 6.5620%
```

```python
forward_rates = []

for i in range(1, len(discount_factors)):
    # 'm' year loan that starts at time 'n' years from now
    n = times[i - 1]
    m = times[i] - times[i - 1]
    # Discount factors for the time intervals
    D_n = discount_factors[i - 1]
    D_nm = discount_factors[i]

    forward_rate = (D_n / D_nm - 1) / m
    forward_rates.append(forward_rate)

print('Forward Rates:')
for i, rate in enumerate(forward_rates, 1):
    print(f"f({times[i-1]}, {times[i]-times[i-1]}): {rate:.4%}")
```

```
Forward Rates:
f(0.5, 0.5): 5.9090%
f(1.0, 0.5): 6.1650%
f(1.5, 0.5): 6.3946%
f(2.0, 0.5): 6.5585%
f(2.5, 0.5): 6.6575%
f(3.0, 0.5): 6.6992%
f(3.5, 0.5): 6.7204%
f(4.0, 0.5): 6.7484%
f(4.5, 0.5): 6.7916%
f(5.0, 0.5): 6.8563%
f(5.5, 0.5): 6.9219%
f(6.0, 0.5): 6.9808%
```

Comparison using Interest Rate Matching:



VI. Challenges

First, accurately translating complex financial theories into computational algorithms requires a deep understanding of both the financial concepts and software development principles. Ensuring that the implemented models faithfully represent the theoretical models while also being computationally efficient and stable is difficult, especially when dealing with stochastic differential equations and their discretizations, which have introduced numerical errors, which then need to be fixed.

Another significant challenge is handling data integrity and input validation. For instance, ensuring non-negative interest rates in the CIR model requires careful mathematical manipulation and programming safeguards. Not implementing such a safeguard will result in exploding results. Additionally, managing external data, especially when it comes from varied formats like CSV and Excel, adds complexity in ensuring the data is loaded and parsed correctly without introducing errors.

VII. Future Scope

- Interest rate modeling is an ocean of a subject and there is a laundry list of topics and other models to be explored, with each being more advanced than the last.
- Better structure for the models, in the form of a python package which can then be called at will for the purposes of simulation/analysis.
- An attempt could be made to incorporate machine learning into the models as an alternative to purely theoretical modeling.
- Real time data feeds: While our current implementation uses static values which are then passed on to the formula for the model, a more robust framework would take in historical data and also update with real time data as it comes.
- Cross asset modeling: We could also try to check for usages of interest rate in other sectors and assets like credit risk models and equity models.

VIII. Conclusion

Throughout this exploration, we've delved into the implementation and application of three foundational interest rate models: the Vasicek, CIR, and BDT models, each encapsulated within its own class using object-oriented programming principles. This approach not only streamlined the complex mathematical operations inherent to financial modeling but also enhanced the clarity and modularity of the code. These models were employed to simulate interest rate paths, calculate yield curves, and plot the results, providing valuable insights into how interest rates might evolve under various economic scenarios. The use of rigorous input validation and error handling further ensured the robustness and reliability of the simulations.

Looking ahead, there is substantial scope for expanding these models by incorporating more sophisticated algorithms, integrating real-time data feeds, and applying machine learning techniques to enhance predictive capabilities. Such advancements could transform these foundational models into more dynamic tools for real-time financial decision-making and risk management. This project sets a solid groundwork, offering a clear pathway for future enhancements that could make significant contributions to the field of financial modeling.

IX. References

- IR Models Course - École Polytechnique Fédérale de Lausanne
- Ch17. Interest Rate Model - MIT Press
- Interest Rate Modeling - Vol II - Term Structures - Andersen & Pitarberg
- Quantpie - Interest Rate Models - YouTube
- https://aleatory.readthedocs.io/en/latest/processes_euler_maruyama.html
- https://www.federalreserve.gov/data/yield-curve-models.htm
- https://www.imf.org/en/Publications/
- The Art of Term Structure Models

X. Appendix

The following code are the main python classes written with all the implemented functionality. Practically, these classes are then imported to Jupyter for analysis and graphing for each of the three interest rate models.

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

class VasicekModel:
    def __init__(self, alpha, beta, sigma, r0):
        """
        Initialize the Vasicek model with the given parameters.
        :param alpha: Speed of reversion to the mean
        :param beta: Long-term mean level
        :param sigma: Volatility of interest rates
        :param r0: Initial short rate
        """
        # Input validation
        if not isinstance(alpha, (int, float)) or alpha <= 0:
            raise ValueError("Alpha must be a positive number.")
        if not isinstance(beta, (int, float)):
            raise ValueError("Beta must be a number.")
        if not isinstance(sigma, (int, float)) or sigma <= 0:
            raise ValueError("Sigma must be a positive number.")
        if not isinstance(r0, (int, float)) or r0 < 0:
            raise ValueError("r0 must be a non-negative number.")

        self.alpha = alpha
        self.beta = beta
        self.sigma = sigma
        self.r0 = r0

    def interest_rate_path(self, time_horizon, steps):
        """
        Generate an interest rate path over the time horizon.
        :param time_horizon: Total time horizon for the rate evolution
        :param steps: Number of steps in the simulation
        :return: A numpy array of interest rates
        """
        # Input Validation
        if not isinstance(time_horizon, (int, float)) or time_horizon <= 0:
            raise ValueError("Time horizon must be a positive number.")
        if not isinstance(steps, int) or steps <= 0:
            raise ValueError("Steps must be a positive integer.")
```

```
        dt = time_horizon / steps
        rates = np.zeros(steps)
        rates[0] = self.r0
        for t in range(1, steps):
            dr = self.alpha * (self.beta - rates[t - 1]) * dt + self.sigma * np.sqrt(dt) * np.random.normal()
            rates[t] = rates[t - 1] + dr
        return rates

    def discount_factor(self, T):
        """
        Calculate the discount factor for a given time T.
        :param T: Time in years
        :return: Discount factor
        """
        B = (1 - np.exp(-self.alpha * T)) / self.alpha
        A = np.exp((self.beta - self.sigma**2 / (2 * self.alpha**2)) * (B - T) - (self.sigma**2 / (4 *
self.alpha)) * B**2)
        return A * np.exp(-B * self.r0)

    def yield_to_maturity(self, T):
        """
        Calculate the yield to maturity for a T-year zero coupon bond.
        :param T: Time to maturity in years
        :return: Yield to maturity
        """
        A = self.discount_factor(T)
        B = (1 - np.exp(-self.alpha * T)) / self.alpha
        ytm = -np.log(A) / T + B / T * self.r0
        return ytm

    def plot_yield_curve(self, max_maturity, num_points):
        """
        Plot the yield curve based on the Vasicek model.
        :param max_maturity: The maximum maturity for plotting
        :param num_points: The number of points on the curve
        """
        maturities = np.linspace(0.1, max_maturity, num_points)
        yields = np.array([self.yield_to_maturity(T) for T in maturities])

        plt.figure(figsize=(10, 6))
        plt.plot(maturities, yields, label='Yield Curve', marker='.')
        plt.title('Yield Curve from the Vasicek Model')
        plt.xlabel('Time to Maturity (Years)')
        plt.ylabel('Yield to Maturity')
        plt.legend()
        plt.grid(True)
        plt.show()
```

```
    def plot_interest_rate_path(self, time_horizon, steps):
        """
        Plot the interest rate path.
        :param time_horizon: The time horizon for the rate evolution
        :param steps: The number of steps in the simulation
        """
        rates = self.interest_rate_path(time_horizon, steps)
        time_steps = np.linspace(0, time_horizon, steps)

        plt.figure(figsize=(10, 6))
        plt.plot(time_steps, rates, label='Interest Rate Path', lw=2)
        plt.title('Simulated Interest Rate Path from the Vasicek Model')
        plt.xlabel('Time (Years)')
        plt.ylabel('Interest Rate')
        plt.legend()
        plt.grid(True)
        plt.show()

# CIR
import numpy as np

class CIRModel:
    def __init__(self, alpha, beta, sigma, r0):
        """
        Initialize the CIR model with the given parameters.
        :param alpha: Speed of reversion to the mean
        :param beta: Long-term mean rate
        :param sigma: Volatility of the rate
        :param r0: Initial short rate
        """
        if not (0 <= alpha <= 1):
            raise ValueError("alpha must be between 0 and 1")
        if not (0 <= beta <= 1):
            raise ValueError("beta must be between 0 and 1")
        if not (0 <= sigma <= 1):
            raise ValueError("sigma must be between 0 and 1")
        if not (0 <= r0 <= 1):
            raise ValueError("r0 must be between 0 and 1")

        self.alpha = alpha
        self.beta = beta
        self.sigma = sigma
        self.r0 = r0

    def simulate(self, time_horizon, num_paths, timestep):
        """
        Simulate interest rate paths using the Cox-Ingersoll-Ross (CIR) model.
```

```
        Parameters
        ----------
        time_horizon : float
            The total length of time for the simulation.
        num_paths : int
            The number of paths to simulate.
        timestep : float
            The time step size for the simulation.

        Returns
        -------
        rates : np.ndarray
            A 2D array of shape (num_paths, num_steps + 1) containing the simulated interest rate paths.

        Raises
        ------
        ValueError
            If time_horizon, num_paths, or timestep are not positive, or if timestep is greater than or equal
to time_horizon.
        """
        if not (0 < time_horizon):
            raise ValueError("time_horizon must be positive")
        if not (0 < num_paths):
            raise ValueError("num_paths must be positive")
        if not (0 < timestep < time_horizon):
            raise ValueError("timestep must be positive and less than time_horizon")

        num_steps = int(time_horizon / timestep)
        rates = np.zeros((num_paths, num_steps + 1))
        rates[:, 0] = self.r0
        for t in range(num_steps):
            dt = timestep
            sqrt_rates = np.sqrt(rates[:, t])
            dw = np.random.normal(size=num_paths) * np.sqrt(dt)
            dr = self.alpha * (self.beta - rates[:, t]) * dt + self.sigma * sqrt_rates * dw
            rates[:, t+1] = rates[:, t] + dr
            rates[rates < 0] = 0  # Ensure non-negative rates
        return rates

    def yield_to_maturity(self, T):
        """
        Calculate the yield to maturity for a T-year zero coupon bond.

        Parameters
        ----------
        T : float
            The time to maturity of the bond.
```

```
        Returns
        -------
        yield_curve : float
            The yield to maturity of the bond.
        """
        gamma = np.sqrt(self.alpha**2 + 2*self.sigma**2)
        term1 = (2*gamma*np.exp((self.alpha+gamma)*T/2)) / (2*gamma +
(self.alpha+gamma)*(np.exp(gamma*T) - 1))
        term2 = 2*self.alpha*self.beta / self.sigma**2
        A_T = np.power(term1, term2)
        B_T = (2*(np.exp(gamma*T) - 1)) / (2*gamma + (self.alpha+gamma)*(np.exp(gamma*T) - 1))
        yield_curve = -np.log(A_T)/T + B_T*self.r0/T
        return yield_curve
```

BDT:

```
import numpy as np
import pandas as pd
from scipy.optimize import fsolve
import seaborn as sns
import matplotlib.pyplot as plt

class BDTModel:
    def __init__(self, volatility_file_path):
        self.volatility_file_path = volatility_file_path
        self.data = None
        self.rates = []
        self.forward_rates = []

    def load_data(self):
        if self.volatility_file_path.endswith('.xlsx'):
            xls = pd.ExcelFile(self.volatility_file_path)
            sigmas = pd.read_excel(xls, sheet_name='Sigmas', header=None, names=['Sigmas'])
            disc_facs = pd.read_excel(xls, sheet_name='D(T)', header=None, names=['D(T)'])
        elif self.volatility_file_path.endswith('.csv'):
            sigmas = pd.read_csv(self.volatility_file_path, header=None, names=['Sigmas'])
            disc_facs = pd.read_csv(self.volatility_file_path, header=None, names=['D(T)'])
        else:
            raise ValueError("Invalid file format. Only .xlsx and .csv files are supported.")

        time_intervals = pd.DataFrame({'Time': [i * 0.5 for i in range(1, 31)]})
        new_row = pd.DataFrame({'Sigmas': [np.nan]})
        sigmas = pd.concat([new_row, sigmas], ignore_index=True).head(30)
        self.data = pd.concat([time_intervals, sigmas, disc_facs], axis=1)

    def calculate_rates(self):
        times = self.data['Time'].values
        self.rates = [-np.log(self.data['D(T)'][0]) / times[0]]
```

```
        for i in range(1, len(times)):
            T = times[i]
            sigma = self.data['Sigmas'][i-1]
            D = self.data['D(T)'][i]
            new_rate = self.calculate_rate(self.rates[-1], sigma, D, T)
            self.rates.append(new_rate)

        self.data['Rates'] = self.rates

    def calculate_rate(self, previous_rate, sigma, D, T):
        if np.isnan(sigma):
            return previous_rate

        def f(r):
            r_u = previous_rate * np.exp(sigma * np.sqrt(T))
            r_d = previous_rate * np.exp(-sigma * np.sqrt(T))
            return np.exp(-r * T) - D  # Bond price

        rate = fsolve(f, previous_rate)[0]
        return rate

    def calculate_forward_rates(self):
        discount_factors = self.data['D(T)'].values
        times = self.data['Time'].values

        for i in range(1, len(discount_factors)):
            n = times[i - 1]
            m = times[i] - times[i - 1]
            D_n = discount_factors[i - 1]
            D_nm = discount_factors[i]
            forward_rate = (D_n / D_nm - 1) / m
            self.forward_rates.append(forward_rate)

    def plot_rates(self):
        plt.figure(figsize=(12, 6))
        sns.lineplot(data=self.data, x='Time', y='Rates', marker='o', label='Expected Short Rates')
        sns.lineplot(data=self.data, x='Time', y=[np.nan] + self.forward_rates, marker='x', label='Forward
Rates', linestyle='--')
        plt.title('Comparison of Expected Short Rates and Forward Rates')
        plt.xlabel('Time Horizon (Years)')
        plt.ylabel('Rates')
        plt.legend()
        plt.grid(True)
        plt.show()
```