finm320-24-hw5

May 3, 2024

1 Problem 1

1.1 (a)

The overall portfolio value at time T_2 is:

$$Payoff = (S_{T_2} - K) - (S_{T_2} - F_t) = F_t - K$$

This payoff is determined at T_2 but since it's not dependent on S_{T_2}, its present value at time t is simply its discounted value:

$$f_t = e^{-r(T_2 - t)}(F_t - K)$$

1.2 (b)

When the underlying asset is crude oil, the physical holder incurs storage fees, unlike a non-dividend stock that requires no extra storage cost. The holder may not be willing to accept F_t alone on the delivery date due to these costs.

- If S is a stock paying no dividends, the forward price is $F_t = S_t e^{r(T_2 - t)}$, avoiding arbitrage.
- For S as the spot price of crude oil, additional costs like storage and transportation arise.
- The price to cover becomes $\left(S_t e^{r(T_2 - t)} + \text{Storage Cost}\right)$. The forward price F_t may not cover this, leading to potential arbitrage, as illustrated when $F_t > \left(S_t e^{r(T_2 - t)} + \text{Storage Cost}\right)$.

[1]:
```
import numpy as np
```

[2]:
```
# Exponential Ornstein-Uhlenbeck process

class XOU:

    def __init__(self, kappa, alpha, sigma, S0, r):

        self.kappa = kappa
        self.alpha = alpha
        self.sigma = sigma
        self.S0 = S0
        self.r = r
```

[3]:
```
hw5dynamics=XOU(kappa = 0.472, alpha = 4.4, sigma = 0.368, S0 = 106.9, r = 0.05)
```

[4]:
```
class CallOnForwardPrice:

    def __init__(self, K1, T1, T2):

        self.K1 = K1
        self.T1 = T1
        self.T2 = T2
```

[5]:
```
hw5contract=CallOnForwardPrice(K1 = 103.2, T1 = 0.5, T2 = 0.75)
```

[6]:
```
class MCengine:

    def __init__(self, N, M, epsilon, seed):

        self.N = N   # Number of timesteps on each path
        self.M = M   # Number of paths
        self.epsilon = epsilon  # For the dC/dS calculation
        self.rng = np.random.default_rng(seed=seed) # Seeding the random number␣
 ↪generator with a specified number helps make the calculations reproducible

    def price_call_XOU(self, contract, dynamics):

        # You complete the coding of this function

        # self.rng.normal() generates pseudo-random normals
        K1 = contract.K1
        T1 = contract.T1
        T2 = contract.T2
        kappa = dynamics.kappa
        alpha = dynamics.alpha
        sigma = dynamics.sigma
        S0 = dynamics.S0
        r = dynamics.r
        N = self.N
        M = self.M
        epsilon = self.epsilon

        delta_t = T1 / N
        sigma_p = sigma*np.sqrt(delta_t)
        X = np.zeros((M, N+1))
        X[:,0] = np.ones(M) * np.log(S0)
        X_delta = np.zeros((M, N+1))
        X_delta[:,0] = np.ones(M) * np.log(S0+epsilon)

        # Walk through the Monte Carlo
```

```
        for i in range(N):
            dW = np.random.randn(M)
            X_t = X[:, i]
            X[:, i+1] = X_t + kappa*(alpha - X_t)* delta_t + sigma_p * dW
            X_delta_t = X_delta[:, i]
            X_delta[:, i+1] = X_delta_t + kappa*(alpha - X_delta_t) * delta_t +␣
↪sigma_p * dW

        t = np.linspace(0, T1, N+1)

        F = np.exp(np.exp(-kappa*(T2 - t))* X + (1-np.exp(-kappa*(T2 -␣
↪t)))*alpha + sigma**2/(4*kappa)*(1-np.exp(-2*kappa*(T2 - t))))
        F_delta = np.exp(np.exp(-kappa*(T2 - t))* X_delta + (1-np.
↪exp(-kappa*(T2 - t)))*alpha + sigma**2/(4*kappa)*(1-np.exp(-2*kappa*(T2 -␣
↪t))))

        # compute the current price of call payoff
        call_value = np.exp(-r*T1) * np.maximum(F[:,-1] - K1, 0)
        call_delta_value = np.exp(-r*T1) * np.maximum(F_delta[:,-1] - K1, 0)

        # get the call price
        call_price = np.mean(call_value)
        call_delta_price = np.mean(call_delta_value)

        # sample standard deviation devide by sqrt of number of path
        standard_error = np.std(call_value, ddof = 1)/np.sqrt(M)

        # check if the number of path is large enough
        print(f'se is valid: {standard_error <= 0.05}, as {standard_error}')

        call_delta = (call_delta_price - call_price)/epsilon

        return(call_price, standard_error, call_delta)
```

[7]:
```
hw5MC = MCengine(N=100, M=90000, epsilon=0.01, seed=0)
# Change M if necessary
```

[8]:
```
(call_price, standard_error, call_delta) = hw5MC.
↪price_call_XOU(hw5contract,hw5dynamics)
```

```
se is valid: True, as 0.04412803128475711
```

[9]:
```
print(call_price, standard_error, call_delta)
```

```
7.663852001148125 0.04412803128475711 0.3392131556547717
```

1.3 (e)

$$F_0 = \exp\left[e^{-\kappa T_2}\log S_0 + \left(1 - e^{-\kappa T_2}\right)\alpha + \frac{\sigma^2}{4\kappa}\left(1 - e^{-2\kappa T_2}\right)\right].$$

$$\frac{\partial F_0}{\partial S_0} = \exp\left[e^{-\kappa T_2}\log S_0 + \text{const}\right]\cdot e^{-\kappa T_2}\frac{1}{S_0}$$
$$= F_0 \cdot e^{-\kappa T_2}\frac{1}{S_0}$$

By this discount factor, we have:

$$\frac{\partial f_0}{\partial S} = e^{-rT_2}\cdot\frac{\partial F_0}{\partial S_0}$$
$$= e^{-rT_2}\cdot F_0 \cdot e^{-\kappa T_2}\frac{1}{S_0} = 0.64651.$$

[10]:
```python
# Numerically
kappa = 0.472
alpha = 4.4
sigma = 0.368
r = 0.05
S0 = 106.9
T2 = 0.75

F0 = np.exp(np.exp(-hw5dynamics.kappa * hw5contract.T2) * np.log(hw5dynamics.
 ↪S0) + (1 - np.exp(-hw5dynamics.kappa * hw5contract.T2)) * hw5dynamics.alpha␣
 ↪+ ((hw5dynamics.sigma**2)/(4*hw5dynamics.kappa)) * (1 - np.
 ↪exp(-2*hw5dynamics.kappa*hw5contract.T2)))

delta_1e = np.exp(-hw5dynamics.r*hw5contract.T2)*F0*np.exp(-hw5dynamics.
 ↪kappa*hw5contract.T2)*(1/hw5dynamics.S0)
print(f'The partial deviative of f0 w.r.t S will be {delta_1e:.5f}.')
```

```
The partial deviative of f0 w.r.t S will be 0.64651.
```

1.4 (f)

We want this portfolio to be delta hedges.

$\frac{\partial C_0}{\partial S}$ by question 1(d). $\frac{\partial f_0}{\partial S}$ by question 1(e).

Therefore,

$$\Delta = \frac{\frac{\partial C_0}{\partial S}}{\frac{\partial f_0}{\partial S}}$$

[11]:
```python
delta_1f = call_delta/delta_1e
print(f'The hedge portfolio at time 0 should be long {delta_1f:.5f} forward␣
 ↪contracts')
```

The hedge portfolio at time 0 should be long 0.52468 forward contracts

1.5 (g)

For the purchase agreement contract, at time T2, the value of the portfolio should be: $V_{T_2} = \theta(F_{T_2} - K) + (\theta_{max} - \theta)(F_{T_1} - K)^+$, the second part represents the opportunity cost for not choosing the maximum number of units of crude oil at T1.

If we discount back to time-0, it becomes $\theta f_0 + (\theta_{max} - \theta)C(S_0)$, which is the answer from question 1(c); therefore, we can compute the time-0 purchase agreement as below.

Since holder will act optimally, so the theta value should be 4000.

From question 1(e), we can get the formula to compute for F0 and f0, so the formula and calculation should be:

[12]:
```
T1 = 0.5
K = 103.2
F0 = S0**(np.exp(-kappa*T2))*np.exp((1-np.exp(-kappa*T2))*alpha+sigma**2/
 ↪(4*kappa)*(1-np.exp(-2*kappa*T2)))
f0 = np.exp(-r*T2)*(F0 - K)
theta = 4000
max_theta = 5000

pa = theta * f0 + (max_theta - theta) * call_price
print(f'The time-0 value of this contract is {pa:.5f}')
```

```
The time-0 value of this contract is 3928.01485
```

2 Problem 2

2.1 (a)

By lecture note:

$$\sigma_{imp} = \bar{\sigma}_t = \sqrt{\frac{1}{T}\int_0^T \sigma^2(t)dt},$$

which is related to T but not K.

Therefore, the dynamics can produce a non-constant term structure of implied volatility but not an implied volatility skew.

1. **Term-Structure of Implied Volatility**: The specified dynamics are capable of generating a non-constant term-structure of implied volatility, because the volatility function $\sigma(t)$ depends on time t. This allows the model to adapt different volatilities at different times, leading to a term-structure where implied volatility changes over different expiration times.
2. **Implied Volatility Skew**: Since $\sigma(t)$ does not depend on the stock price S, this model does not naturally produce an implied volatility skew with respect to the strike price K. The model's volatility is uniform across different strike prices at any given time because it does

not incorporate factors like leverage or the stochastic nature of volatility, which are typically needed to model skew.

2.2 (b)

[13]:
```
import numpy as no
from scipy.stats import norm
from scipy.optimize import brentq
```

```
/opt/anaconda3/lib/python3.8/site-packages/scipy/__init__.py:146: UserWarning: A
NumPy version >=1.16.5 and <1.23.0 is required for this version of SciPy
(detected version 1.24.4
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
```

[14]:
```
class GBMdynamics:

    def __init__(self, S, r, rGrow, sigma=None):
        self.S = S
        self.r = r
        self.rGrow = rGrow
        self.sigma = sigma

    def update_sigma(self, sigma):
        self.sigma = sigma
        return self
```

[15]:
```
class CallOption:

    def __init__(self, K, T, price=None):
        self.K = K
        self.T = T
        self.price = price

    def BSprice(self, dynamics):
        # ignores self.price if given, because this function calculates price␣
↪based on the dynamics

        F = dynamics.S*np.exp(dynamics.rGrow*self.T)
        sd = dynamics.sigma*np.sqrt(self.T)
        d1 = np.log(F/self.K)/sd+sd/2
        d2 = d1-sd
        return np.exp(-dynamics.r*self.T)*(F*norm.cdf(d1)-self.K*norm.cdf(d2))

    def trysigma(self, inputsigma, dynamics, targetprice):
        dynamics.sigma = inputsigma
        return self.BSprice(dynamics) - targetprice
```

```python
    def IV(self, dynamics):
        # ignores dynamics.sigma, because this function solves for sigma.

        if self.price is None:
            raise ValueError('Contract price must be given')

        df = np.exp(-dynamics.r*self.T)  #discount factor
        F = dynamics.S / df
        lowerbound = np.max([0,(F-self.K)*df])
        C = self.price
        if C<lowerbound:
            return np.nan
        if C==lowerbound:
            return 0
        if C>=F*df:
            return np.nan

        dytry = dynamics
        # We "try" values of sigma until we find sigma that generates price C

        # First find lower and upper bounds
        dytry.sigma = 0.2
        while self.BSprice(dytry)>C:
            dytry.sigma /= 2
        while self.BSprice(dytry)<C:
            dytry.sigma *= 2
        hi = dytry.sigma
        lo = hi/2

        impliedVolatility = brentq(self.trysigma, hi, lo, args=(dynamics, self.
↪price), maxiter=1000)
        return impliedVolatility
```

[16]:
```python
# first IV
dynamics = GBMdynamics(S=100, r=0.05, rGrow=0.05)
contract1 = CallOption(K=100, T=0.1, price=5.25)
imp_vol_1 = contract1.IV(dynamics)
imp_vol_1
```

[16]: 0.397320385795576

[17]:
```python
# second IV
contract2 = CallOption(K=100, T=0.2, price=7.25)
imp_vol_2 = contract2.IV(dynamics)
imp_vol_2
```

[17]: 0.380171291551054

```
[18]: # thrid IV
contract3 = CallOption(K=100, T=0.5, price=9.5)
imp_vol_3 = contract3.IV(dynamics)
imp_vol_3
```

```
[18]: 0.2950972521756794
```

There is $\bar{\sigma}_T = \sqrt{\frac{1}{T}\int_0^T \sigma^2(t)dt}$.

Therefore, we can obtain the time-varying function:

$$\sigma(t) = \begin{cases} \bar{\sigma}_{0.1} & 0 \le t \le 0.1 \\ \sqrt{2\bar{\sigma}_{0.2}^2 - \bar{\sigma}_{0.1}^2} & 0.1 < t \le 0.2 \\ \sqrt{\frac{5}{3}\bar{\sigma}_{0.5}^2 - \frac{2}{3}\bar{\sigma}_{0.2}^2} & 0.2 < t \le 0.5 \end{cases}$$

2.3 (c)

$\bar{\sigma}_{0.4} = \sqrt{\frac{1}{0.4}\int_0^{0.4} \sigma^2(t)dt}$

$= \sqrt{\frac{1}{0.4} * (0.1\bar{\sigma}_1^2 + 0.1(2\bar{\sigma}_2^2 - \bar{\sigma}_1^2) + 0.2(\frac{1}{3}(5\bar{\sigma}_3^2 - 2\bar{\sigma}_2^2))}$

$= \sqrt{\frac{5}{6}\bar{\sigma}_3^2 + \frac{1}{6}\bar{\sigma}_2^2}$

```
[19]: #IV
implied_vol_4 = np.sqrt(5/6*imp_vol_3**2 + 1/6*imp_vol_2**2)
implied_vol_4
```

```
[19]: 0.31089712985910606
```

```
[20]: def bs_call_formula(X, t, K, T, rGrow, r, sigma):
    F = X*np.exp(rGrow*(T-t))
    d1 = np.log(F/K)/(sigma*np.sqrt(T-t)) + sigma*np.sqrt(T-t)/2
    d2 = np.log(F/K)/(sigma*np.sqrt(T-t)) - sigma*np.sqrt(T-t)/2
    call_price = np.exp(-r*(T-t))*(F*norm.cdf(d1) - K*norm.cdf(d2))
    return call_price
```

```
[21]: #call price
bs_call = bs_call_formula(X = 100, t = 0, K = 100, T = 0.4, rGrow = 0.05, r = 0.
 ↪05, sigma = implied_vol_4)
bs_call
```

```
[21]: 8.784201775930828
```