Computational Finance - Case Study - 3

Sowrya Gali - sg4150

April 2023

1 Question-1

To obtain the parameters for VGSA and Heston models, we have used the NM simplex and BFGS optimization procedures. The cost function is RMSE with equally weighted and inverse spread weighted variants. The data is obtained from the links provided in the documentation. The standard date is selected for each month to compile the data.

1.1 Data Processing

The following snippet 1 details processing the data obtained from the website and extracting out-of-money options.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import fmin, fmin_bfgs
from matplotlib import cm


file_prefix = './data/All/spx_quotedata_'
file_suffix = '.csv'
month_list = ['May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec', 'Jan', 'Feb',
    'Mar', 'Apr']
filenames = [file_prefix + month + file_suffix for month in month_list]


def get_dividend(c,p,S,K,r,T):
    return -np.log((c - p + K * np.exp(-r*T)) / S) / T


def extract_OTM(filename, spot, current_time, rate):
    df = pd.read_csv(filename, skiprows=3)
    df['Expiration Date'] = pd.to_datetime(df['Expiration Date'])
    T = (df['Expiration Date'][0] - current_time).days / 365
    df['Maturity'] = T

    # Setting option price as average of bid and ask prices
    df['CallPrice'] = (df['Bid']+df['Ask'])/2.0
    df['PutPrice'] = (df['Bid.1']+df['Ask.1'])/2.0

    # To get the call and put prices for the nearest price to the spot
    idx = np.argmin(np.abs(df['Strike'] - spot))
    nearest_trade = df.iloc[idx,:]
    df['dividend'] = get_dividend(nearest_trade['CallPrice'], nearest_trade['
    PutPrice'], spot, nearest_trade['Strike'], rate, T)

    # Get Out-of-Money Options -- 20% range
    c = df[['CallPrice', 'Strike', 'Maturity', 'dividend', 'Bid', 'Ask']]
    c = c[(c['Strike'] < spot*1.2) & (c['Strike'] > spot)].reset_index(drop=True)
    c.columns = ['Price', 'Strike', 'Maturity', 'Dividend', 'Bid', 'Ask']

    p = df[['PutPrice', 'Strike', 'Maturity', 'dividend', 'Bid.1', 'Ask.1']]
    p = p[(p['Strike'] > 0.8*spot) & (p['Strike'] < spot)].reset_index(drop=True)
    p.columns = ['Price', 'Strike', 'Maturity', 'Dividend', 'Bid', 'Ask']

    return c,p
```

Listing 1: Data processing

Here the function "get_dividend" extracts calculates the dividend based on the put-call parity.

$$c - p = Se^{-qT} - Ke^{-rT} \tag{1}$$

We take the average of the Bid and Ask price as the price of the option. The dividend is obtained as mentioned before. The consider the basis price for comparison, we take the nearest trade to the strike strike price and obtain options that are within 20% range.

1.2 Characteristic Functions and FFT

We have coded the VGSA and Heston models and their corresponding characteristic functions in the following snippet 2.

```
def VG(x, theta, nu, sig):
    return -np.log(1 - 1j*x*theta*nu + nu*(sig*x)**2/2) / nu


def timechangeCF(u, t, y0, kappa, eta, lamda):
    g = np.sqrt(kappa**2 - 2 * lamda**2 * 1j * u)
    log_numer = eta * t * kappa**2 / lamda ** 2
    power = 2 * kappa * eta / lamda**2
    log_denum = power * np.log(np.cosh(g*t/2) + (kappa/g) * np.sinh(g*t/2))
    B = 2j*u / (kappa + g/np.tanh(g*t/2))
    log_phi = log_numer - log_denum + B * y0
    return np.exp(log_phi)


def generic_CF(u, params, S0, r, q, T, model):

    if model == 'Heston':
        kappa = params[0]
        theta = params[1]
        sigma = params[2]
        rho = params[3]
        v0 = params[4]

        tmp = (kappa - 1j * rho * sigma * u)
        g = np.sqrt((sigma**2) * (u**2 + 1j*u) + tmp**2)
        pow1 = 2 * kappa * theta / (sigma**2)
        numer1 = (kappa * theta * T * tmp) / (sigma**2) + 1j * u * T * (r-q) + 1j *
    u * np.log(S0)
        log_denum1 = pow1 * np.log(np.cosh(g*T/2) + (tmp/g) * np.sinh(g*T/2))
        tmp2 = ((u*u + 1j*u) * v0) / (g / np.tanh(g*T/2) + tmp)
        log_phi = numer1 - log_denum1 - tmp2
        phi = np.exp(log_phi)

    elif model == 'VGSA':
        sigma = params[0]
        nu = params[1]
        theta = params[2]
        kappa = params[3]
        eta = params[4]
        lamda = params[5]

        num = timechangeCF(-1j*VG(u, theta, nu, sigma), T, 1/nu, kappa, eta, lamda)
        denum = timechangeCF(-1j*VG(-1j, theta, nu, sigma), T, 1/nu, kappa, eta,
    lamda) ** (1j * u)
        exp =  np.exp(1j * u * (np.log(S0) + (r-q) * T))
        phi = exp * num / denum

    return phi
```

Listing 2: Characteristic Functions

We then wrote a generic FFT to obtain the prices. The code for FFT is directly taken from the sample code, since that is not the main point of this assignment.

```
def genericFFT(params, S0, K, r, q, T, alpha, eta, n, model):

    N = 2**n

    # step-size in log strike space
    lda = (2*np.pi/N)/eta

    #Choice of beta
#     beta = np.log(S0)-N*lda/2
```

```
    beta = np.log(K)

    # forming vector x and strikes km for m=1,...,N
    km = np.zeros((N))
    xX = np.zeros((N))

    # discount factor
    df = np.exp(-r*T)

    nuJ = np.arange(N)*eta
    psi_nuJ = generic_CF(nuJ-(alpha+1)*1j, params, S0, r, q, T, model) / ((alpha + 1
    j*nuJ)*(alpha+1+1j*nuJ))

    for j in range(N):
        km[j] = beta+j*lda
        if j == 0:
            wJ = (eta/2)
        else:
            wJ = eta
        xX[j] = np.real(np.exp(-1j*beta*nuJ[j])*df*psi_nuJ[j]*wJ)

    yY = np.fft.fft(xX)
    cT_km = np.zeros((N))
    for i in range(N):
        multiplier = np.exp(-alpha*km[i])/np.pi
        cT_km[i] = multiplier*np.real(yY[i])

    return km, cT_km
```

Listing 3: FFT

1.3 Cost Function and Optimizers

We have used two variants of cost functions, they are equally weighted and weighted as inverse of bid-ask spread. We used two different types of optimizers with NM simplex and BFGS. The NM simplex only uses only function values whereas BFGS uses gradients. The following snippet 4 has the code for the cost function.

```
alpha = 1.5
n_FFT = 10
eta = 0.2
# x0, spot, K, rate, q, T, alpha, eta, n_FFT, model
def cost(params, call_df, put_df, weight, model):
    call_modelPrice = np.zeros(len(call_df))
    put_modelPrice = np.zeros(len(put_df))

    for i in range(len(call_df)):
        # use implied dividend from put call parity
        K, q, T = call_df['Strike'][i], call_df['Dividend'][i], call_df['Maturity'][
    i]

        km, cT_km = genericFFT(params, spot_price, K, current_libor, q, T, alpha,
    eta, n_FFT, model)
        call_modelPrice[i] = np.interp(np.log(K), km, cT_km)

    for j in range(len(put_df)):
        K, q, T = put_df['Strike'][j], put_df['Dividend'][j], put_df['Maturity'][j]
        km, cT_km = genericFFT(params, spot_price, K, current_libor, q, T, alpha,
    eta, n_FFT, model)
        put_modelPrice[j] = np.interp(np.log(K), km, cT_km) + K * np.exp(-
    current_libor*T) - spot_price * np.exp(-q*T)

    if weight == 'Equal':
        ans = np.sum((call_df['Price'] - call_modelPrice) ** 2) + np.sum((put_df['
    Price'] - put_modelPrice) ** 2)

```

```
    elif weight == 'inverseSpread':
        call_spread = np.abs(call_df['Bid'] - call_df['Ask'])
        put_spread = np.abs(put_df['Bid'] - put_df['Ask'])
        ans = np.sum((call_df['Price'] - call_modelPrice)**2 / call_spread) + np.sum
    ((put_df['Price'] - put_modelPrice)**2 / put_spread)

    rmse = np.sqrt(ans / (len(call_df) + len(put_df)))
    return rmse

def optimize(algorithm, initial_parameters, call_mrkt_price, put_mrkt_price, weight,
     model):
    result = None
    if algorithm == 'NM':
        result = fmin(cost,
                      initial_parameters,
                      args=(call_mrkt_price, put_mrkt_price, weight, model),
                      ftol=1,
                      full_output=True,
                      disp=True,
                      maxiter=100)
    elif algorithm == 'BFGS':
        result = fmin_bfgs(cost,
                           initial_parameters,
                           args=(call_mrkt_price, put_mrkt_price, weight, model),
                           gtol=1,
                           full_output=True,
                           disp=True,
                           maxiter=100)
    return result
```

Listing 4: Cost Function

1.4 Creating Snapshot

I have selected August month's data for this snapshot. I have taken the current LIBOR rate as 0.0533114. The current spot price is $4108.94 for S&P 500 option. The following snippet 5 contains snapshot and optimization details.

```
# spot price on 11th April 2023
spot_price = 4108.94
current_date = np.datetime64('2023-04-11').astype('int64')

# LIBOR on april 03 2023
current_libor = 0.0533114
SNAPSHOT = 'Aug'
filename = filenames[month_list.index(SNAPSHOT)]
call, put = extract_OTM(filename, spot_price, pd.to_datetime('Tue Apr 11 2023'),
    current_libor)


heston_initial = [2.0, 0.05, 0.30, -0.70, 0.04]
heston_equal_NM = optimize('NM', heston_initial, call, put, 'Equal', 'Heston')
heston_invspread_NM = optimize('NM', heston_initial, call, put, 'inverseSpread', '
    Heston')
heston_equal_BFGS = optimize('BFGS', heston_initial, call, put, 'Equal', 'Heston')
heston_invspread_BFGS = optimize('BFGS', heston_initial, call, put, 'inverseSpread',
     'Heston')


vgsa_initial = [0.3, 0.05, 0.2, 0.1, 0.1, 0.2]
vgsa_equal_NM = optimize('NM', vgsa_initial, call, put, 'Equal', 'VGSA')
vgsa_invspread_NM = optimize('NM', vgsa_initial, call, put, 'inverseSpread', 'VGSA')
vgsa_equal_BFGS = optimize('BFGS', vgsa_initial, call, put, 'Equal', 'VGSA')
vgsa_invspread_BFGS = optimize('BFGS', vgsa_initial, call, put, 'inverseSpread', '
    VGSA')
```

Listing 5: Snapshot and calibration

1.5 Obtained Parameters

Parameters	Equal NM	Inv. Spread NM	Equal BFGS	Inv. Spread BFGS
κ	0.62725943	2.55214716	1.58567736	1.56329621
θ	0.06743303	0.07819582	0.11103702	0.20598782
σ	0.62400565	0.89708291	0.8301472	1.06803211
ρ	-0.74341581	-0.77116244	-0.76917028	-0.77450532
v_0	0.03758794	0.02210765	0.0198931	-0.00824193

Table 1: Heston parameters from calibration

Parameters	Equal NM	Inv. Spread NM	Equal BFGS	Inv. Spread BFGS
σ	0.04244723	0.04328787	-0.10616325	-0.16136585
ν	0.06908975	0.06206301	1.01832753	1.01191276
θ	0.09992668	0.06503928	-0.09330056	-0.10068673
κ	0.13499837	0.15792134	-10.22337921	-10.85637891
η	0.11216314	0.11334961	0.8476369	0.95510057
λ	0.23976073	0.21171196	-0.03785071	0.03940367

Table 2: VGSA parameters from calibration

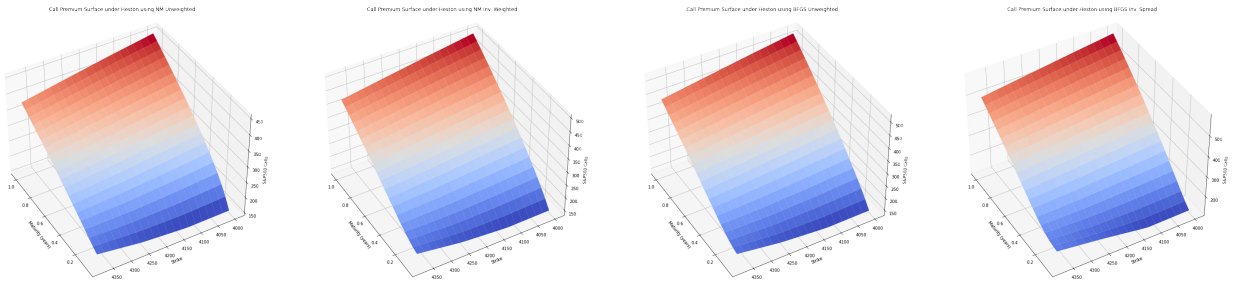
2 Question-2

2.1 Generating call-option premium surface

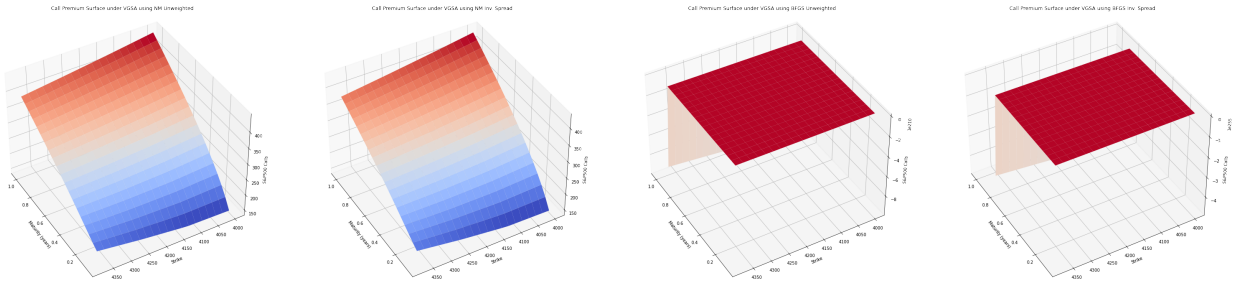
We are using the following code snippet 6 to generate the call-option premium surface.

```
q = call['Dividend'][0]

def build_premium_surface(K, KSteps, T, TSteps, spot_price, params, model, opt, weight):
    strikes = K + 20*np.arange(KSteps)
    times = T + 0.05*np.arange(TSteps)
    prices = np.zeros((KSteps, TSteps))

    for i in range(TSteps):
        T = times[i]
        km, cT_km = genericFFT(params, spot_price, K, current_libor, q, T, alpha, eta, n_FFT, model)
        prices[:, i] = np.interp(np.log(strikes), km, cT_km)

    fig = plt.figure(figsize=(12.,12.))
    ax = fig.add_subplot(111, projection='3d')
    X, Y = np.meshgrid(times, strikes)
    ax.plot_surface(X, Y, prices, cmap=cm.coolwarm)
    ax.set_xlabel('Maturity (years)')
    ax.set_ylabel('Strike')
    ax.set_zlabel('S&P500 Calls')
    ax.view_init(40, 150)
    plt.title('Call Premium Surface under ' + model + ' using ' + opt + ' ' + weight)
    plt.show()

KSteps, TSteps = 20, 20
K0 = 4000
T0 = 0.05
```

Listing 6: Call Option Surface



(a) Heston NM-Simplex Unweighted (b) Heston NM-Simplex Inv. Spread (c) Heston BFGS Unweighted (d) Heston BFGS Inv. Spread



(e) VGSA NM-Simplex Unweighted (f) VGSA NM-Simplex Inv. Spread (g) VGSA BFGS Unweighted (h) VGSA BFGS Inv. Spread

Figure 1: Call Option Premium Surfaces for August options

The following figures were generated. An interesting observation is that call option surface generated for VGSA and BFGS are bad. The reasons for this are discussed in the next section.

2.2 Local Volatility Surface

The local volatility surface is obtained using the following formula.

$$\sigma(K,T) = \left(2 \frac{\frac{\partial C}{\partial T} + q(T)C + (r(T) - q(T))K\frac{\partial C}{\partial K}}{K^2 \frac{\partial^2 C}{\partial K^2}} \right)^{\frac{1}{2}} \tag{2}$$

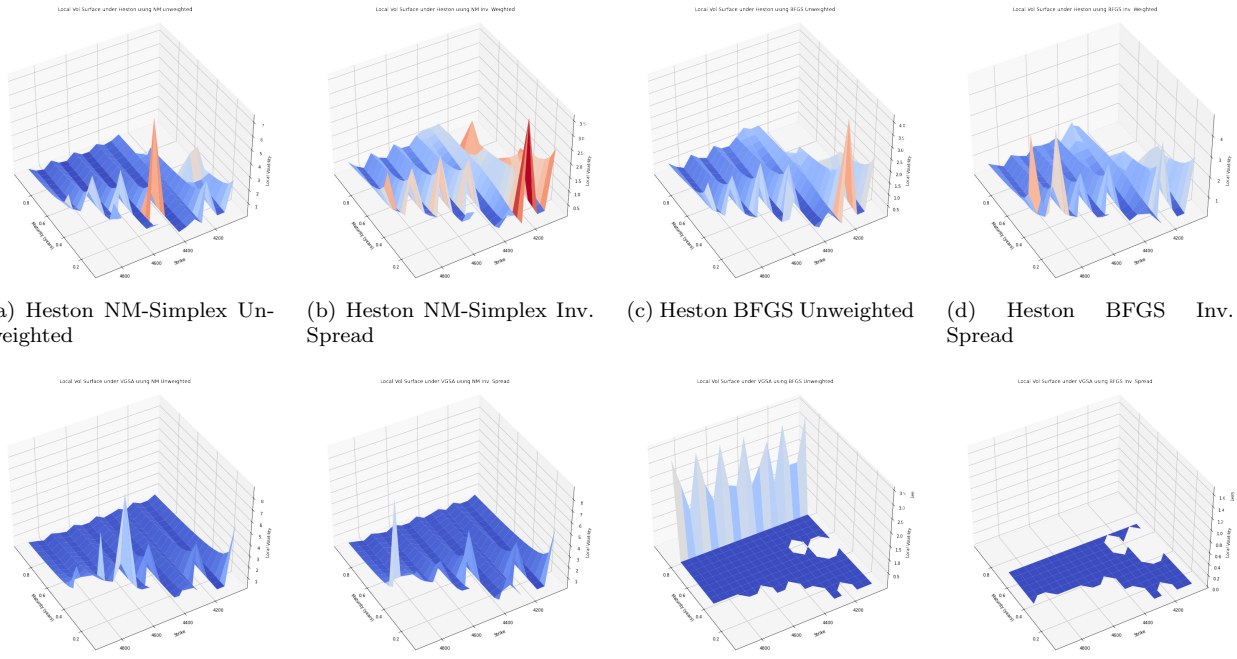
The following code snippet 7 calculates the Local Volatility Surface.

```
def LVS(K0, ksteps, T0, tsteps, spot_price, params, model, opt, weight):
    strikes = K0 + 50 * np.arange(ksteps)
    times = T0 + 0.05 * np.arange(tsteps)
    prices = np.zeros((ksteps, tsteps))

    for i in range(tsteps):
        T = times[i]
        km, cT_km = genericFFT(params, spot_price, K0, current_libor, q, T, alpha,
    eta, n_FFT, model)
        prices[:, i] = np.interp(np.log(strikes), km, cT_km)

    # use finite difference to estimate the derivative
    localVol = np.zeros((ksteps-2, tsteps-2))
    for i in range(1, tsteps-1):
        for j in range(1, ksteps-1):
            dt = (prices[j, i+1] - prices[j, i-1]) / (2*0.05)
            dk = (prices[j+1, i] - prices[j-1, i]) / (2*50)
            dk_second = (prices[j+1, i] - 2*prices[j, i] + prices[j-1, i]) / (50**2)
            c = prices[j, i]
```

```
            localVol[j-1, i-1] = np.sqrt(2 * (dt + q*c + (current_libor-q)*strikes[j
]*dk) / (strikes[j]**2 * dk_second))

# print(localVolatility)

fig = plt.figure(figsize=(12.,12.))
ax = fig.add_subplot(111, projection='3d')
X, Y = np.meshgrid(times[1:tsteps-1], strikes[1:ksteps-1])

ax.plot_surface(X, Y, localVol, cmap=cm.coolwarm)
ax.set_xlabel('Maturity (years)')
ax.set_ylabel('Strike')
ax.set_zlabel('Local Volatility')
ax.view_init(40, 150)
plt.title('Local Vol Surface under ' + model + ' using ' + opt + ' ' + weight)
plt.show()
```

Listing 7: Local Volatility Surface

The following figure shows the volatility surface generated for the August month's data. As expected



(a) Heston NM-Simplex Unweighted

(b) Heston NM-Simplex Inv. Spread

(c) Heston BFGS Unweighted

(d) Heston BFGS Inv. Spread

(e) VGSA NM-Simplex Unweighted

(f) VGSA NM-Simplex Inv. Spread

(g) VGSA BFGS Unweighted

(h) VGSA BFGS Inv. Spread

Figure 2: Local Volatility Surfaces for August options

from the option premium surface, the local volatility surface for the VGSA and BFGS combination are bad.

3 Question-3

3.1 Comparison across models

- The VGSA's local volatility surface is smoother than that of Heston's except for those obtained through BFGS.
- The reason for BFGS being failed is due to its dependency on the gradients. More explanation in the next subsection, where optimization schemes are compared.

- In terms of the range of the local volatility, Heston model is much smaller than that of VGSA model.
- The nature of VGSA process is to incorporate jumps in the price process. Therefore, we should expect a higher the vol in VGSA model.

3.2 Comparison across optimization schemes

- Optimizing based on inverse spread have lower volatility compared to unweighted.
- This is anticipated because if the bid ask spread is high, the option is not traded frequently.
- The price of such options should be sensitive to the parameters like maturity and strike price and hence should be given less importance to make volatility surface smooth.

3.3 Comparison across optimizers

- In Heston, the BFGS worked better than the NM-Simplex, as we can from the figure. The volatility is lower and surface is much smoother.
- However, in the VGSA scheme the BFGS generated calibrated to wrong parameters.
- The primary reason for this failure is using gradients of VGSA characteristic functions.
- BFGS uses the line search, which tries to find a step size where the approximations in BFGS are still valid. When the Hessian of the function or its gradient are ill-behaved in some way, the bracketed step size could be computed as zero, even though the gradient is non-zero.