finm320_24_final

May 22, 2024

[16]:
```python
import numpy as np
from scipy.stats import norm
from scipy.optimize import newton
```

[2]:
```python
class GBM:

    def __init__(self,S0,r,sigma=None):
        self.S0=S0
        self.r=r
        self.sigma=sigma
```

[3]:
```python
class CallOnAverage:

    def __init__(self,K,T0,T1):
        self.K = K
        self.T0 = T0
        self.T1 = T1
```

[4]:
```python
class MCengine:

    def __init__(self,M,seed):
        self.M = M  #number of paths
        self.rng = np.random.default_rng(seed=seed) # Seeding the random number␣
↪generator with a specified number helps make the calculations reproducible

    def randomLogreturn(self,dynamics,deltat):
        return (dynamics.r-dynamics.sigma**2/2)*deltat + dynamics.sigma*np.
↪sqrt(deltat)*self.rng.normal(size=self.M)


    def price_CallOnAverage_GBM_Conditional(self, contract, dynamics):
        S05 = dynamics.S0 * np.exp(self.randomLogreturn(dynamics, contract.T0))
        zeropayoff = np.zeros(self.M)

        # Special case for S05 >= 24
        special_case_indices = S05 >= 24
```

```
        S10_special = S05[special_case_indices]  # S10 in this case is just S05␣
↪since S05 is very large
        payoff_special = np.maximum(S05[special_case_indices] - contract.K,␣
↪zeropayoff[special_case_indices])

        # General case
        logreturn_S10 = (dynamics.r - dynamics.sigma**2 / 2) * (contract.T1 -␣
↪contract.T0) + dynamics.sigma * np.sqrt(contract.T1 - contract.T0) * self.
↪rng.normal(size=self.M)
        S10 = S05 * np.exp(logreturn_S10)
        payoff_general = np.maximum((S05 + S10) / 2 - contract.K, zeropayoff)

        # Combine payoffs
        payoff = np.copy(payoff_general)
        payoff[special_case_indices] = payoff_special

        payoffdiscounted = np.exp(-dynamics.r * contract.T1) * payoff
        price = np.mean(payoffdiscounted)
        standard_error = np.std(payoffdiscounted) / np.sqrt(self.M)
        return price, standard_error
```

[5]:
```
p3dynamics = GBM(sigma=0.70,S0=10,r=0.02)
```

[6]:
```
p3contract = CallOnAverage(K=12,T0=0.5,T1=1.0)
```

[7]:
```
p3MC = MCengine(M=100000,seed=0)
(p3price, p3standard_error) = p3MC.
 ↪price_CallOnAverage_GBM_Conditional(p3contract,p3dynamics)
print(p3price, p3standard_error)
```

```
1.567995896631187 0.012507517756089392
```

[ ]:

1 Problem 5

[19]:
```
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

def implied_volatility(call_price, X, t, K, T, rGrow, r):
    # Define the objective function for the root-finding algorithm
    def objective_function(sigma):
        return bs_call_formula(X, t, K, T, rGrow, r, sigma) - call_price
```

```
    # Initial guess for the volatility
    sigma_initial = 0.2

    # Use the Newton-Raphson method to find the root
    implied_vol = newton(objective_function, sigma_initial)

    return implied_vol
```

[26]:
```
#call price for T1
bs_call = bs_call_formula(X = 100, t = 0, K = 100, T = 0.3, rGrow = 0, r = 0,␣
 ↪sigma = 0.34)
print(bs_call)

#implied volatility for T2
implied_vol = implied_volatility(call_price = 8.01, X = 100, t = 0, K = 100, T␣
 ↪= 0.4, rGrow = 0, r = 0)
print(implied_vol)

#call price for T3
bs_call2 = bs_call_formula(X = 100, t = 0, K = 100, T = 0.6, rGrow = 0, r = 0,␣
 ↪sigma = 0.3)
print(bs_call2)
```

```
7.418607894112
0.317997649891402
9.24976421293605
```

[ ]:
```
```