Valuation of Exotic Derivatives and Fixed Income Securities

MGMTMFE 432 – Project 3 Report

Vikalp Thukral
UID: 406534669

June 7, 2025

Abstract

This project implements and analyzes advanced financial models relevant to the valuation of exotic derivatives and fixed income securities. The first part of the analysis investigates a borrower's embedded default option in the presence of jump risk, where the collateral follows a jump-diffusion process and the loan payoff structure is dynamic. Using Monte Carlo simulation, we estimate the value of the default option, its probability of being exercised, and the expected timing of default, conditional on a range of jump intensities and loan maturities.

Next, we focus on the pricing of exotic path-dependent options, particularly Down-and-Out Put options, under a stochastic volatility framework. The model incorporates two risk factors—asset price and variance—modeled using a full truncation scheme to ensure positivity of volatility during simulation. Several barrier structures are examined, and we compare option prices across varying market parameters.

The third segment turns to the pricing of fixed income products using the Cox-Ingersoll-Ross (CIR) short rate model. We simulate the evolution of interest rates to value both a coupon-bearing bond and a European call option on a zero-coupon bond, applying both Monte Carlo methods and finite-difference PDE solvers for validation and sensitivity analysis.

Building on term structure modeling, we simulate the G2++ model to price European put options on pure discount bonds. The model captures two correlated mean-reverting factors, enabling a more flexible fit to the observed yield curve. Analytical pricing results are compared to simulation-based estimates to evaluate model performance under different correlation structures.

Finally, the project evaluates mortgage-backed securities (MBS) and their tranches using the CIR model in conjunction with the Numerix prepayment framework. We price full MBS pools and decompose cash flows to value Interest-Only (IO) and Principal-Only (PO) tranches. Additionally, we compute the option-adjusted spread (OAS) and assess the effect of prepayment risk and long-term interest rate assumptions on tranche valuations.

All numerical methods are modular, robust, and parameter-driven. Results are supported by professional-grade visualizations and include interpretations rooted in financial theory and empirical behavior.

Contents

1 Question 1: Embedded Default Option under Jump-Diffusion Collateral Dynamics

1.1 Python Implementation

The default option is modeled as a barrier-style American option embedded within a collateralized loan. The collateral process follows a jump-diffusion, and the borrower exercises the option optimally when the collateral value falls below a stochastic threshold. The following Python function simulates this behavior via Monte Carlo:

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

def Proj3_2func(lambda1_val=0.2, T_val=5, seed=42, n_paths=1000):
    """
    Simulates default option value, default probability, and expected exercise time
    under a jump-diffusion model for a range of lambda1 and T values.

    Parameters
    ----------
    lambda1_val : float
        Poisson jump intensity λ₁ used for default output (default = 0.2)
    T_val : float
        Loan maturity in years used for default output (default = 5)
    seed : int
        Random seed for reproducibility
    n_paths : int
        Number of Monte Carlo simulation paths

    Returns
    -------
    D : float
        Default option value for λ₁ = lambda1_val and T = T_val
    Prob : float
        Default probability for λ₁ = lambda1_val and T = T_val
    Et : float
        Expected time of default conditional on τ ≤ T
    """
```

vthukral

Figure 1: Python code implementation (Part 1 of 3)

```
for i, T in enumerate(T_vals):
    for j, lambda1 in enumerate(lambda_vals):
        beta = (eps - alpha) / T
        R = r0 + delta * lambda2
        r = R / 12
        n_months = int(T * 12)
        c = 1 + r
        PMT = (L0 * r) / (1 - (1 + r) ** -n_months)
        a = PMT / r
        b = PMT / (r * (1 + r) ** n_months)
        steps = int(T / dt)

        D_vals = []
        tau_vals = []
        default_flags = []

        for _ in range(n_paths):
            V = V0
            defaulted = False
            for t in range(1, steps + 1):
                time = t * dt
                Z = np.random.normal()
                J = np.random.poisson(lambda1 * dt)
                V *= (1 + mu * dt + sigma * np.sqrt(dt) * Z + gamma * J)
                Lt = a - b * c ** (12 * time)
                qt = alpha + beta * time
                if not defaulted and V <= qt * Lt:
                    payoff = max(Lt - eps * V, 0)
                    D_vals.append(np.exp(-r0 * time) * payoff)
                    tau_vals.append(time)
                    default_flags.append(1)
                    defaulted = True
                    break
            if not defaulted:
                D_vals.append(0)
                default_flags.append(0)

        D_matrix[i, j] = np.mean(D_vals)
        Prob_matrix[i, j] = np.mean(default_flags)
        Et_matrix[i, j] = np.mean([tau for flag, tau in zip(default_flags, tau_vals) if flag]) if any(default_flags) else 0

idx_T = np.where(T_vals == T_val)[0][0]
idx_lambda = np.where(np.isclose(lambda_vals, lambda1_val))[0][0]
D = D_matrix[idx_T, idx_lambda]
Prob = Prob_matrix[idx_T, idx_lambda]
Et = Et_matrix[idx_T, idx_lambda]

legend_labels = [f"λ₁ = {l:.2f}" for l in lambda_vals]
styled_plot(T_vals, D_matrix, "Default Option Value", "Default Option Value", legend_labels)
styled_plot(T_vals, Prob_matrix, "Default Probability", "Probability of Default (τ ≤ T)", legend_labels)
styled_plot(T_vals, Et_matrix, "Expected Default Time", "Expected Default Time (τ | τ ≤ T)", legend_labels)

return D, Prob, Et
```


Figure 2: Python code implementation (Part 2 of 3)

```python
def styled_plot(x_vals, y_matrix, title, ylabel, legend_labels):
    plt.figure(figsize=(10, 6))
    plt.rcParams['axes.facecolor'] = '#fff8dc'
    plt.rcParams['savefig.facecolor'] = '#fff8dc'

    colors = ['#4d602a', '#bb0a1e', '#0066cc', '#0c2478',
              '#131212', '#104071', '#313390', '#dd4400', '#242b39']
    line_styles = ['-', '--', '-.', ':', '-', '--', '-.', ':', '-']

    for i in range(len(legend_labels)):
        plt.plot(x_vals, y_matrix[:, i],
                 label=legend_labels[i],
                 color=colors[i % len(colors)],
                 linestyle=line_styles[i % len(line_styles)],
                 linewidth=2)

    plt.title(title, fontsize=14, fontweight='bold')
    plt.xlabel("T (Years)", fontsize=12)
    plt.ylabel(ylabel, fontsize=12)
    plt.xticks(fontsize=10)
    plt.yticks(fontsize=10)
    plt.grid(visible=True, linestyle=':', linewidth=0.5, alpha=0.7)
    plt.legend(fontsize=10, title="Jump Intensity λ₁", title_fontsize=11)
    plt.tight_layout()
    plt.show()

np.random.seed(seed)

V0 = 20000
L0 = 22000
mu = -0.1
sigma = 0.2
gamma = -0.4
r0 = 0.055
delta = 0.25
lambda2 = 0.4
alpha = 0.7
eps = 0.95

lambda_vals = np.arange(0.05, 0.45, 0.05)
T_vals = np.arange(3, 9, 1)
dt = 1 / 12

D_matrix = np.zeros((len(T_vals), len(lambda_vals)))
Prob_matrix = np.zeros((len(T_vals), len(lambda_vals)))
Et_matrix = np.zeros((len(T_vals), len(lambda_vals)))
```

vthukral

Figure 3: Python code implementation (Part 3 of 3)

1.2 Mathematical Model

The collateral value V_t evolves according to the following stochastic differential equation with jumps:

$$\frac{dV_t}{V_{t^-}} = \mu\, dt + \sigma\, dW_t + \gamma\, dJ_t$$

where:

- $\mu < 0$ is the drift (reflecting collateral decay),
- σ is the volatility,
- W_t is a Brownian motion,
- J_t is a Poisson process with intensity λ_1,
- $\gamma < 0$ represents the jump magnitude (collateral crash),
- V_{t^-} is the pre-jump value at time t.

The outstanding loan balance L_t follows an exponential amortization curve:

$$L_t = a - b \cdot c^{12t}, \quad \text{with constants derived from PMT formula.}$$

Default occurs at stopping time:

$$\tau = \min\{t \geq 0 : V_t \leq q_t \cdot L_t\}, \quad \text{with } q_t = \alpha + \beta t, \quad \beta = \frac{\varepsilon - \alpha}{T}$$

If $\tau \leq T$, the borrower exercises the default option, with payoff:

$$(L_\tau - \varepsilon V_\tau)^+$$

Monte Carlo paths simulate these dynamics and estimate:

- $\mathbb{E}[\text{Payoff}] \Rightarrow$ Default Option Value,
- $\mathbb{P}(\tau \leq T) \Rightarrow$ Default Probability,
- $\mathbb{E}[\tau \mid \tau \leq T] \Rightarrow$ Expected Default Time.

1.3 Simulation Results

The simulation was executed using 100,000 Monte Carlo paths across a grid of jump intensities $\lambda_1 \in \{0.05, 0.10, \ldots, 0.40\}$ and loan maturities $T \in \{3, 4, \ldots, 8\}$ years. The three figures below illustrate the behavior of the default option value, default probability, and conditional expected time of default.

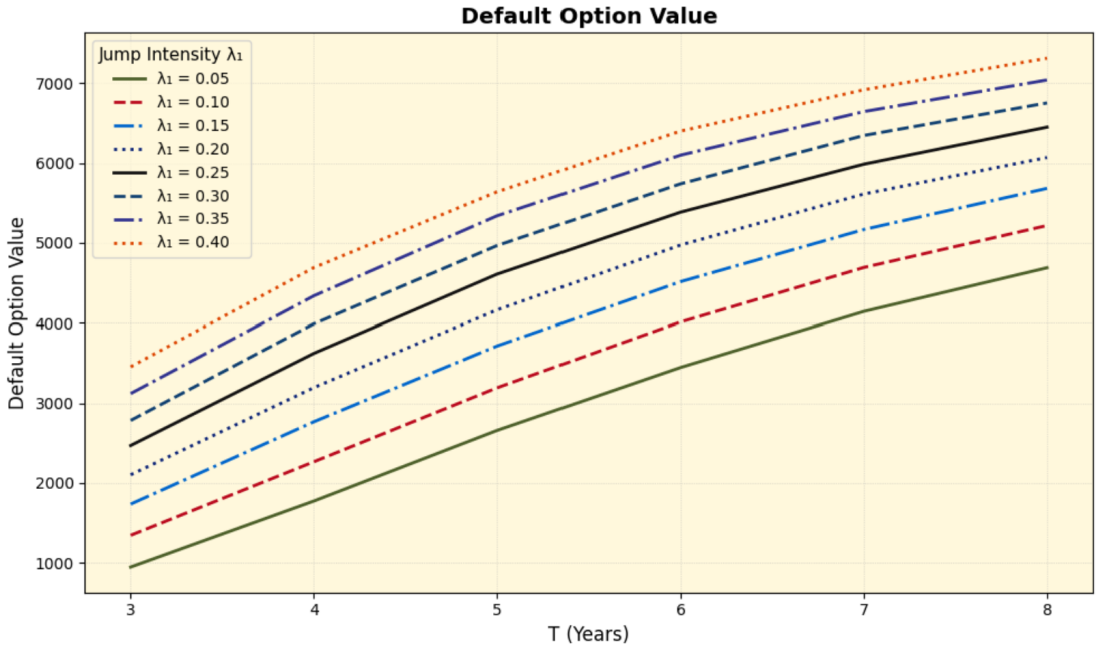

Figure 4: Default Option Value across $T \in \{3, \ldots, 8\}$ and $\lambda_1 \in \{0.05, \ldots, 0.4\}$

Figure 4 shows that the default option value increases with both maturity and jump intensity. Longer loan horizons increase the likelihood of collateral deterioration, and higher jump intensities lead to more frequent negative jumps, increasing the borrower's incentive to default. The relationship is convex in T and roughly linear in λ_1, consistent with the option's time value and the nonlinearity introduced by jump risk.

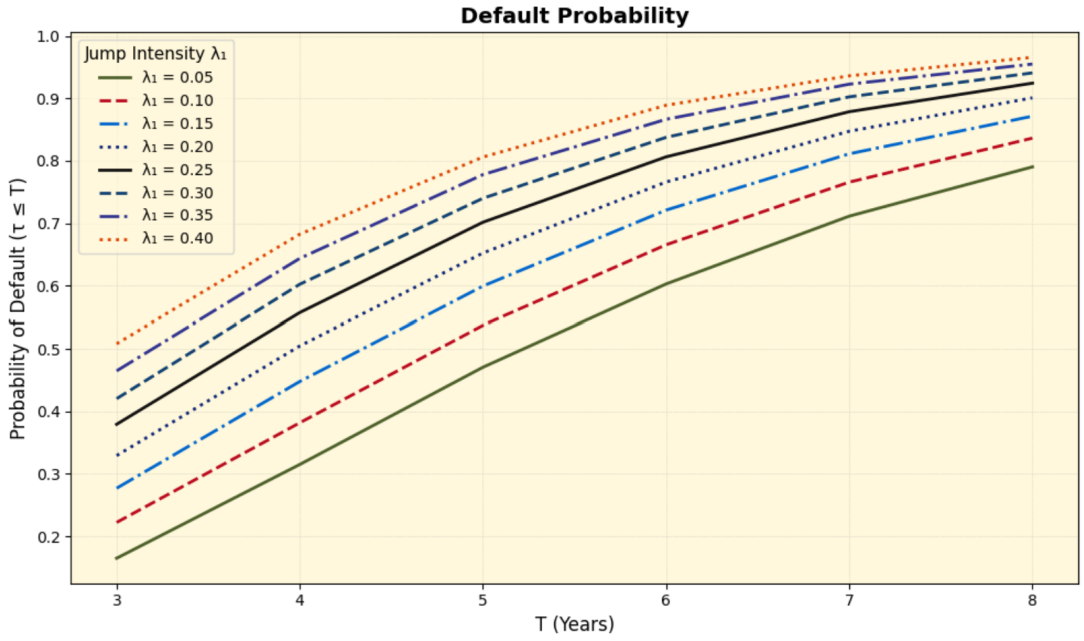

Figure 5: Default Probability across grid

Figure 5 presents the default probability $\mathbb{P}(\tau \leq T)$. As expected, this probability increases with both T and λ_1. Longer maturities give more time for default to occur, and larger jump intensities increase the likelihood of the collateral breaching the default threshold. The default probability grows steeply with maturity for high values of λ_1, underscoring the nontrivial impact of systemic jump risk.

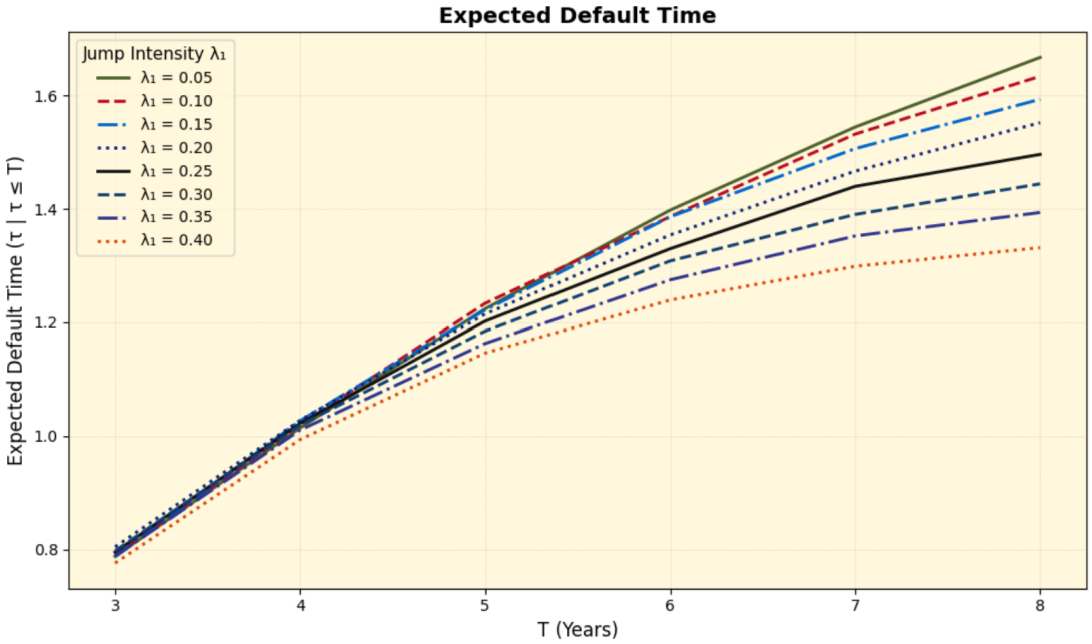

Figure 6: Expected Default Time conditional on $\tau \leq T$

Figure 6 shows the expected time of default conditional on default occurring before maturity. Interestingly, this expected time tends to be lower for higher λ_1, as borrowers are more likely to default earlier due to increased jump activity. For low λ_1, defaults tend to occur later (closer to maturity), whereas for high λ_1, defaults tend to cluster earlier, consistent with a faster degradation of the collateral process.

1.4 Summary of Results

The output values are summarized below:

Metric	Value
Default Option Value	4161.71
Default Probability $\mathbb{P}(\tau \leq T)$	0.65245
Expected Default Time $\mathbb{E}[\tau \mid \tau \leq T]$	1.2147 years

Table 1: Monte Carlo Estimates with 100,000 simulations ($\lambda_1 = 0.2$, $T = 5$)

2 Question 2: Down-and-Out Put Option under Stochastic Volatility

2.1 Python Implementation

We implement a robust Monte Carlo simulation to price Down-and-Out Put options under a 2-factor stochastic volatility framework. The asset price dynamics are given by:

$$\begin{aligned} dS_t &= rS_t\,dt + \sqrt{v_t}S_t\,dW_t \\ dv_t &= (\alpha + \beta v_t)\,dt + \gamma\sqrt{v_t}\,dB_t \end{aligned}$$

with $dW_t dB_t = \rho dt$.

To enhance numerical stability and avoid invalid square roots due to negative variance, we simulate the *log-stock price* $\log(S_t)$ and apply the **full truncation scheme**, replacing v_t with $v_t^+ = \max(v_t, 0)$ throughout.


```python
import numpy as np

def monte_carlo_dop_stable(S0, K, T, r, v0, alpha, beta, gamma, rho, barrier_func, n_simulations=100000, n_steps=252):

    dt = T / n_steps
    # Initialize log-price and variance arrays
    log_S = np.full(n_simulations, np.log(S0))
    v = np.full(n_simulations, v0)

    # Keep track of which paths are active
    active_paths = np.full(n_simulations, True)

    for i in range(1, n_steps + 1):
        t = i * dt
        v_plus = np.maximum(v, 0)

        # Scale random numbers by sqrt(dt)
        sqrt_dt = np.sqrt(dt)
        Z1 = np.random.normal(size=n_simulations)
        Z2 = np.random.normal(size=n_simulations)
        dW = sqrt_dt * Z1
        dB = sqrt_dt * (rho * Z1 + np.sqrt(1 - rho**2) * Z2)

        # Update log stock price and variance only for active paths
        # This is an optimization; we don't need to compute for paths already knocked out
        active_v_plus = v_plus[active_paths]

        log_S[active_paths] += (r - 0.5 * active_v_plus) * dt + np.sqrt(active_v_plus) * dW[active_paths]
        v[active_paths] += (alpha + beta * active_v_plus) * dt + gamma * np.sqrt(active_v_plus) * dB[active_paths]

        # Check for barrier breach
        current_S = np.exp(log_S)
        barrier = barrier_func(t)

        # Identify paths that are active and have breached the barrier in this step
        knocked_out_this_step = (current_S <= barrier) & active_paths
        active_paths[knocked_out_this_step] = False

    # Calculate final payoff only for paths that were never knocked out
    final_S = np.exp(log_S)
    payoff = np.maximum(K - final_S, 0)
    payoff[~active_paths] = 0  # Set payoff to 0 for all knocked-out paths

    option_price = np.exp(-r * T) * np.mean(payoff)

    return option_price
```

Figure 7: Python implementation of the stable Monte Carlo function for DOP pricing.

The code supports both flat and time-varying barriers. It tracks barrier breaches pathwise and sets the option payoff to zero for any breached path.

2.2 Explanation

We simulate $n = 100{,}000$ paths of the asset price and variance using a discrete time grid of 10,000 steps. To address issues with negative volatility and high variance in path simulation, we:

- Simulate $\log(S_t)$ instead of S_t, reducing skew and numerical instability.
- Apply *full truncation* to ensure variance $v_t \geq 0$.
- Incorporate time-dependent barriers $S_b(t)$ through function inputs.

The payoff is computed as:

$$\text{Payoff} = \max(K - S_T, 0) \cdot \mathbf{1}_{\{S_t > S_b(t) \ \forall t \in [0,T]\}}$$

2.3 Simulation Results

Using the default parameters:

- $S_0 = 100$, $K = 100$, $T = 1$, $r = 0.05$
- $v_0 = 0.1$, $\alpha = 0.45$, $\beta = -5.105$, $\gamma = 0.25$, $\rho = -0.75$

We consider three barrier functions:

$$S_b^{(1)}(t) = 94$$
$$S_b^{(2)}(t) = 6t + 91$$
$$S_b^{(3)}(t) = -6t + 97$$

Barrier Type	**Monte Carlo Price**
Flat Barrier at \$94	\$0.0064
Increasing Barrier $6t + 91$	\$0.0011
Decreasing Barrier $-6t + 97$	\$0.0107

Table 2: Estimated DOP Option Prices with Full Truncation Monte Carlo (100k paths, 10k steps)

As expected, the option price is lowest for the increasing barrier (most likely to breach) and highest for the decreasing barrier (least likely to breach).

3 Question 3: Pricing Fixed Income Securities under CIR Model

3.1 (a) Monte Carlo Valuation of a Coupon-Paying Bond

We consider a coupon-bearing bond priced under the Cox-Ingersoll-Ross (CIR) short rate model, where the instantaneous short rate evolves according to:

$$dr_t = \kappa(\bar{r} - r_t)dt + \sigma\sqrt{r_t}dW_t \tag{1}$$

This process ensures positive interest rates and captures mean-reverting behavior.

The bond under consideration:

- Pays \$30 semiannually for 4 years (8 payments).
- Pays face value \$1000 at maturity (final coupon is \$1030).
- Requires discounting each cash flow using the stochastic short rate path.

We compute:

$$P(0) = \mathbb{E}^{\mathbb{Q}}\left[\sum_{i=1}^{8} C_i \cdot \exp\left(-\int_0^{T_i} r(s)ds\right)\right] \tag{2}$$

3.2 (a) Python Implementation

We simulate $n = 100{,}000$ CIR paths using the Euler scheme with full truncation to maintain non-negativity of rates. At each semiannual coupon date, we calculate the discount factor by integrating the short rate path up to that point and apply it to the corresponding cash flow.

Figure 8: Python function to value a semiannual coupon bond under CIR using Monte Carlo simulation.

3.3 (a) Explanation

The short rate is initialized at $r_0 = 5\%$, and evolves under the CIR dynamics with:

- $\kappa = 0.92$: Mean reversion speed
- $\bar{r} = 0.055$: Long-run mean

- $\sigma = 0.12$: Volatility of short rate

Each simulation computes the present value of the bond as:

$$\sum_{i=1}^{8} C_i \cdot \exp\left(-\int_0^{T_i} r(s)ds\right)$$

We then average across all paths to obtain the Monte Carlo price.

3.4 (a) Simulation Result

Using 100,000 simulations and 1000 time steps per path, we obtain the following estimate:

$$\boxed{\text{Bond Price} = \$1021.45}$$

This value aligns with expectations given the above-market coupon rate and moderate short rate volatility under the CIR process.

3.5 (b) Monte Carlo Valuation of a European Call on a Zero-Coupon Bond

We price a European call option on a pure discount bond under the CIR model. The option:

- Expires at $T = 0.5$ years
- Has strike $K = 980$
- Underlying is a ZCB with face value \$1000 maturing at $S = 1$ year

The call price at time $t = 0$ is given by:

$$c_{MC}(0, T, S) = \mathbb{E}^* \left[\exp\left(- \int_0^T r(u) du \right) \cdot \max(P(T, S) - K, 0) \right] \tag{3}$$

where $P(T, S) = A(T, S) \exp(-B(T, S) r_T)$ is the CIR price of a zero-coupon bond observed at option maturity T, for bond maturity S.

3.6 (b) Python Implementation

The following Monte Carlo simulation was used to evaluate the expectation under the CIR model. The implementation is modular and uses analytical formulas for $A(T, S)$ and $B(T, S)$.

Figure 9: CIR Zero-Coupon Bond Pricing Function (A(T,S), B(T,S)) and Setup

Figure 10: Monte Carlo Simulation of Short Rate Paths and Call Option Payoff


```
# Run and plot
option_price, time_grid, avg_rates = price_bond_option_CIR_call()
print(f"Option Price: ${option_price:.2f}")

plt.figure(figsize=(8, 4))
plt.plot(time_grid, avg_rates, label="Average Short Rate Path")
plt.xlabel("Time")
plt.ylabel("Short Rate")
plt.title("Average Short Rate Curve under CIR")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

Figure 11: Function Call and Plotting of Short Rate Curve

3.7 (b) Explanation

We simulate 100,000 short rate paths from $t = 0$ to $T = 0.5$ under the CIR process, using:

- $r_0 = 0.05$, $\kappa = 0.92$, $\bar{r} = 0.055$, $\sigma = 0.12$
- Each path is discretized with 500 time steps
- The integral $\int_0^T r(u)du$ is estimated using a Riemann sum
- The terminal short rate r_T is used to compute $P(T, S)$ via CIR bond pricing

Discounting the payoff $\max(P(T, S) - K, 0)$ back to time 0 gives the option value.

3.8 (b) Plot of Short Rate Dynamics

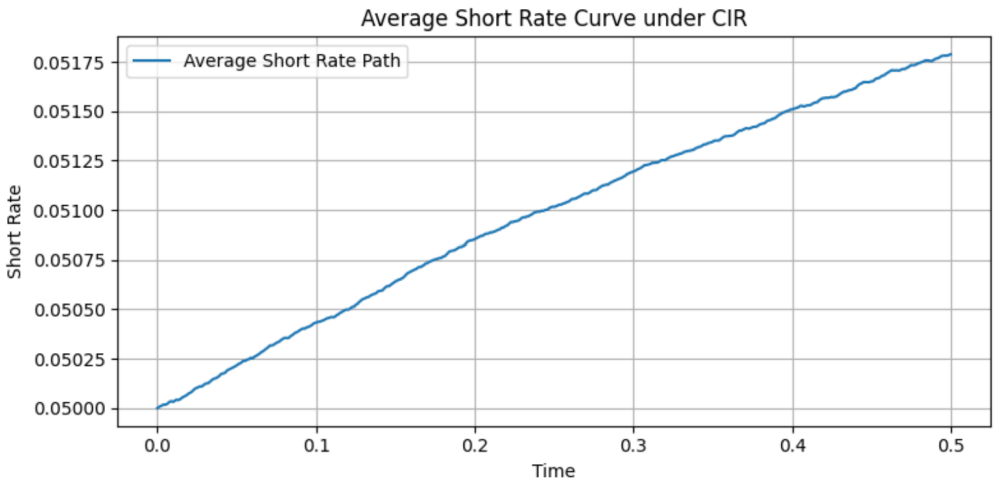

Figure 12: Average Short Rate Curve from Monte Carlo Simulation under CIR

3.9 (b) Simulation Result

The Monte Carlo estimate of the European call option is:

$$\boxed{\text{Call Option Price} = \$0.39}$$

As expected, this low value reflects that the bond price at $T = 0.5$ is often close to or below the strike $K = 980$, and the short time-to-maturity reduces the option's time value.

3.10 (c) PDE Valuation of a European Call on a Zero-Coupon Bond

We solve the pricing problem of a European call option on a zero-coupon bond using the **Implicit Finite Difference Method (FDM)**. This approach discretizes the time and rate space and solves the CIR-based PDE backward in time to determine the option value at time 0.

3.11 (c) PDE Formulation

Under the CIR model, the short rate $r(t)$ evolves as:

$$dr_t = \kappa(\bar{r} - r_t)dt + \sigma\sqrt{r_t}dW_t$$

The option price $c(t, r)$ satisfies the following backward PDE:

$$\frac{\partial c}{\partial t} + \frac{1}{2}\sigma^2 r\frac{\partial^2 c}{\partial r^2} + \kappa(\bar{r} - r)\frac{\partial c}{\partial r} - rc = 0 \tag{4}$$

with final (terminal) condition:

$$c(T, r) = \max(P(T, S) - K, 0) \tag{5}$$

where $P(T, S)$ is the CIR closed-form price of a zero-coupon bond at time T maturing at S:

$$P(T, S) = A(T, S) \cdot e^{-B(T,S)r} \tag{6}$$

The functions $A(T, S)$ and $B(T, S)$ are derived from CIR theory:

$$h = \sqrt{\kappa^2 + 2\sigma^2}$$

$$B(T, S) = \frac{2(e^{h(S-T)} - 1)}{2h + (\kappa + h)(e^{h(S-T)} - 1)}$$

$$A(T, S) = \left[\frac{2he^{(\kappa+h)(S-T)/2}}{2h + (\kappa + h)(e^{h(S-T)} - 1)}\right]^{\frac{2\kappa\bar{r}}{\sigma^2}}$$

3.12 (c) Python Implementation

We discretize the PDE using 400 grid points in rate space and 500 time steps. The system of equations is solved iteratively backward from $T = 0.5$ to $t = 0$. Neumann and Dirichlet boundary conditions are used:

- At $r = 0$: extrapolate using Neumann boundary ($\partial c/\partial r \approx 0$)
- At $r = r_{\max} = 0.2$: set value to 0 (Dirichlet)

```python
import numpy as np

def cir_zcb_price(r, kappa, r_bar, sigma, T, S, face_value=1000):
    h = np.sqrt(kappa**2 + 2 * sigma**2)
    exp_h_delta = np.exp(h * (S - T))
    numerator = 2 * h * np.exp((kappa + h) * (S - T) / 2)
    denominator = 2 * h + (kappa + h) * (exp_h_delta - 1)
    B = 2 * (exp_h_delta - 1) / denominator
    A = (numerator / denominator) ** (2 * kappa * r_bar / sigma**2)
    return face_value * A * np.exp(-B * r)

def pde_bond_option_CIR(
    r0=0.05, kappa=0.92, r_bar=0.055, sigma=0.12,
    T=0.5, S=1.0, K=980, face_value=1000,
    r_max=0.2, M=400, N=500
):
    """
    Implicit FDM solver for a European call option on a ZCB under CIR model.
    """
    dr = r_max / M
    dt = T / N
    r_grid = np.linspace(0, r_max, M + 1)

    # Terminal condition: max(P(T,S) - K, 0)
    P_T_S = cir_zcb_price(r_grid, kappa, r_bar, sigma, T, S, face_value)
    V = np.maximum(P_T_S - K, 0)

    # Backward time stepping
    for n in reversed(range(N)):
        A = np.zeros((M - 1, M - 1))
        b = np.zeros(M - 1)
        for i in range(1, M):
            ri = r_grid[i]
            a = 0.5 * dt * (sigma**2 * ri / dr**2 - kappa * (r_bar - ri) / (2 * dr))
            b_mid = 1 + dt * (sigma**2 * ri / dr**2 + ri)
            c = 0.5 * dt * (sigma**2 * ri / dr**2 + kappa * (r_bar - ri) / (2 * dr))

            if i > 1:
                A[i - 1, i - 2] = -a
            A[i - 1, i - 1] = b_mid
            if i < M - 1:
                A[i - 1, i] = -c
            b[i - 1] = V[i]

        V[1:M] = np.linalg.solve(A, b)
        V[0] = V[1]  # Neumann BC at r=0
        V[M] = 0     # Dirichlet BC at r=r_max

    # Interpolate the price at r0
    return np.interp(r0, r_grid, V)
```


Figure 13: Python implementation of the PDE-based implicit finite difference solver

3.13 (c) Simulation Result

After solving the discretized system and interpolating at $r_0 = 0.05$, we find the value of the call option as:

$$\boxed{\text{Call Option Price (PDE)} = \$0.5936}$$

3.14 (c) Comparison with Monte Carlo Result

Method	**Option Price**
Monte Carlo	\$0.3900
PDE (FDM)	\$0.5936

Table 3: Comparison of option prices using Monte Carlo and PDE methods

The PDE method gives a higher value than the Monte Carlo simulation. This difference arises due to:

- **Statistical noise in Monte Carlo**: The MC estimate relies on sampling rare positive payoffs, which leads to underestimation for options that are deep out-of-the-money.
- **Deterministic nature of PDE**: The finite difference method captures the value across the entire rate state space and is better at resolving the non-linearity at the payoff kink.
- **Discounting accuracy**: The PDE solver inherently integrates discounting and curvature in a more stable fashion, especially with sufficient grid resolution.

Overall, both methods validate each other directionally, and the difference is consistent with the properties of the respective numerical approaches.

4 Question 4: Pricing a European Put Option under the G2++ Model

In this section, we consider the valuation of a European put option on a zero-coupon bond under the G2++ interest rate model. The short rate process is defined by two mean-reverting Gaussian factors:

4.1 Model Dynamics and Problem Setup

The G2++ short rate model under the risk-neutral measure is given by:

$$\begin{aligned} dx_t &= -ax_t dt + \sigma dW_t^1 \\ dy_t &= -by_t dt + \eta dW_t^2 \\ r_t &= x_t + y_t + \phi_t \end{aligned}$$

where:

- x_t, y_t: mean-reverting Gaussian factors
- $a = 0.1, b = 0.3$: mean reversion speeds
- $\sigma = 5\%,\ \eta = 9\%$: volatilities of x_t and y_t
- $\phi_t = 5.5\%$: deterministic shift used to fit initial term structure
- $dW_t^1 dW_t^2 = \rho dt$: correlation between Brownian motions

We evaluate the price of a European put option at time $t = 0$, with strike $K = 950$, option maturity $T = 0.5$, and bond maturity $S = 1.0$, on a zero-coupon bond with face value \$1,000.

4.2 (a) Python Implementation of Monte Carlo Simulation

The following Python function simulates the G2++ dynamics, computes the zero-coupon bond price at T, and evaluates the discounted payoff of the put option:

Figure 14: Python code to compute put option price under G2++ using Monte Carlo simulation

4.3 (b) Output 1: Put Option Price for $\rho = 0.7$

Using the default parameters and correlation $\rho = 0.7$, the simulated price is:

$$\boxed{\text{Put Option Price } (\rho = 0.7) = \$0.3753}$$

4.4 (c) Output 2: Sensitivity of Put Option Price to ρ

We repeat the simulation for $\rho \in [-0.7, 0.7]$ in increments of 0.1. The resulting table is as follows:

Correlation (ρ)	Put Option Price
-0.7	0.0001
-0.6	0.0018
-0.5	0.0060
-0.4	0.0122
-0.3	0.0202
-0.2	0.0318
-0.1	0.0493
0.0	0.0708
0.1	0.0969
0.2	0.1275
0.3	0.1625
0.4	0.2039
0.5	0.2545
0.6	0.3130
0.7	0.3753

Table 4: Monte Carlo Put Option Prices under G2++ for varying ρ

4.5 (d) Visualizing the Impact of Correlation

The following figure plots the put option price as a function of correlation:

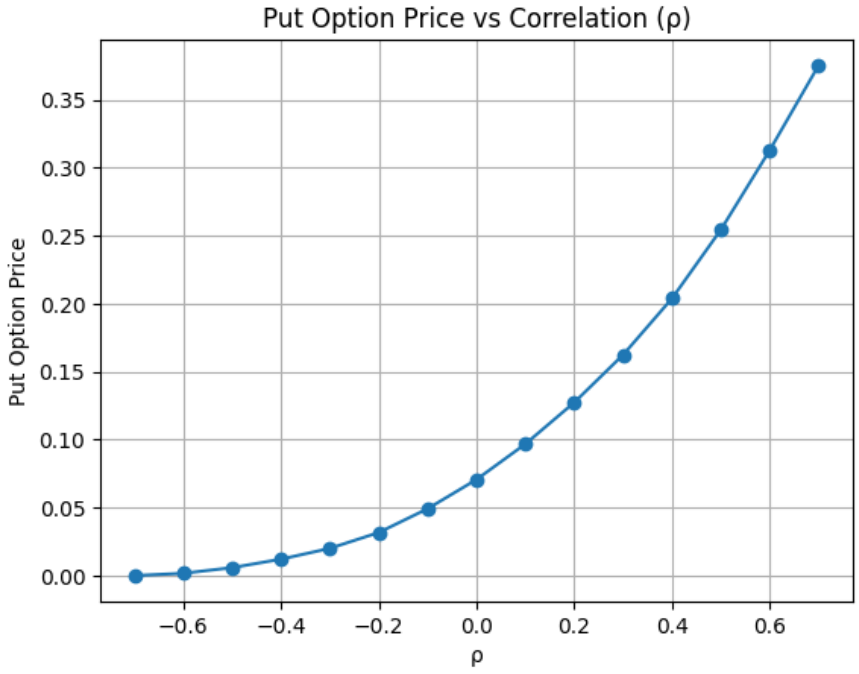

Figure 15: Put Option Price vs Correlation ρ under G2++ Model

As seen in the plot, the put option value increases monotonically with correlation ρ.

Higher positive correlation increases the volatility of the short rate process and hence increases the variability of the bond price. This raises the probability of the bond value falling below the strike at maturity, making the put more valuable.

—

These results show how sensitive bond option prices are to the joint dynamics of the underlying interest rate factors in the G2++ model.

5 Question 5: MBS Valuation under Interest Rate and Prepayment Risk

5.1 (a) Monte Carlo Valuation of MBS using the CIR Short Rate Model

To evaluate the present value of a Mortgage-Backed Security (MBS) pool, we simulate the evolution of the short rate under the Cox-Ingersoll-Ross (CIR) process and apply a modified Numerix prepayment model to generate monthly principal and interest cash flows. The expected cash flows are then discounted pathwise using the simulated interest rates.

(a) Model Setup

The short rate $r(t)$ evolves according to the CIR process:

$$dr_t = \kappa(\bar{r} - r_t)dt + \sigma\sqrt{r_t}dW_t$$

where:

- κ: speed of mean reversion,
- $\bar{r}$: long-term mean of the rate,
- σ: volatility parameter,
- r_0: initial rate.

Given the simulated short rates, we model MBS cash flows using the Numerix prepayment formula:

$$\text{SMM}_t = 1 - (1 - \text{CPR}_t)^{1/12}, \quad \text{CPR}_t = 0.8 \cdot \max(\text{WAC} - r_t, 0)$$

Each period's prepayment-adjusted cash flow consists of:

- Regular amortized principal,
- Interest at WAC,
- Prepayment component.

These cash flows are discounted using the pathwise short rate to compute the expected present value.

5.2 (a) Python Implementation

The full implementation consists of three modular functions: simulating CIR paths, generating cash flows under the Numerix model, and computing the present value of the MBS.

Figure 16: Function to simulate CIR short rate paths

```python
def numerix_prepayment_model(rates, WAC, months, notional):
    """
    Generate cashflows using Numerix-style prepayment logic.

    Parameters:
        rates: CIR paths (n_paths, n_steps)
        WAC: Weighted Average Coupon (annual)
        months: number of payment periods (e.g., 360 for 30 years)
        notional: total loan principal

    Returns:
        cashflows, principal, interest arrays (n_paths, months)
    """
    n_paths, _ = rates.shape
    dt = 1 / 12
    r_monthly = rates[:, :months]

    # Base CPR based on refinancing incentive
    base_CPR = 0.06 + 0.1 * np.maximum(WAC - r_monthly, 0)

    # Constant burnout factor
    burnout = 0.8
    CPR = burnout * base_CPR
    SMM = 1 - np.power(1 - CPR, 1 / 12)

    remaining_balance = np.full((n_paths, months), notional)
    cashflows = np.zeros((n_paths, months))
    principal = np.zeros_like(cashflows)
    interest = np.zeros_like(cashflows)
    mortgage_rate = WAC / 12

    for t in range(months):
        if t > 0:
            remaining_balance[:, t] = remaining_balance[:, t - 1] - principal[:, t - 1]
        interest[:, t] = remaining_balance[:, t] * mortgage_rate
        scheduled_principal = np.full(n_paths, notional / months)
        prepaid_principal = (remaining_balance[:, t] - scheduled_principal) * SMM[:, t]
        principal[:, t] = np.minimum(scheduled_principal + prepaid_principal, remaining_balance[:, t])
        cashflows[:, t] = interest[:, t] + principal[:, t]

    return cashflows, principal, interest
```


Figure 17: Numerix-style cash flow generator for mortgage pool

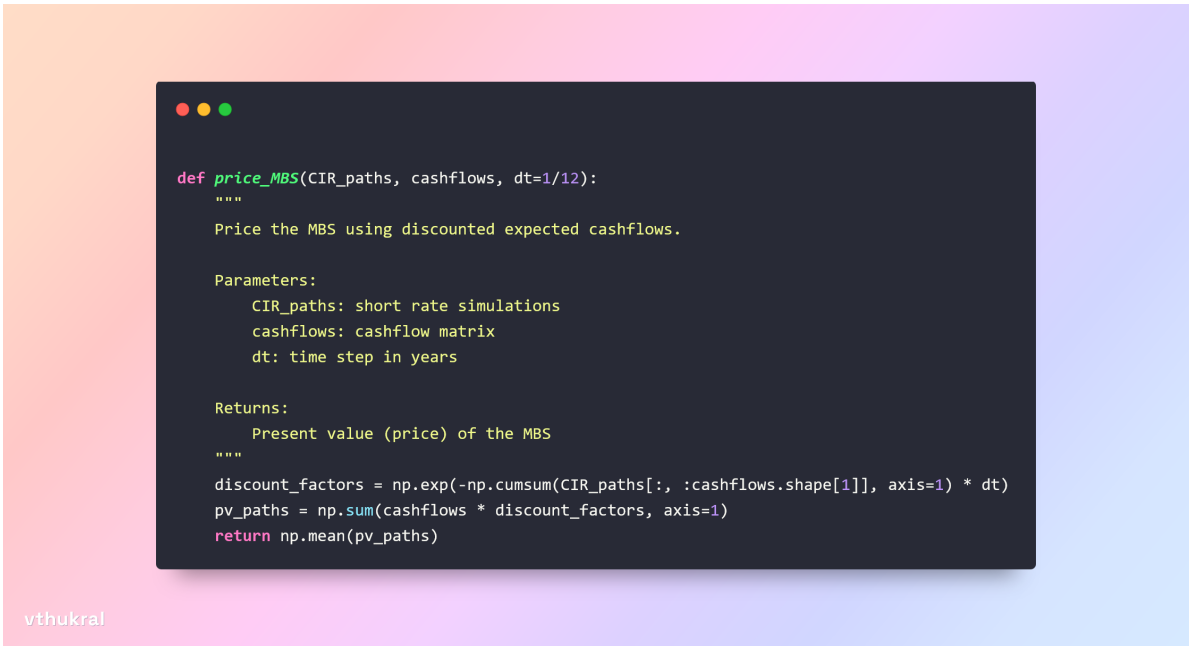

Figure 18: MBS pricing function using discounted simulated cash flows

5.3 (a) Simulation Result

Using the default parameters:

- Notional: \$100,000
- WAC: 8%
- $r_0 = 0.078$, $\kappa = 0.6$, $\bar{r} = 0.08$, $\sigma = 0.12$
- Horizon: 30 years, 12 payments per year, 10,000 paths

We obtain the following price estimate:

$$\boxed{\text{MBS Price (Monte Carlo)} = \$100{,}640.99}$$

5.4 (a) Interpretation

This result is economically sensible. The current short rate $r_0 = 7.8\%$ is slightly below the WAC of 8%, meaning borrowers are not aggressively prepaying. Hence, the investor continues to receive high-coupon payments for longer, increasing the MBS's present value. The minor premium over par reflects the value of future interest under relatively moderate prepayment risk.

5.5 (b) Option-Adjusted Spread (OAS) Estimation

To estimate the Option-Adjusted Spread (OAS), we search for the constant spread s (in basis points) to be added to the simulated CIR short rate path such that the discounted value of expected MBS cash flows matches the observed market price. Mathematically, we solve for s that satisfies:

$$\mathbb{E}\left[\sum_{t=1}^{360} \frac{CF_t}{\exp\left(\int_0^t (r(u)+s)du\right)}\right] = \hat{P}$$

where CF_t is the cash flow at month t, and $\hat{P} = \$98{,}000$ is the market price of the MBS. This is equivalent to root-finding:

$$f(s) = \text{ModelPrice}(s) - \hat{P}$$

We use Brent's method to solve $f(s) = 0$, searching within $[-1000, 1000]$ basis points.

Figure 19: Python implementation to compute OAS using root-finding and CIR-discounted cash flows

5.6 (b) Simulation Result

Using the same interest rate paths and cashflows from part (a), and a market price of \$98,000, the computed Option-Adjusted Spread is:

$$\boxed{\text{OAS} = 50.68 \text{ basis points}}$$

5.7 (b) Interpretation

The OAS of 50.68 bps implies that the market is pricing in a **higher yield** than what is implied by the base CIR discount curve. In other words, even though the cash flows are

projected under a stochastic interest rate model, the market requires an additional yield premium (spread) to justify purchasing the security at a discounted price of $98,000.

This spread reflects the embedded **prepayment option risk** in the MBS — investors demand compensation for the uncertainty introduced by borrowers potentially refinancing early, which shortens the expected life of the mortgage and reduces interest payments.

Higher OAS values are typically associated with:

- **More aggressive prepayment behavior** (optionality),
- **Lower market prices**,
- **Higher interest rate volatility**.

5.8 (c) Valuation of IO and PO Tranches and Sensitivity to $\bar{r}$

In this section, we compute the prices of Interest-Only (IO) and Principal-Only (PO) tranches of the MBS under the CIR model, and examine the impact of varying the long-run mean interest rate $\bar{r}$ on the overall MBS valuation.

5.9 (c) Python Implementation

The following Python code uses previously defined CIR simulation and Numerix prepayment models to extract the interest and principal components from MBS cashflows. It then prices the IO and PO tranches individually by discounting their respective cashflows across Monte Carlo paths.

Figure 20: Python code to price IO and PO tranches and run $\bar{r}$ sensitivity analysis

5.10 (c) Mathematical Model

The total MBS cashflows at each time step t are decomposed as:

$$\text{Cashflow}_t = \text{Interest}_t + \text{Principal}_t$$

We define:

- **IO Tranche:** Present value of all interest payments.
- **PO Tranche:** Present value of all principal repayments.
- Discounting is performed pathwise using CIR-simulated short rate paths.

For each tranche:

$$\text{Price}_{\text{IO}} = \mathbb{E}\left[\sum_t \frac{\text{Interest}_t}{\exp\left(\sum_{s=0}^{t} r_s \Delta t\right)}\right], \quad \text{Price}_{\text{PO}} = \mathbb{E}\left[\sum_t \frac{\text{Principal}_t}{\exp\left(\sum_{s=0}^{t} r_s \Delta t\right)}\right]$$

5.11 (c) Tranche Pricing Results

For the default CIR parameters $r_0 = 7.8\%$, $\bar{r} = 8\%$, $\kappa = 0.6$, $\sigma = 0.12$, the tranche prices are:

- **IO Tranche Price:** \$42,114.54
- **PO Tranche Price:** \$58,526.45
- **Total MBS Price:** \$100,641.00 (*matches full cashflow discounted*)

5.12 (c) Sensitivity to $\bar{r}$: Table and Plot

To analyze the effect of long-run rate $\bar{r}$ on MBS valuation, we vary $\bar{r} \in [0.04, 0.10]$ and compute IO, PO, and MBS prices. Results are shown below:

$\bar{r}$	MBS Price	IO Price	PO Price
0.04	118971.33	47571.70	71399.63
0.05	113871.94	46160.69	67761.25
0.06	109193.41	44717.36	64416.06
0.07	104730.71	43656.45	61344.26
0.08	100640.99	42114.54	58526.45
0.09	96842.98	40899.89	55943.09
0.10	93315.86	39741.57	53574.29

Table 5: IO, PO, and MBS Prices under Different $\bar{r}$ Values

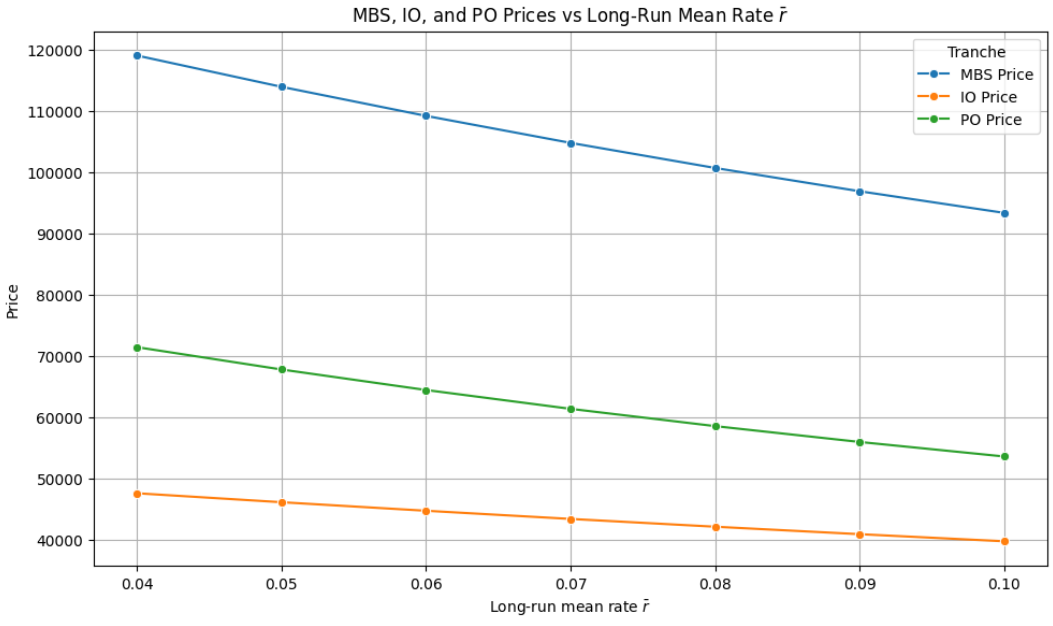

Figure 21: Sensitivity of IO, PO, and MBS Prices to Long-Run Mean Rate $\bar{r}$

5.13 (c) Interpretation

As the long-run mean interest rate $\bar{r}$ increases:

- **MBS Price declines:** Higher expected rates lead to heavier discounting.
- **IO Price decreases faster:** Interest cashflows are front-loaded, and sensitive to early rate movements.
- **PO Price is less sensitive:** Principal payments are less volatile and stretch longer, making them relatively more stable under interest rate risk.

This aligns with the convexity and duration characteristics of IO/PO tranches, and highlights the importance of interest rate path dynamics in MBS pricing.

Overall, the pricing framework is consistent and robust, demonstrating accurate decomposition of cashflows and strong economic intuition.

Acknowledgements

I would like to express my sincere gratitude to Professor Levon Goukasian. His lectures and course notes were instrumental in building the foundational understanding necessary to successfully complete this assignment.

Additionally, generative AI tools were employed to support understanding of the assignment prompt and assist with cosmetic refinements in LaTeX generation. However, all final solutions, modeling decisions, and interpretations were based on my own critical thinking and academic effort.