MFE 405: Project 3

Aliaksei Kanstantsinau

In [125…
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

Problem 1:

Value of collateral follows a jump diffusion process: $$ \frac{dV_t}{V_{t}^{-}} = \mu dt + \sigma dW_t + \gamma dJ_t$$ where J is a Poisson process with intensity λ_1, independent of the Brownian motion.

Consider a collateralized loan, with a contract rate per period r and maturity T on the above-collateral, and assume the outstanding balance of that loan follows: $$L_t = a - bc^{12t}$$

(a) Estimate the value of the default option for the following ranges of parameters: λ_1 from .05 to .4 in increments of .05; T from 3 to 8 in increments of 1.

(b) Estimate the default probability for the following ranges of parameters: λ1 from 0.05 to 0.4 in increments of 0.05; T from 3 to 8 in increments of 1;

(c) Find the Expected option Exercise Time of the default option, conditional on $\tau < T$. That is, estimate $E(\tau \mid \tau < T)$ for the following ranges of parameters: λ1 from 0.05 to 0.4 in increments of 0.05; T from 3 to 8 in increments of 1;

In [126…
```
# define problem 1 function
def problem1a(seed, lambda1, T):
    np.random.seed(seed)
    # set the given parameters excluding lambda 1 and T
    V0 = 20000
    L0 = 22000
    mu = -.1
    sigma = .2
    gamma = -.4
    r0 = .055
    delta = .25
    lambda2 = .4
    alpha = .7
    epsilon = .95
    beta = (epsilon - alpha)/T
    r = (r0 + delta * 0.4) / 12
    n = T*12
    PMT = L0*r / (1 - 1/(1 + r)**n)
    a = PMT/r
    b = PMT/(r*(1 + r)**n)
    c = 1 + r
```

```python
    simulations = 10000
    steps = 100
    dt = T/steps

    # define jump diffusion function
    def jump_diffusion():
        V = np.zeros(steps + 1)    # V process -> zeros
        V[0] = V0    # set the initial point
        jump_times = [] # initiate jumps array to store jump times
        jump_time = 0   # set j to zero
        while jump_time < T:    # realization of the jump not in [0, T]
            Y = np.random.exponential(lambda1 * T)  # E(Y) = 1/(lambda * T)
            jump_time += Y  # handle j's
            if jump_time < T:
                jump_times.append(jump_time)

        for j in range(1, steps + 1):  # get time steps
            t = j * dt  # get current time t
            dW = np.random.normal(0, np.sqrt(dt))   # simulate brownian moti
            dJ = np.random.poisson(lambda1 * T)
            V[j] = V[j-1] + (mu*dt + sigma*dW + gamma*dJ)
            if any(abs(t - jump_time) < dt/2 for jump_time in jump_times):
                V[j] *= (1 + gamma)
        return V

    def loan_balance(t):
        return a - b*c**(12*t)

    # def tau (stop time)
    def tao(V, L):
        for t in range(len(V)):     # get qt for each time
            q_t = alpha + beta * t * dt
            if V[t] <= q_t * L[t]:
                return t * dt, V[t], L[t]
        return T, V[-1], L[-1]    # Get  T, V, L stop times

    payoffs = []
    default_option_values = []
    default_times = []
    default_flags = []

    # simulate and get values
    for simulation in range(simulations):
        V = jump_diffusion()    # V
        L = [loan_balance(t) for t in np.linspace(0, T, steps + 1)] # L
        tau_1, V_tau, L_tau = tao(V, L)   # stopping times

        S = np.random.exponential(1/(lambda2 * T))  # simulate S
        tau_2 = min(tau_1, S)

        # if tau > T, there is no defualt option exercise
        if tau_1 >= T:
            payoff = 0
            default_flag = False
        else:
            payoff = max(L_tau - epsilon * V_tau, 0)
```

```
            default_flag = True

        discounted_payoff = payoff/(1+r0)**tau_2
        default_option_values.append(discounted_payoff)
        default_times.append(tau_2)
        default_flags.append(default_flag)
        payoffs.append(discounted_payoff)

    D = np.mean(default_option_values)
    Prob = np.mean(default_flags)
    Et = np.mean([t for t, flag in zip(default_times, default_flags) if flag

    return D, Prob, Et
```

In [127…

```
problem_1a = problem1a(seed=1234, lambda1=.2, T=5)
problem_1a
```

Out[127…

```
(7315.746336788343, 0.9316, 0.33341976008202695)
```

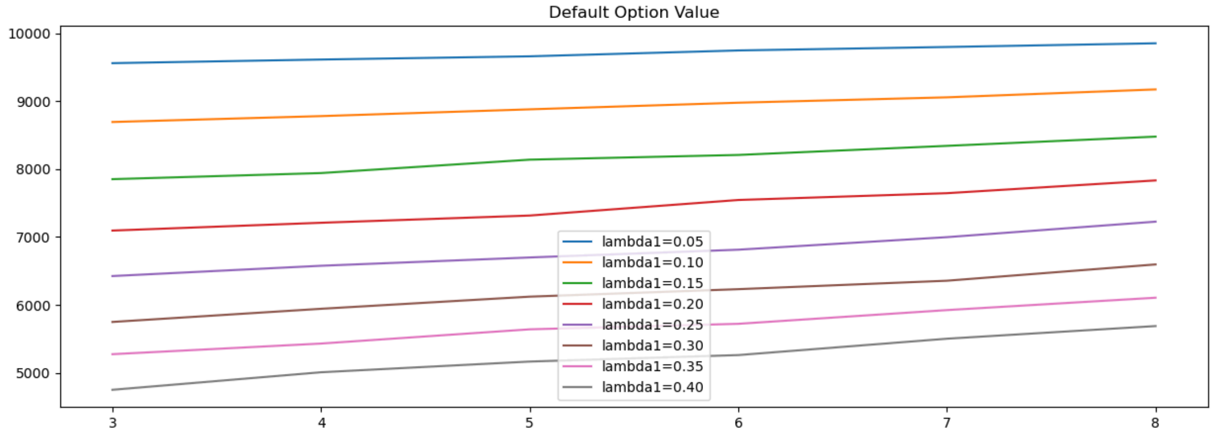
In [128…

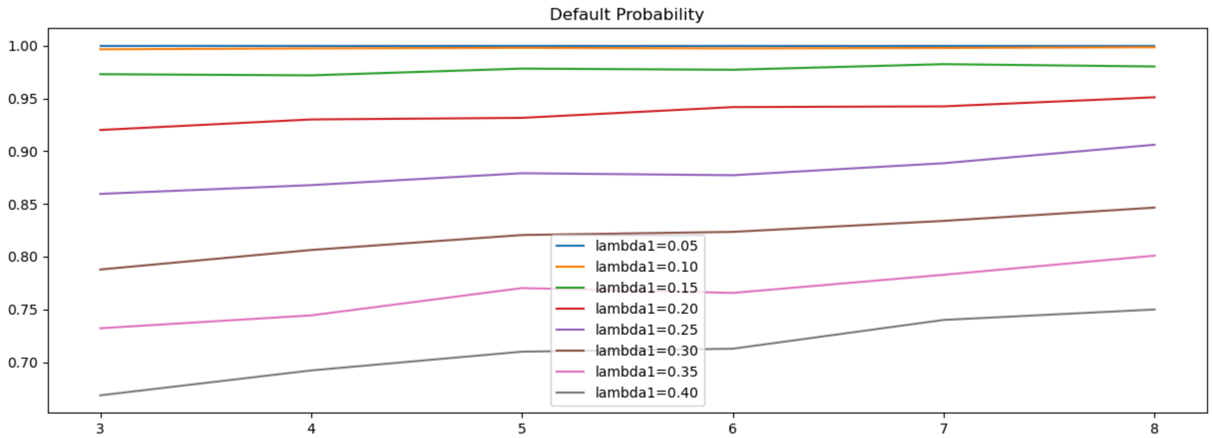
```
#   set ranges
lambda1_range = np.arange(0.05, 0.45, 0.05)
T_range = np.arange(3, 9, 1)
```

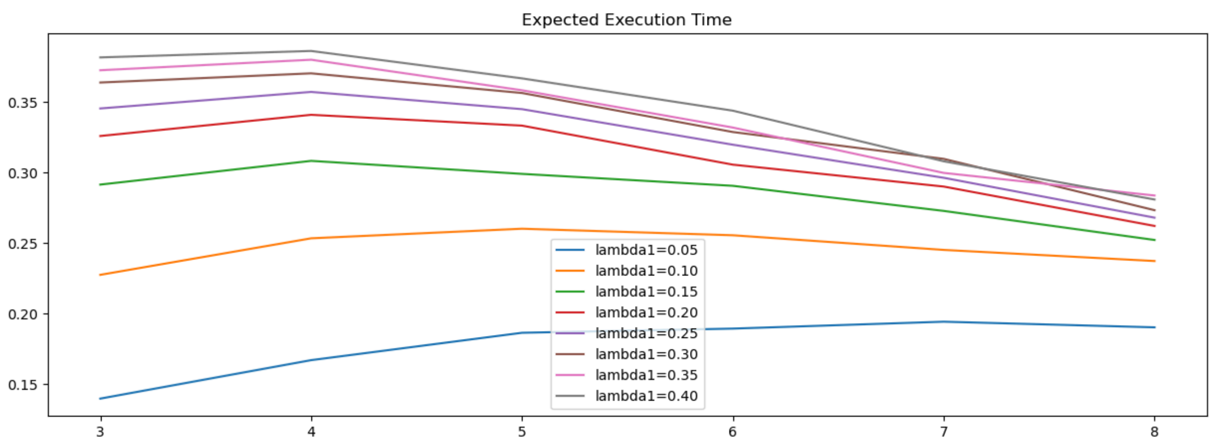
In [129…

```
# get visual on value difference
default_values = {}
default_probabilities = {}
expected_times = {}

for lambda1 in lambda1_range:
    default_values[lambda1] = []
    default_probabilities[lambda1] = []
    expected_times[lambda1] = []
    for T in T_range:
        D, Prob, Et = problem1a(seed=1234, lambda1=lambda1, T=T)
        default_values[lambda1].append(D)
        default_probabilities[lambda1].append(Prob)
        expected_times[lambda1].append(Et)

plt.figure(figsize=(15, 5))
for lambda1 in lambda1_range:
    plt.plot(T_range, default_values[lambda1], label=f'lambda1={lambda1:.2f}
plt.title('Default Option Value')
plt.legend()
plt.show()

plt.figure(figsize=(15, 5))
for lambda1 in lambda1_range:
    plt.plot(T_range, default_probabilities[lambda1], label=f'lambda1={lambc
plt.title('Default Probability')
plt.legend()
plt.show()

# Plot Expected Default Times
plt.figure(figsize=(15, 5))
```

```python
for lambda1 in lambda1_range:
    plt.plot(T_range, expected_times[lambda1], label=f'lambda1={lambda1:.2f}
plt.title('Expected Execution Time')
plt.legend()
plt.show()
```







Problem 2.

Consider the following 2-factor model for a stock price process, under the risk-neutral measure:

$dSt = rStdt + \sqrt{vt}\ St\ dWt$
$dvt = (\alpha + \beta vt)dt + \gamma\sqrt{vt}\ dBt$

where Wt and Bt are correlated Brownian Motion processes with $dWtdBt = \rho dt$. Default parameter values: $v0$ = 0.1, α = 0.45, β = −5.105, γ = 0.25, $S0$ = \$100, r = 0.05, ρ = −0.75, K = \$100, T = 1.

(a) Estimate the Price ($P1$) of a Down-and-Out Put option with the barrier at $S_{b}^{1}
(b) = 94$.

(b) Estimate the Price ($P2$) of a Down-and-Out Put option with time-dependent barrier $S_{b}^{2}(t) = \frac{6}{T}t + 91$.

(c) Estimate the Price ($P3$) of a Down-and-Out Put option with time-dependent barrier $S_{b}^{3}(t) = -\frac{6}{T}t + 97$

In [130…

```
# gamma .25, K = 100, T = 1
def problem2(K, T, gamma):
    # define parameters
    V0 = .1
    alpha = .45
    beta = -5.105
    S0 = 100
    r = .05
    rho = -.75
    T = 1
    simulations = 10000
    steps = 100
    dt = T/steps

    c_matrix = np.array([[1, rho], [rho, 1]])
    L = np.linalg.cholesky(c_matrix)    # cholesky decomposition

    def simulate_paths():
        S_paths = np.zeros((simulations, steps + 1))
        V_paths = np.zeros((simulations, steps + 1))

        S_paths[:, 0] = S0  # set initial value
        V_paths[:, 0] = V0

        for t in range(1, steps + 1):
            W = np.random.normal(size=(simulations, 2)) #BM
            dW_dB = np.dot(W, L.T) * np.sqrt(dt)
            dW = dW_dB[:, 0]
            dB = dW_dB[:, 1]

            # simulate paths S and V
            S_paths[:, t] = S_paths[:, t-1] + r * S_paths[:, t-1]*dt + np.sc
            V_paths[:, t] = V_paths[:, t-1] + (alpha + beta *V_paths[:, t-1]
        return S_paths, V_paths

    S_paths, V_paths = simulate_paths()

    ko1 = np.any(S_paths <= 94, axis=1)
    payoffs1 = np.maximum(K - S_paths[:, -1], 0)
    payoffs1[ko1] = 0
```

```
    discounted_payoffs1 = np.exp(-r * T) * payoffs1
    option_price1 = np.mean(discounted_payoffs1)

    ko2 = np.any(S_paths <= 6/T + 91, axis=1)
    payoffs2 = np.maximum(K - S_paths[:, -1], 0)
    payoffs2[ko2] = 0
    discounted_payoffs2 = np.exp(-r * T) * payoffs2
    option_price2 = np.mean(discounted_payoffs2)

    ko3 = np.any(S_paths <= -(6/T) + 97, axis=1)
    payoffs3 = np.maximum(K - S_paths[:, -1], 0)
    payoffs3[ko3] = 0
    discounted_payoffs3 = np.exp(-r * T) * payoffs3
    option_price3 = np.mean(discounted_payoffs3)

    return option_price1, option_price2, option_price3
```

In [131…

```
problem_2 = problem2(100, 1, .25)
problem_2
```

Out[131…

```
(0.013579186929673585, 0.001239224338542639, 0.05208453722676879)
```

Write-Up:

Intuition-wise the expectation is that the option price will be low, however, not that low perhaps. (Not sure if I got the correct answer)

Based on different barriers it seems like (c) is very close to be the average (a) and (b)

Problem 3

Assume the dynamics of the short-term interest rate, under the risk-neutral measure, are given by the following SDE (CIR model):

$$drt = \kappa(\bar{r} - rt)dt + \sigma\sqrt{rt}dWt$$

with $r0$ = 5%, σ = 12%, κ = 0.92, $\bar{r}$ = 5.5%.

(a) Use Monte Carlo Simulation to find the price of a coupon-paying bond, with Face Value of $1,000, paying semiannual coupons of $30, maturing in T = 4 years: ... where C = {Ci = $30 for i = 1,2, ... ,7; and $C8$ = $1,030}, and T = {$T1$, $T2$, $T3$, $T4$, $T5$, $T6$, $T7$, $T8$} = {0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4}.

In [132…

```
def problem3a(r0, sigma, kappa, r_bar):
    # T = 4, dt = 1 trading day, N - total time steps, use 10000 simulations
    T = 4
    steps = 100
    dt = T/steps
    N = int(T/dt)
    simulations = 10000
```

```
    # initialize simulated interest rate storage
    ir_paths = np.zeros((simulations, N))
    ir_paths[:, 0] = r0

    # simulate paths
    for t in range(1, N):
        dW = np.random.normal(0, 1) * np.sqrt(dt)   # simulate brownian moti
        ir_paths[:, t] = ir_paths[:, t-1] + kappa*(r_bar - ir_paths[:, t-1])

    payments = np.array([0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4])   # payment dates
    cfs = np.array([30, 30, 30, 30, 30, 30, 30, 1030])  # cash flows

    simulated_bonds = np.zeros(simulations)    # store simulated bond prices
    for i in range(simulations):
        df = np.exp(-np.cumsum(ir_paths[i, ::int(N/7)]) * payments * dt)
        simulated_bonds[i] = np.sum(df * cfs)  # get price of bond
    bond_price = np.mean(simulated_bonds)
    return bond_price
```

In [133…
```
problem_3a = problem3a(.05, .12, .92, .055)
problem_3a
```

Out[133… 1166.473794432001

(b) Use Monte Carlo Simulation to find at time $t = 0$ the price $cMC(t, T, S)$ of a European Call option, with strike price of $K = \$980$ and expiration in $T = 0.5$ years on a Pure Discount Bond that has Face Value of $1,000 and matures in $S = 1$ year:

In [134…
```
def problem3b(r0, sigma, kappa, r_bar, T, S):
    steps = 100
    dt_T = T/steps
    dt_S = S/steps
    N_T = int(T/dt_T)
    N_S = int(S/dt_S)
    simulations = 10000
    K = 980

    # initialize simulated interest rate storage
    ir_paths_T = np.zeros((simulations, N_T)) # time T
    ir_paths_S = np.zeros((simulations, N_S)) # time S
    ir_paths_T[:, 0] = r0
    ir_paths_S[:, 0] = r0

    # simulate paths
    for t in range(1, N_T):
        dW = np.random.normal(0, 1) * np.sqrt(dt_T)  # simulate brownian mot
        ir_paths_T[:, t] = ir_paths_T[:, t-1] + kappa*(r_bar - ir_paths_T[:,

    for t in range(1, N_S):
        dW = np.random.normal(0, 1) * np.sqrt(dt_S)   # simulate brownian mo
        ir_paths_S[:, t] = ir_paths_S[:, t-1] + kappa*(r_bar - ir_paths_S[:,

    # calculate bond price at time S and T
    P_T = np.exp(-np.cumsum(ir_paths_T, axis=1)*dt_T)[:, -1]
```

```
    P_S = np.exp(-np.cumsum(ir_paths_S, axis=1)*dt_S)[:, -1]

    payoff = np.maximum(P_S * 1000 - K, 0) * np.exp(-np.sum(ir_paths_T, axis
    option_price = np.mean(payoff)
    return option_price
```

In [135…

```
problem_3b = problem3b(.05, .12, .92, .055, .5, 1)
problem_3b
```

Out[135… 0.0

(c) Use the Implicit Finite-Difference Method to find at time $t = 0$ the price $cPDE(t, T, S)$ of a European Call option, with strike price of $K = \$980$ and expiration in $T = 0.5$ years on a Pure Discount Bond that has Face Value of $1,000 and matures in $S = 1$ year. The PDE is given as follows ...

In [136…

```
import scipy.linalg
```

In [137…

```
def problem3c(r0, sigma, kappa, r_bar, T, S):
    K = 980
    F = 1000
    r_max = r0
    Nr = 100
    Nt = 100

    dt = T / Nt
    dr = r_max / Nr
    r = np.linspace(0, r_max, Nr + 1)
    t = np.linspace(0, T, Nt + 1)

    c = np.zeros((Nr + 1, Nt + 1))

    P_S = F * np.exp(-r * S)
    c[:, -1] = np.maximum(P_S - K, 0)

    alpha = 0.5 * sigma**2 * r * dt / dr**2
    beta = (kappa * (r_bar - r) * dt) / (2 * dr)
    gamma = r * dt

    A = np.zeros((Nr + 1, Nr + 1))
    B = np.zeros((Nr + 1, Nr + 1))

    for i in range(1, Nr):
        A[i, i-1] = alpha[i] - beta[i]
        A[i, i] = 1 + gamma[i] + 2 * alpha[i]
        A[i, i+1] = alpha[i] + beta[i]

        B[i, i-1] = -alpha[i] + beta[i]
        B[i, i] = 1 - gamma[i] - 2 * alpha[i]
        B[i, i+1] = -alpha[i] - beta[i]

    A[0, 0] = A[-1, -1] = 1
    B[0, 0] = B[-1, -1] = 1
```

```
    for j in range(Nt - 1, -1, -1):
        c[:, j] = scipy.linalg.solve(A, B @ c[:, j + 1])

    option_price = np.interp(r0, r, c[:, 0])
    return option_price
```

In [138…
```
problem_3c = problem3c(.05, .12, .92, .055, .05, 1)
problem_3c
```

Out[138… 0.0

Write-Up: both b and c result in 0 which makes sense; intuition-wise since call can easily be OTM.

Problem 4.

Assume the dynamics of the short-term interest rate, under the risk-neutral measure, are given by the following system of SDEs (G2++ model): $dxt = -axtdt + \sigma dWt1$ $dyt = -bytdt + \eta dWt2$ $rt = xt + yt + \phi t$

Use Monte Carlo Simulation to find at time $t = 0$ the price $p(t, T, S, K, \rho)$ of a European Put option, with strike price of $K = \$950$, expiration in $T = 0.5$ years on a Pure Discount Bond with Face value of \$1,000 that matures in $S = 1$ year. Compare it with the price found by the explicit formula and comment on it.

In [171…
```
def problem4a(T, S, K, rho):
    # set parameters
    x0 = 0
    y0 = 0
    a = .01
    b = .3
    sigma = .05
    phi0 = .055
    r0 = .055
    eta = .09
    phi_t = .055
    FV = 1000
    simulations = 10000
    steps = 100
    dt = T/steps

    # simulate correlated brownian motions
    Z1 = np.random.normal(0, 1, (simulations, steps))
    Z2 = np.random.normal(0, 1, (simulations, steps))
    # first bm
    W1 = np.cumsum(Z1*np.sqrt(dt), axis=1)
    W2 = np.cumsum(rho*W1 + np.sqrt(1 - rho**2) * Z2*np.sqrt(dt), axis=1)

    # simulate xt and yt
    x_t = np.zeros((simulations, steps))
    y_t = np.zeros((simulations, steps))
    r_t = np.zeros((simulations, steps))
```

```
    x_t[:, 0] = x0
    y_t[:, 0] = y0
    r_t[:, 0] = r0

    for t in range(1, steps):
        x_t[:, t] = x_t[:, t-1] - a*x_t[:, t-1]*dt + sigma*(W1[:, t] - W1[:,
        y_t[:, t] = y_t[:, t-1] - b*y_t[:, t-1]*dt + eta*(W2[:, t] - W2[:, t
        r_t[:, t] = x_t[:, t] + y_t[:, t] + phi_t

    df = np.exp(-np.cumsum(r_t*dt, axis=1))
    prices = df[:, -1]*FV
    payoff = np.maximum(K - prices, 0)
    option_price = np.mean(payoff) * np.exp(r0 * T)
    return option_price
```

In [172…

```
# value should be below 50
problem_4a = problem4a(.5, 1, 950, .7)
problem_4a
```

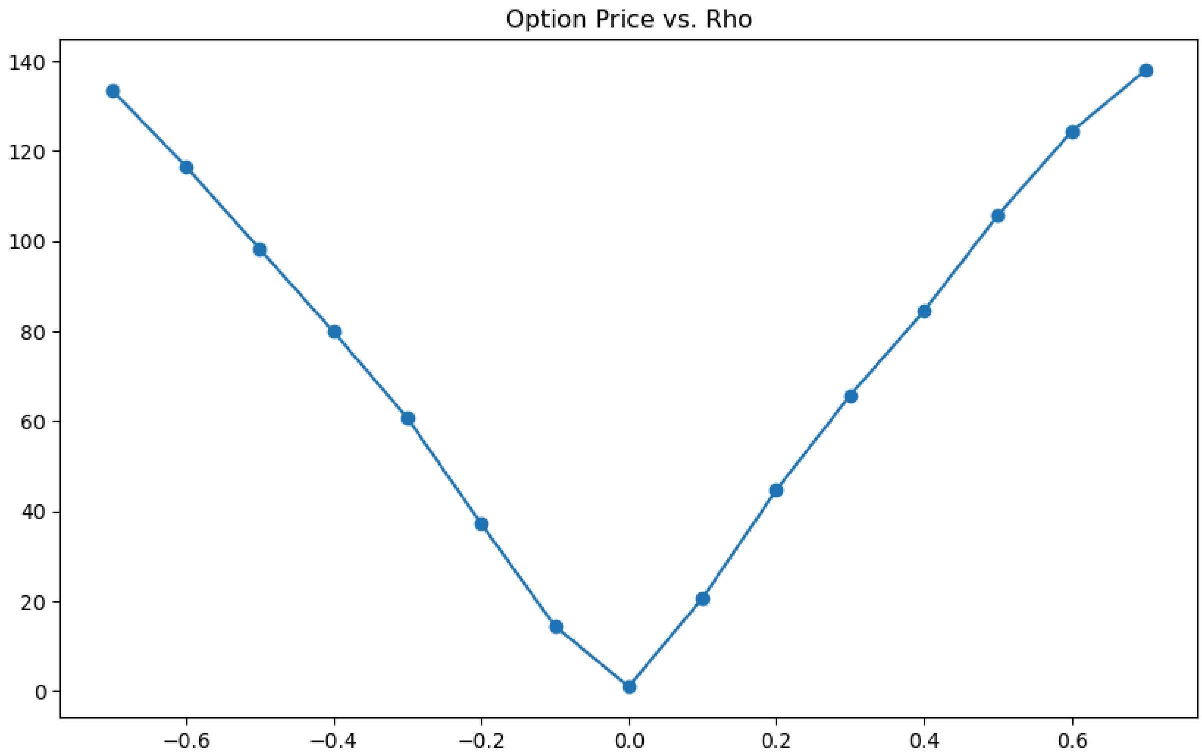
Out[172…

```
137.96750022307972
```

In [175…

```
rho_values = np.linspace(-0.7, 0.7, 15)
option_prices = []

for rho in rho_values:
    price = problem4a(.5, 1, 950, rho)
    option_prices.append(price)
    print(f"Rho: {rho:.2f}, Option Price: {price:.2f}")

plt.figure(figsize=(10, 6))
plt.plot(rho_values, option_prices, marker='o')
plt.title('Option Price vs. Rho')

plt.show()
```

```
Rho: -0.70, Option Price: 133.64
Rho: -0.60, Option Price: 116.67
Rho: -0.50, Option Price: 98.31
Rho: -0.40, Option Price: 79.92
Rho: -0.30, Option Price: 60.77
Rho: -0.20, Option Price: 37.30
Rho: -0.10, Option Price: 14.51
Rho: 0.00, Option Price: 1.02
Rho: 0.10, Option Price: 20.70
Rho: 0.20, Option Price: 44.71
Rho: 0.30, Option Price: 65.75
Rho: 0.40, Option Price: 84.49
Rho: 0.50, Option Price: 105.78
Rho: 0.60, Option Price: 124.38
Rho: 0.70, Option Price: 137.96
```



In [147…

```
from scipy.stats import norm
```

In [173…

```
# aint no way this is right
def problem4b(T, S, K):
    r = .055
    a = .01
    b = .3
    sigma = .05
    eta = .09
    FV = 1000
    # PT and PS
    P_T = FV * np.exp(-r*T)
    P_S = FV * np.exp(-r*S)
    print(P_T)
    print(P_S)

    sm = np.sqrt((sigma**2/2*a**3) * (1 - np.exp(-a*(S-T)))**2 + (eta**2/2*b
                 + 2*rho*((sigma*eta)/(a*b*(a+b))) * (1 - np.exp(-a*(S-T)))

    d1 = np.log(P_S/(K*P_T))/sm + .5*sm
    d2 = np.log(P_S/(K*P_T))/sm - .5*sm
    print(norm.cdf(d1))
    print(norm.cdf(d2))

    payoff = P_S * norm.cdf(d1) - P_T * K * norm.cdf(d2)
    print(payoff)
    option_price = K - payoff
    print('Intuition-wise the price should be around:', P_T - P_S)
    return option_price
```

In [174…

```
problem_4b = problem4b(.5, 1, 950)
```

```
problem_4b
```

```
972.874682553454
946.4851479534839
0.0
0.0
0.0
Intuition-wise the price should be around: 26.389534599970148
```

Out[174…

```
950.0
```

5. Consider a 30-year MBS with a fixed weighted-average-coupon, WAC = 8%. Monthly cash flows are

starting in January of this year. The Notional Amount of the Pool is $100,000. Use the CIR model of interest rates, $drt = \kappa(\bar{r} - rt)dt + \sigma\sqrt{rt}dWt$, with the following default parameters: $r0$ = 0.078, k = 0.6, $\bar{r}$ = 0.08, σ = 0.12.

Consider the Numerix Prepayment Model in all problems below.

(a) Compute the price of the MBS. The code should be generic: the user is prompted for inputs and the program runs and gives the output.

In [187…

```
def CIR(T, r0, kappa, r_bar, sigma, steps=100):
    dt = T/steps  # use 100 time steps
    r = np.zeros(steps+1)   # store simulated values
    r[0] = r0    # set r0
    for t in range(1, steps+1):
        dW = np.random.normal(0, 1) * np.sqrt(dt)   # simulate BM
        r[t] = r[t-1] + kappa*(r_bar - r[t-1])*dt + sigma*np.sqrt(r[t-1])*dW
        r[t] = np.max(r[t], 0)
    return r
```

In [190…

```
def problem5a(r_bar, kappa, sigma):
    steps = 360
    r = CIR(30, .078, kappa, r_bar, sigma, steps)
    wac = .08
    notional = 100000
    monthly_rate = wac / 12
    payment = (notional * monthly_rate) / (1 - (1 + monthly_rate) ** -steps)
    cash_flows = np.zeros(steps)
    outstanding_principal = notional

    for t in range(steps):
        interest_payment = outstanding_principal * monthly_rate
        principal_payment = payment - interest_payment
        outstanding_principal -= principal_payment
        cash_flows[t] = payment

    discount_factors = np.exp(-np.cumsum(r[:steps]) * (30 / steps))
    present_value = np.sum(cash_flows * discount_factors)
    return present_value
```

In [191…
```
problem_5a = problem5a(.08, .6, .12)
problem_5a
```

Out[191…
```
92253.09496118424
```

(b) Compute the Option-Adjusted-Spread (OAS) if the Market Price of MBS is P =$98,000

In [179…
```
from scipy.optimize import minimize
```

In [208…
```
def calculate_cash_flows(wac, notional, steps):
    monthly_rate = wac / 12
    payment = (notional * monthly_rate) / (1 - (1 + monthly_rate) ** -steps)
    cash_flows = np.zeros(steps)
    interest_payments = np.zeros(steps)
    principal_payments = np.zeros(steps)
    remaining_balance = notional


    for t in range(steps):
        interest_payments[t] = remaining_balance * monthly_rate
        principal_payments[t] = payment - interest_payments[t]
        remaining_balance -= principal_payments[t]

    return interest_payments, principal_payments
```

In [209…
```
def problem5b(r_bar, kappa, sigma, P_hat):
    initial_spread = 0
    steps = 360
    dt = 30 / steps
    r = CIR(30, 0.078, kappa, r_bar, sigma, steps)
    wac = 0.08
    notional = 100000
    interest_payments, principal_payments = calculate_cash_flows(wac, notior

    def objective_function(spread):
        discount_factors = np.exp(-np.cumsum(r[:steps] + spread) * dt)
        cash_flows = interest_payments + principal_payments
        present_value = np.sum(cash_flows * discount_factors)
        return (present_value - P_hat) ** 2

    result = minimize(objective_function, initial_spread)
    return result.x[0]
```

In [213…
```
problem_5b = problem5b(.08, .6, .12, 98000)
problem_5b
```

Out[213…
```
0.014435322346019979
```

(c) Consider the MBS described above and the IO and PO tranches. Price the IO and PO tranches

In [214…

```
def problem5c(r_bar, kappa, sigma, OAS):
    steps = 360
    r = CIR(30, 0.078, kappa, r_bar, sigma, steps)
    wac = 0.08
    notional = 100000
    interest_payments, principal_payments = calculate_cash_flows(wac, notior

    dt = 30 / steps
    discount_factors = np.exp(-np.cumsum(r[:steps] + OAS) * dt)
    IO_price = np.sum(interest_payments * discount_factors)
    PO_price = np.sum(principal_payments * discount_factors)

    return IO_price, PO_price
```

In [215…

```
problem_5c = problem5c(.08, .6, .12, problem_5b)
problem_5c
```

Out[215…

```
(74673.2292706732, 22801.310138807683)
```