MF796hw4

March 19, 2024

0.1 MF 796: Computational Methods of Mathematical Finance

Professors Christopher Kelliher and Eugene Sorets Spring 2024

0.1.1 Problem set 4

Problem 1: Simulation in the Heston Model Suppose that the underlying security SPY evolves according to the Heston model. That is, we know its dynamics are defined by the following system of SDEs:

$$dS_t = (r - q)S_t dt + \sqrt{\nu_t} S_t dW_{t1}$$

$$d\nu_t = \kappa(\theta - \nu_t)dt + \sigma\sqrt{\nu_t} dW_{t2}$$

$$\text{Cov}(dW_{t1}, dW_{t2}) = \rho dt$$

You know that the last closing price for SPY was 500. You also know that the dividend yield for SPY is 1.35% and the corresponding risk-free rate is 3.5%.

Using this information, you want to build a simulation algorithm to price a knock-out option on SPY, where the payoff is a European call option contingent on the option not being knocked out, and the knock-out is an upside barrier that is continuously monitored. We will refer to this as an up-and-out call.

This payoff can be written as:

$$c_0 = \mathbb{E}\left[(S_T - K_1)^+ \mathbf{1}_{\{M_T < K_2\}}\right]$$

where M_T is the maximum value of S over the observation period, and $K_1 < K_2$ are the strikes of the European call and the knock-out trigger, respectively.

(a) Find a set of Heston parameters that you believe govern the dynamics of SPY. You may use code from a prior homework, do this via a new calibration, or some other empirical process. Explain how you got these and why you think they are reasonable.

[39]:
```python
import yfinance as yf
import pandas as pd
from datetime import datetime
```

```
today = datetime.today().date()

def get_option_data(expiry, option_type='calls'):
    spy = yf.Ticker("SPY")
    options = spy.option_chain(expiry)

    data = options.calls if option_type == 'calls' else options.puts

    data['expDays'] = (pd.to_datetime(expiry) - pd.Timestamp(today)).days
    data['expT'] = data['expDays'] / 365

    data = data.rename(columns={'strike': 'K', 'bid': f'{option_type}_bid',␣
 ↪'ask': f'{option_type}_ask'})
    data = data[['expDays', 'expT', 'K', f'{option_type}_bid',␣
 ↪f'{option_type}_ask']]

    return data

expiries = ['2024-03-28', '2024-06-28', '2024-09-30']

all_calls = pd.concat([get_option_data(expiry, 'calls') for expiry in expiries])
all_puts = pd.concat([get_option_data(expiry, 'puts') for expiry in expiries])

options_data = pd.merge(all_calls, all_puts, on=['expDays', 'expT', 'K'],␣
 ↪how='inner')

options_data = options_data.sort_values(by=['expDays', 'K']).
 ↪reset_index(drop=True)

print(options_data.head())
options_data.to_excel('/Users/apple/Downloads/SPY_Options_Data_Combined.xlsx',␣
 ↪index=False)
```

```
   expDays      expT      K  calls_bid  calls_ask  puts_bid  puts_ask
0        9  0.024658  275.0     240.93     241.36       0.0      0.01
1        9  0.024658  280.0     235.94     236.37       0.0      0.01
2        9  0.024658  285.0     230.96     231.38       0.0      0.01
3        9  0.024658  290.0     225.98     226.38       0.0      0.01
4        9  0.024658  295.0     220.99     221.39       0.0      0.01
```

[40]:
```
# Calculate the mid prices for call and put options
options_data['calls_mid'] = (options_data['calls_bid'] +␣
 ↪options_data['calls_ask']) / 2
options_data['puts_mid'] = (options_data['puts_bid'] +␣
 ↪options_data['puts_ask']) / 2
```

```
# Check for arbitrage opportunities at the mid price by ensuring monotonic␣
↪decrease in call option prices
# and monotonic increase in put option prices with increasing strike price
arbitrage_free_calls_mid = options_data['calls_mid'].diff().dropna() <= 0
arbitrage_free_puts_mid = options_data['puts_mid'].diff().dropna() >= 0

# Identifying rows where arbitrage violations occur
violations_calls_mid = arbitrage_free_calls_mid[arbitrage_free_calls_mid ==␣
↪False]
violations_puts_mid = arbitrage_free_puts_mid[arbitrage_free_puts_mid == False]

# Remove identified arbitrage violations from the dataset
options_data_cleaned = options_data.drop(violations_calls_mid.index.
↪union(violations_puts_mid.index))

# Check and adjust for bid-ask spread
# For call options: remove any option where a higher strike has a higher bid␣
↪price than a lower strike's ask price
# For put options: remove any option where a lower strike has a higher ask␣
↪price than a higher strike's bid price
call_bid_ask_violations = options_data_cleaned['calls_bid'].diff().dropna() > 0
put_bid_ask_violations = options_data_cleaned['puts_ask'].diff().dropna() < 0

# Identifying rows where bid-ask arbitrage violations occur
violations_call_bid_ask = call_bid_ask_violations[call_bid_ask_violations ==␣
↪True]
violations_put_bid_ask = put_bid_ask_violations[put_bid_ask_violations == True]

# Final cleaning by removing identified bid-ask spread violations
final_cleaned_data = options_data_cleaned.drop(violations_call_bid_ask.index.
↪union(violations_put_bid_ask.index))

# Summary of actions
summary = {
    'initial_data_count': len(options_data),
    'after_mid_check_count': len(options_data_cleaned),
    'final_data_count': len(final_cleaned_data),
    'removed_for_mid_violations': len(violations_calls_mid) +␣
↪len(violations_puts_mid),
    'removed_for_bid_ask_violations': len(violations_call_bid_ask) +␣
↪len(violations_put_bid_ask)
}

summary
```

```
[40]: {'initial_data_count': 424,
       'after_mid_check_count': 365,
       'final_data_count': 331,
       'removed_for_mid_violations': 68,
       'removed_for_bid_ask_violations': 39}
```

```
[41]: import numpy as np
      from scipy import interpolate
      import time
      import matplotlib.pyplot as plt
      import matplotlib.ticker as ticker
```

```
[42]: #Define the characteristic equation of the Heston Model
      def heston_characteristic_function(u, params):
          iu = 1j * u

          kappa, theta, sigma, rho, v0, t, S0, r, q  = params['kappa'],␣
        ↪params['theta'], params['sigma'], params['rho'], params['v0'], params['t'],␣
        ↪params['S0'], params['r'], params['q']

          lambda_ = np.sqrt(sigma**2 * (u**2 + iu) + (kappa - iu * rho * sigma)**2)

          omega_numerator =  np.exp(iu * np.log(S0) + iu * (r - q) * t + kappa *␣
        ↪theta * t * (kappa - iu * rho * sigma) / sigma**2)

          omega_denominator  = (np.cosh(lambda_ * t * 0.5) + (kappa - iu * rho *␣
        ↪sigma) / lambda_ * np.sinh(lambda_ * t * 0.5))**(2 * kappa * theta *␣
        ↪sigma**(-2))

          omega_u = omega_numerator/omega_denominator

          coth_lambda_t = 1/np.tanh(0.5 * lambda_ * t)

          phi = omega_u * np.exp(-((u**2 + iu) * v0)/(lambda_ * coth_lambda_t +␣
        ↪(kappa - iu * rho * sigma)))

          return phi
```

```
[52]: #Calculate the European Call Option Price of the Heston Model using FFT
      def calc_fft_heston_call_prices(alpha, params, N, delta_v, K = None):
          #delta is the indicator function
          kappa, theta, sigma, rho, v0, t, S0, r, q  = params['kappa'],␣
        ↪params['theta'], params['sigma'], params['rho'], params['v0'], params['t'],␣
        ↪params['S0'], params['r'], params['q']

          delta = np.zeros(N)
          delta[0] = 1
```

```
    delta_k = (2*np.pi)/(N*delta_v)

    if K == None:
        #middle strike is at the money
        beta = np.log(S0) - delta_k*N*0.5

    else:
        #middle strike is K
        beta = np.log(K) - delta_k*N*0.5

    k_list = np.array([(beta +(i-1)*delta_k) for i in range(1,N+1) ])
    v_list = np.arange(N) * delta_v

    #building fft input vector
    x_numerator = np.array( [((2-delta[i])*delta_v)*np.exp(-r*t)  for i in␣
 ↪range(N)] )
    x_denominator = np.array( [2 * (alpha + 1j*i) * (alpha + 1j*i + 1) for i in␣
 ↪v_list] )
    x_exp = np.array( [np.exp(-1j*(beta)*i) for i in v_list] )

    x_list = (x_numerator/x_denominator)*x_exp* np.
 ↪array([heston_characteristic_function(i - 1j*(alpha+1),params) for i in␣
 ↪v_list])

    #fft output
    y_list = np.fft.fft(x_list)
    #recovering prices
    prices = np.array( [(1/np.pi) * np.exp(-alpha*(beta +(i-1)*delta_k)) * np.
 ↪real(y_list[i-1]) for i in range(1,N+1)] )

    return prices, np.exp(k_list)
```

[44]:
```
from scipy.optimize import minimize

def optimization_objective(params, market_prices, K, expiries, S0, r, q, N,␣
 ↪delta_v, alpha):
    total_error = 0
    for i, expiry in enumerate(expiries):
        params['t'] = expiry
        model_prices, strikes, _ = calc_fft_heston_call_prices(alpha, params,␣
 ↪N, delta_v, K=K[i])
        interp_func = interpolate.interp1d(strikes, model_prices, kind='cubic',␣
 ↪fill_value="extrapolate")
        model_price_at_K = interp_func(K[i])
        error = (model_price_at_K - market_prices[i]) ** 2
        total_error += error
    return total_error
```

[47]:
```
market_prices = final_cleaned_data['calls_mid'].values
K = final_cleaned_data['K'].values
expiries = final_cleaned_data['expT'].values

S0 = 500  # Last closing price for SPY
r = 0.035  # Corresponding risk-free rate
q = 0.0135  # Dividend yield for SPY
N = 4096  # Assuming this is a numerical parameter for the model, unchanged
delta_v = 0.25  # Assuming this is a numerical parameter for the model,␣
 ↪unchanged
alpha = 1.5  # Assuming this is a numerical parameter for the model, unchanged

initial_params = [1.5, 0.04, 0.2, -0.5, 0.04]  # kappa, theta, sigma, rho, v0
bounds = [(0.1, 10), (0.01, 0.5), (0.01, 1), (-0.99, 0.99), (0.01, 0.5)]  #␣
 ↪Parameter bounds

def print_status(xk):
    print(f"Current params: {xk}")

def optimization_objective(params, market_prices, K, expiries, S0, r, q, N,␣
 ↪delta_v, alpha):
    total_error = 0
    for i, expiry in enumerate(expiries):
        local_params = {'kappa': params[0], 'theta': params[1], 'sigma':␣
 ↪params[2], 'rho': params[3], 'v0': params[4], 'S0': S0, 'r': r, 'q': q, 't':␣
 ↪expiry}
        model_prices, strikes = calc_fft_heston_call_prices(alpha,␣
 ↪local_params, N, delta_v)
        interp_func = interpolate.interp1d(strikes, model_prices, kind='cubic',␣
 ↪fill_value="extrapolate")
        model_price_at_K = interp_func(K[i])
        error = (model_price_at_K - market_prices[i]) ** 2
        total_error += error
    print(f"Total error: {total_error}")
    return total_error


result = minimize(optimization_objective, initial_params,
                  args=(market_prices, K, expiries, S0, r, q, N, delta_v,␣
 ↪alpha),
                  bounds=bounds,
                  method='L-BFGS-B',
                  options={'disp': True, 'maxiter': 50, 'gtol': 1e-3, 'ftol':␣
↪1e-3},
                  callback=print_status)

print("initial_params", initial_params)
print("bounds", bounds)
print("Final optimized parameters:", result.x)
```

```
Total error: 105129.02377165061
Total error: 105129.02377354392
Total error: 105129.02219357052
Total error: 105129.02369444337
Total error: 105129.02381215923
Total error: 105129.0183902101
RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =            5     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f=  1.05129D+05    |proj g|=  1.40000D+00
Total error: 316296.4366520973
Total error: 316296.4367308555
Total error: 316296.43644895486
Total error: 316296.43738772016
Total error: 316296.4367156501
Total error: 316296.42744003684
Total error: 88613.60467777867
Total error: 88613.6046817821
Total error: 88613.60525730687
Total error: 88613.60459113528
Total error: 88613.60470523426
Total error: 88613.60610169194
Current params: [ 1.21672456  0.13307621  0.36187168 -0.5991464   0.13307621]

At iterate    1    f=  8.86136D+04    |proj g|=  1.11672D+00
Total error: 86150.14190568066
Total error: 86150.14190857923
Total error: 86150.14226279454
Total error: 86150.14182844367
Total error: 86150.14193612682
Total error: 86150.14243969515
Current params: [ 1.27046548  0.11306     0.3325147  -0.58080873  0.11462672]

At iterate    2    f=  8.61501D+04    |proj g|=  1.17047D+00
```

```
Total error: 85620.62277374986
Total error: 85620.62277579126
Total error: 85620.62293597944
Total error: 85620.6227009584
Total error: 85620.62280678794
Total error: 85620.62260812482
Current params: [ 1.30658547  0.09428533  0.31453981 -0.5692773   0.10336973]

At iterate    3    f=  8.56206D+04    |proj g|=  1.20659D+00
Total error: 85500.90176606143
Total error: 85500.901767581
Total error: 85500.9019367487
Total error: 85500.90169284103
Total error: 85500.90179926952
Total error: 85500.90163898941
Total error: 85063.19923740062
Total error: 85063.19923620606
Total error: 85063.19944171992
Total error: 85063.19916237355
Total error: 85063.19927120423
Total error: 85063.19926434709
Current params: [ 1.28934737  0.07654668  0.33280611 -0.57901633  0.11256952]

At iterate    4    f=  8.50632D+04    |proj g|=  8.71065D+00
Total error: 88106.83425310781
Total error: 88106.8341802806
Total error: 88106.8349173541
Total error: 88106.83414314364
Total error: 88106.83428260083
Total error: 88106.83623800972
Total error: 84792.5691831286
Total error: 84792.56917357548
Total error: 84792.56950674736
Total error: 84792.56910303902
Total error: 84792.5692166434
Total error: 84792.56968445322
Current params: [ 1.26602817  0.06203611  0.36458908 -0.59533003  0.12769149]

At iterate    5    f=  8.47926D+04    |proj g|=  8.73397D+00
Total error: 85523.60607902039
Total error: 85523.60602876225
Total error: 85523.60661345496
Total error: 85523.60598168672
Total error: 85523.60611123324
Total error: 85523.607503361
Total error: 84739.14898533432
Total error: 84739.14896955808
Total error: 84739.14935576098
Total error: 84739.14890232276
Total error: 84739.1490188057
Total error: 84739.14968560723
Current params: [ 1.25572262  0.05133261  0.38140456 -0.60375309  0.1355179 ]
initial_params [1.5, 0.04, 0.2, -0.5, 0.04]
bounds [(0.1, 10), (0.01, 0.5), (0.01, 1), (-0.99, 0.99), (0.01, 0.5)]
Final optimized parameters: [ 1.25572262  0.05133261  0.38140456 -0.60375309
0.1355179 ]

At iterate    6    f=  8.47391D+04    |proj g|=  8.74428D+00

           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
    5      6     11     10     0     0   8.744D+00   8.474D+04
  F =   84739.148985334323

CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
```

calibrating the Heston model parameters to market data of SPY options using a combination of preprocessing to ensure no arbitrage in the options data, followed by an optimization routine to minimize the difference between market and model prices.

(b) Choose a discretization for the Heston SDE. In particular, choose the time spacing, T as well as the number of simulated paths, N. Explain why you think these choices will lead to an accurate result. Assuming we're pricing a European option with a maturity of 1 year, we could choose:
$\Delta T = 1/252$, which corresponds to one trading day.
N = 10,000 paths initially, with tests for convergence by increasing N to 50,000 or 100,000.
These choices would provide a good starting point. From there, adjustments can be made based on empirical results and available computing resources.

(c) Write a simulation algorithm to price a European call with strike K = 540 and time to expiry T = 1. Calculate the price of this European call using FFT and comment on the difference in price.

[54]:
```python
# Heston model parameters
kappa = 1.25  # Speed of mean reversion
```

```
theta = 0.05  # Long-term volatility
sigma = 0.38  # Volatility of volatility
rho = -0.60  # Correlation between asset and volatility
v0 = 0.13  # Initial volatility
S0 = 500  # Last closing price for SPY
K = 540  # Strike price
T = 1  # Time to expiry
r = 0.035  # Risk-free rate
q = 0.0135  # Dividend yield

# Simulation parameters
M = 252  # Number of time steps
dt = T / M  # Time step size
N = 10000  # Number of simulation paths

# Initialize the price and variance processes
S_t = np.full(N, S0)
nu_t = np.full(N, v0)

# Generate correlated random sequences
rand_z1 = np.random.normal(size=(N, M))
rand_z2 = np.random.normal(size=(N, M))
rand_z2 = rho * rand_z1 + np.sqrt(1 - rho ** 2) * rand_z2

# Simulation loop
for t in range(1, M + 1):
    nu_t = np.maximum(nu_t, 0)  # Ensure variance is non-negative
    S_t = S_t * np.exp((r - q - 0.5 * nu_t) * dt + np.sqrt(nu_t * dt) *␣
 ↪rand_z1[:, t - 1])
    nu_t = nu_t + kappa * (theta - nu_t) * dt + sigma * np.sqrt(nu_t * dt) *␣
 ↪rand_z2[:, t - 1]

# Compute the payoffs for European Call
payoffs = np.maximum(S_t - K, 0)

# Discount the payoffs to present value and average
option_price = np.exp(-r * T) * np.mean(payoffs)

print("The European Call option price using Monte Carlo simulation is:",␣
 ↪option_price)
```

```
The European Call option price using Monte Carlo simulation is:
43.31770310210859
```

[53]:
```
# Heston model parameters
params = {
    'S0': 500,  # Initial stock price
    'K': 540,  # Strike price
    't': 1,  # Time to maturity
    'r': 0.035,  # Risk-free rate
    'q': 0.0135,  # Dividend yield
    'sigma': 0.38,  # Volatility of variance
    'v0': 0.13,  # Initial variance
    'kappa': 1.25,  # Rate of mean reversion
    'rho': -0.60,  # Correlation between asset and variance
    'theta': 0.05,  # Long-term variance
}

# FFT parameters
alpha = 1.5      # Damping factor
N = 2**12        # Number of FFT points
delta_v = 0.25   # Step size in the frequency domain

# Calculate option price
prices, strikes = calc_fft_heston_call_prices(alpha, params, N, delta_v, K)

# Select a specific price to display, for example, the middle price
mid_index = len(prices) // 2
print("Fast Fourier Transform (FFT) method")
print(f"The option price at selected strike ({strikes[mid_index]}) is:␣
 ↪{prices[mid_index]}")
```

```
Fast Fourier Transform (FFT) method
The option price at selected strike (540.0000000000002) is: 44.648072051799886
```

The results of the two methods are very similar, so it can be considered that the implementation of the two methods is reasonable

(d) Update your simulation algorithm to price an up-and-out call with T = 1, K1 = 540, and K2 = 600. Try this for several values of N. What do you need to get an accurate price?

[36]:
```
# Function to price an up-and-out European call option using the Heston model␣
 ↪with Monte Carlo simulation
def price_up_and_out_call_monte_carlo(S0, K1, K2, T, r, q, kappa, theta, sigma,␣
 ↪rho, v0, M, N):
    dt = T / M  # Time step size
    S_t = np.full(N, S0)  # Initialize the price process
    nu_t = np.full(N, v0)  # Initialize the variance process
    payoff_sum = 0.0  # Initialize the sum of payoffs

    for i in range(N):
        max_S_t = S0
        S_ti = S0
        nu_ti = v0
        for t in range(1, M + 1):
            # Generate correlated random sequences
            rand_z1 = np.random.normal()
            rand_z2 = np.random.normal()
            rand_z2 = rho * rand_z1 + np.sqrt(1 - rho ** 2) * rand_z2

            # Update the variance and price processes
            nu_ti = np.maximum(nu_ti + kappa * (theta - nu_ti) * dt + sigma *␣
 ↪np.sqrt(nu_ti * dt) * rand_z2, 0)
            S_ti = S_ti * np.exp((r - q - 0.5 * nu_ti) * dt + np.sqrt(nu_ti *␣
 ↪dt) * rand_z1)

            # Check for the maximum price in the path
            max_S_t = max(max_S_t, S_ti)

            # If the barrier is hit, the option is knocked out
            if max_S_t >= K2:
                break

        # If the option has not been knocked out, calculate its payoff
        if max_S_t < K2:
            payoff = max(S_ti - K1, 0)
            payoff_sum += payoff

    # Discount the average payoff to get the option price
    option_price = np.exp(-r * T) * (payoff_sum / N)
    return option_price

# Parameters for the up-and-out call option
K1 = 540  # Strike price
K2 = 600  # Barrier level

# Try several values of N for convergence
path_counts = [1000, 5000, 10000, 20000, 50000]

# Calculate and print the option prices for different numbers of paths
up_and_out_prices = []
for N in path_counts:
    option_price = price_up_and_out_call_monte_carlo(S0, K1, K2, T, r, q,␣
 ↪kappa, theta, sigma, rho, v0, M, N)
    up_and_out_prices.append(option_price)
    print(f"The up-and-out Call option price using Monte Carlo simulation with␣
 ↪N={N} paths is: {option_price:.4f}")

# Analyze the result to determine what is needed for an accurate price
up_and_out_prices
```

```
The up-and-out Call option price using Monte Carlo simulation with N=1000 paths
is: 1.2236
The up-and-out Call option price using Monte Carlo simulation with N=5000 paths
is: 0.9383
The up-and-out Call option price using Monte Carlo simulation with N=10000 paths
is: 0.9446
The up-and-out Call option price using Monte Carlo simulation with N=20000 paths
is: 0.8744
The up-and-out Call option price using Monte Carlo simulation with N=50000 paths
is: 0.9306
```

```
[36]: [1.2235905967383782,
       0.9383011510888746,
       0.944560512009503,
       0.8743540395354134,
       0.9306073692532968]
```

To get an accurate price for an up-and-out call option using a Monte Carlo simulation, consider the following factors:
Sufficient Number of Paths (N): You need to simulate enough paths to accurately capture the variability and the likelihood of hitting the barrier. The path count should be high enough to ensure the estimate converges. Based on the computational constraints, we were able to compute a price with (N = 10,000) paths. Typically, more paths may be required—on the order of (N = 100,000) or more—to achieve more accuracy, especially for path-dependent options like knock-out calls.
Fine Time Discretization ((t)): A smaller time step helps to capture the barrier touches more accurately. This is particularly important for barrier options where the path can hit the barrier within a very short time frame.

```
[ ]:
```