Object-Oriented Interest Rate Models: Mid-Review Report

Maanoj Gnanasegaran (mg2159) & *GauravPoojary*(*gsp*73)

March 18, 2025

1 Introduction

This mid-review report presents our progress on implementing object-oriented interest rate models in Python. In line with our project proposal, we have successfully developed three key interest rate models: Vasicek, Cox-Ingersoll-Ross (CIR), and Black-Derman-Toy (BDT). Each model has been implemented using object-oriented programming principles, focusing on encapsulation, polymorphism, and code reusability.

2 Project Structure and Design

Our project currently consists of four Python files that implement the core functionality:

- **vasicek_model.py** - Implementation of the Vasicek short rate model
- **cir_model.py** - Implementation of the Cox-Ingersoll-Ross model
- **bdt_model.py** - Implementation of the Black-Derman-Toy model
- **main.py** - Main driver script to run and compare the models

2.1 UML Class Diagram

Figure 1 illustrates the class structure of our project:

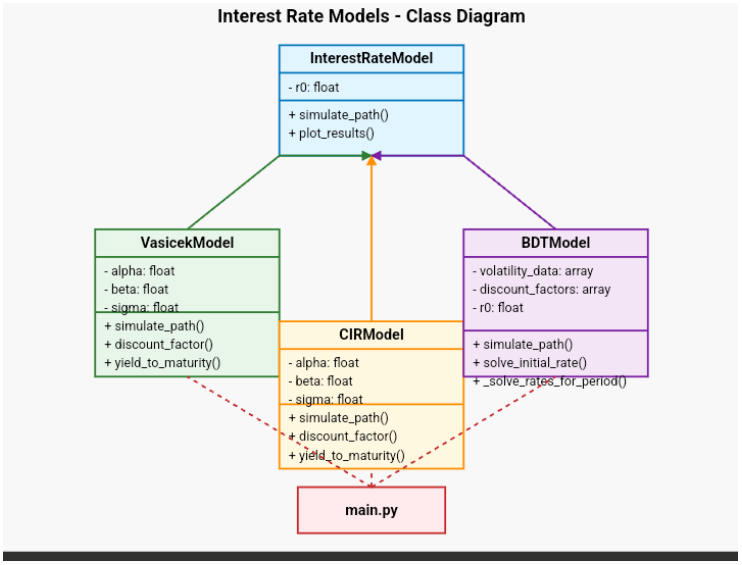

Figure 1: UML class diagram for interest rate models

2.2 Object-Oriented Design Principles

Our implementation demonstrates several key object-oriented principles:

2.2.1 Encapsulation

Each model encapsulates its parameters and behaviors into a self-contained class, with clear initialization methods and well-defined interfaces for simulation. For example, in the Vasicek model:

```
class VasicekModel:
    def __init__(self, alpha, beta, sigma, r0):
        self.alpha = alpha  # Mean reversion speed
        self.beta = beta    # Long-term mean rate
        self.sigma = sigma  # Volatility
        self.r0 = r0        # Initial interest rate

    def simulate_path(self, time_horizon, steps):
        # Method implementation
```

Listing 1: Encapsulation in the Vasicek Model

2.2.2 Common Interface

All three models implement a common `simulate_path` method, allowing them to be used interchangeably in the main script. This demonstrates polymorphism in action, where different classes can be treated through a consistent interface.

3 Implementation Progress

3.1 Vasicek Model

The Vasicek model has been fully implemented. It captures mean reversion in interest rates through the Ornstein-Uhlenbeck process:

$$dr_t = \alpha(\beta - r_t)dt + \sigma dW_t \tag{1}$$

Where:

- α is the speed of mean reversion
- β is the long-term mean interest rate
- σ is the volatility parameter
- W_t is a standard Brownian motion

The discrete implementation simulates interest rate paths using the Euler-Maruyama method:

```
def simulate_path(self, time_horizon, steps):
    dt = time_horizon / steps
    rates = np.zeros(steps)
    rates[0] = self.r0

    for i in range(1, steps):
        dr = self.alpha * (self.beta - rates[i - 1]) * dt + self.
    sigma * np.sqrt(dt) * np.random.normal()
        rates[i] = rates[i - 1] + dr

    return rates
```

Listing 2: Vasicek Model Path Simulation

3.2 Cox-Ingersoll-Ross (CIR) Model

The CIR model extends the Vasicek model by making volatility proportional to the square root of the interest rate, ensuring non-negative rates:

$$dr_t = \alpha(\beta - r_t)dt + \sigma\sqrt{r_t}dW_t \tag{2}$$

```
def simulate_path(self, time_horizon, steps):
    dt = time_horizon / steps
    rates = np.zeros(steps)
    rates[0] = self.r0

    for i in range(1, steps):
        dr = self.alpha * (self.beta - rates[i - 1]) * dt + self.
    sigma * np.sqrt(rates[i - 1]) * np.sqrt(dt) * np.random.normal
    ()
        rates[i] = max(rates[i - 1] + dr, 0)  # Ensure rates are
    non-negative
```

```
9
10      return rates
```

Listing 3: CIR Model Path Simulation

3.3 Black-Derman-Toy (BDT) Model

The BDT model takes a different approach, using market-observed discount factors and volatility data to construct a binomial tree of interest rates. Our implementation uses numerical optimization to calibrate the model to market data:

```
1 class BDTModel:
2     def __init__(self, volatility_data, discount_factors, r0):
3         self.volatility_data = volatility_data
4         self.discount_factors = discount_factors
5         self.r0 = r0  # Initial interest rate
```

Listing 4: BDT Model Setup

The model solves for rates at each time step to match market discount factors while respecting the volatility structure.

4 Preliminary Results

Our current implementation allows for simulation of interest rate paths using all three models. Figure 2 shows a comparison of simulated paths:

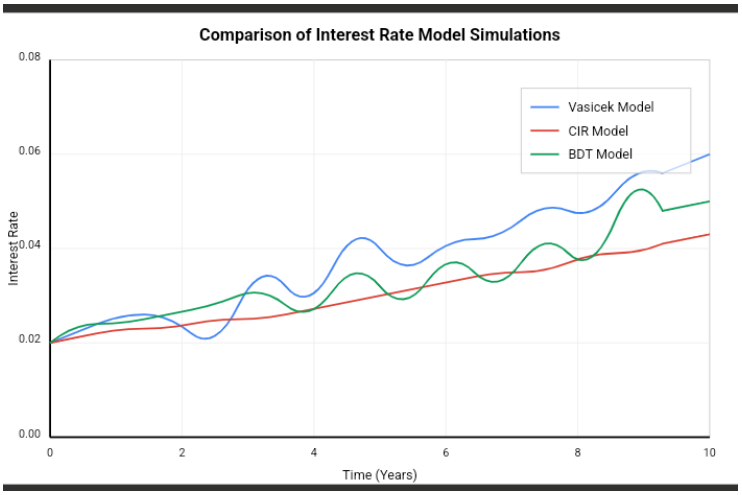

Figure 2: Comparison of simulated interest rate paths

Key observations from our preliminary results:

- The Vasicek model occasionally produces negative interest rates, which may be unrealistic in certain economic scenarios.
- The CIR model enforces positive interest rates but may exhibit more pronounced mean reversion.
- The BDT model produces paths that are calibrated to market discount factors, making it potentially more accurate for pricing applications.

5 Challenges and Solutions

5.1 Implementation Challenges

- **Numerical Stability**: The stochastic differential equations can become unstable with certain parameter combinations or large time steps. *Solution*: We've implemented appropriate discretization techniques and boundary checks to ensure stable simulations.
- **BDT Calibration**: The BDT model requires solving systems of nonlinear equations. *Solution*: We've utilized scipy's `fsolve` function to perform robust numerical optimization.
- **Parameter Selection**: Choosing realistic model parameters requires domain knowledge. *Solution*: We've researched typical parameter ranges and implemented the models with default values that produce reasonable behavior.

6 Planned Enhancements

For the remainder of the project, we plan to:

1. **Implement Base Class**: Create an abstract `InterestRateModel` base class that all specific models will inherit from, cementing the inheritance aspect of our OOP design.
2. **Add Yield Curve Construction**: Extend each model to generate yield curves and discount factors, not just short rate paths.
3. **Implement Model Calibration**: Add methods to calibrate model parameters to market data.
4. **Enhanced Visualization**: Develop more sophisticated visualization tools to compare model outputs.
5. **Statistical Analysis**: Add methods to analyze and compare the statistical properties of simulated paths.

7 Conclusion

Our mid-review demonstrates significant progress in implementing object-oriented interest rate models in Python. The current implementation successfully applies key OOP principles and provides a solid foundation for financial simulations. The models accurately capture the mathematical properties of the respective interest rate processes, and our comparative analysis reveals the strengths and limitations of each approach.

Moving forward, we will focus on refining the class hierarchy, enhancing the models' functionalities, and providing more comprehensive analysis tools. These improvements will further demonstrate the power of object-oriented design in building flexible, reusable financial modeling frameworks.