option_pricing_notebook

December 23, 2022

1 Financial Engineering and Risk Management Part III

1.1 Option Pricing Notebook

This notebook serves as a guide for answering the programming questions of the quiz "Option Pricing Assignment Part IV". Most of the auxilliary functions and the basic structure of the code have been provided and you will only be required to code some missing parts.

After completing this notebook, you will be able to price European put option using either numerical integration or FFT. The second approach is useful since you only need to specify the model's characteristic function, instead of the density which may note be readily available.

Most of the code is similar to the one used by Prof. Hirsa in the videos. We invite you to carefully study the video before going through the notebook.

Once that you are confident that the notebook is running correctly, please input your answers manually on the quiz "Option Pricing Assignment Part IV".

If you wish to run this notebook (as it is) on your local computer you will need to have installed Python 3.6, Jupyter notebooks, and the following Python packages:

- numpy
- time

1.1.1 Import Python modules

```
In [ ]: import numpy as np # for fast vector computations
        from time import time # to obtain the running time of some functions
```

1.1.2 Pricing puts using via numerical integration

Questions 9, 10, 11, and 12 of the quiz "Option Pricing Assignment Part III" For these questions we assume that the stock price follows a lognormal distribution. The following is a function that computes the lognormal density.

```
In [ ]: def logNormal(S, r, q, sig, S0, T):
            ''' Computes the log normal density function for the stock price. '''
            f = np.exp(-0.5*((np.log(S/S0)-(r-q-sig**2/2)*T)/(sig*np.sqrt(T)))**2) / (sig*S*np
            return f
```

Warning: be careful when evaluating this function at $S = 0.0$. The theoretical value should be $f = 0.0$, but the function above will give you a warning and a NaN value. A quick fix is to evaluate it at $S = \epsilon$, where ϵ is a small positive number such as 1e-8.

We want to estimate the price of a European put with the following parameters:

```
In [ ]: # Parameters for put
        S0 = 100
        K = 90
        r = 0.04
        q = 0.02
        sig = 0.25
        T = 1.0
```

Now, we will use the numerical integration method of Module 3 to price this put. For this, complete the following function that takes as arguments the above parameters and a positive integer N which is the number of grid points.

```
In [ ]: def numerical_integral_put(r, q, S0, K, sig, T, N):
            ''' Numerical integration for puts. '''

            # temporary value for output
            eta = 0.0 # spacing of the grid
            priceP = 0.0 # price of the put

            ###############################################################
            ###############################################################
            # your code starts here
            ###############################################################

            ###############################################################
            # your code ends here
            ###############################################################
            ###############################################################

            return eta, priceP
```

We use the previous function to price the put for different values of $N = 2^n, n = 1, 2, \ldots, 15$.

```
In [ ]: # vector with all values of n
        n_min = 1
        n_max = 15
        n_vec = np.arange(n_min, n_max + 1, dtype=int)
```

```
In [ ]: # compute the numerical integration for each value of n
        start_time = time()
        # will store the results in vectors
        eta_vec = np.zeros(n_max)
        put_vec = np.zeros(n_max)
        for i in range(n_max):
```

```
        N = 2** n_vec[i]
        ################################################################
        ################################################################
        # your code starts here
        ################################################################

        ################################################################
        # your code ends here
        ################################################################
        ################################################################
    end_time = time()
    print('Elapsed time (seconds): ' + str(end_time - start_time))
```

We print a table with the results.

```
In [ ]: # print a table with the numerical integration for each value of n
        print('N\teta\tP_0')
        for i in range(n_max):
            print('2^%i\t%.3f\t%.4f' % (n_vec[i], eta_vec[i], put_vec[i]))
```

Question 9 Input manually on the quiz page.

```
In [ ]: # eta, when n = 3
        n_question = 3
        # remember that Python indices start at 0
        print(eta_vec[n_question-1])
```

Question 10 Input manually on the quiz page.

```
In [ ]: # eta, when n = 10
        n_question = 10
        # remember that Python indices start at 0
        print(eta_vec[n_question-1])
```

Question 11 Input manually on the quiz page.

```
In [ ]: # put, when n = 4
        n_question = 4
        # remember that Python indices start at 0
        print(put_vec[n_question-1])
```

Question 12 Input manually on the quiz page.

```
In [ ]: # put, when n = 15
        n_question = 15
        # remember that Python indices start at 0
        print(put_vec[n_question-1])
```

1.1.3 Pricing puts using via FFT

Questions 13, 14, 15 of the quiz "Option Pricing Assignment Part III" In the videos, we saw how to price call options using FFT. The extension to put options is straighforward and we refer you to "Option Pricing Assignment Part II" for more information.

We first need a function that computes the characteristic function of the log-stock price for 3 different models: BMS, Heston, and VG (each one with different sets of parameters). The function below will do the job! The part of Heston and VG is already complete, you just need to implement the BMS part.

```
In [ ]: def generic_CF(u, params, S0, r, q, T, model):
            ''' Computes the characteristic function for different models (BMS, Heston, VG). '

            if (model == 'BMS'):
                # unpack parameters
                sig = params[0]
                # cf
                ################################################################
                ################################################################
                # your code starts here
                ################################################################

                ################################################################
                # your code ends here
                ################################################################
                ################################################################
            elif(model == 'Heston'):
                # unpack parameters
                kappa  = params[0]
                theta  = params[1]
                sigma  = params[2]
                rho    = params[3]
                v0     = params[4]
                # cf
                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 + 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 == 'VG'):
                # unpack parameters
                sigma  = params[0];
                nu     = params[1];
                theta  = params[2];
```

```
        # cf
        if (nu == 0):
            mu = np.log(S0) + (r-q - theta -0.5*sigma**2)*T
            phi  = np.exp(1j*u*mu) * np.exp((1j*theta*u-0.5*sigma**2*u**2)*T)
        else:
            mu  = np.log(S0) + (r-q + np.log(1-theta*nu-0.5*sigma**2*nu)/nu)*T
            phi = np.exp(1j*u*mu)*((1-1j*nu*theta*u+0.5*nu*sigma**2*u**2)**(-T/nu))

    return phi
```

We now provide you with a function that uses FFT to price European option for any of the 3 models we have discussed. The same function works for both calls and puts (you should ask yourself: how and why?). The function is complete, you don't need to add any code. It return two vectors, one with the log-strikes and one with option prices.

```
In [ ]: def genericFFT(params, S0, K, r, q, T, alpha, eta, n, model):
            ''' Option pricing using FFT (model-free). '''

            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 # the log strike we want is in the middle of the array
            beta = np.log(K) # the log strike we want is the first element of the array

            # 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

            km = beta + lda * np.arange(N)
            w = eta * np.ones(N)
            w[0] = eta / 2
            xX = np.exp(-1j * beta * nuJ) * df * psi_nuJ * w

            yY = np.fft.fft(xX)
            cT_km = np.zeros(N)
            multiplier = np.exp(-alpha * km) / np.pi
            cT_km = multiplier * np.real(yY)

            return km, cT_km
```

We want to estimate a European put with the following parameters:

```
In [ ]: # parameters
        S0 = 100
        K = 80
        r = 0.05
        q = 0.01
        T = 1.0
```

For the 3 models, we consider the same values of α, η, and n.

```
In [ ]: # parameters for fft
        eta_vec = np.array([0.1, 0.25])
        n_vec = np.array([6, 10])
        alpha_vec = np.array([-1.01, -1.25, -1.5, -1.75, -2., -5.])
        num_prices = len(eta_vec) * len(n_vec) * len(alpha_vec)
```

We write a function that given a model and the above parameters will compute the put price.

```
In [ ]: def price_all_puts(params, S0, K, r, q, T, model, alpha_vec, eta_vec, n_vec):
            num_prices = len(eta_vec) * len(n_vec) * len(alpha_vec)
            # output is a matrix, the columns correspond to eta, n, alpha, and put price
            put_matrix = np.zeros([num_prices, 4])
            i = 0
            for eta in eta_vec:
                for n in n_vec:
                    for alpha in alpha_vec:
                        # pricing via fft
                        put = 0 # store here the put price
                        ###############################################################
                        ###############################################################
                        # your code starts here
                        ###############################################################

                        ###############################################################
                        # your code ends here
                        ###############################################################
                        ###############################################################
                        put_matrix[i] = np.array([eta, n, alpha, put])
                        i += 1
            return put_matrix
```

BMS These are the parameters for the model.

```
In [ ]: # model-specific parameters
        mod = 'BMS'
        sig = 0.3
        params = [sig]
```

```
In [ ]: start_time = time()
        bms_matrix = price_all_puts(params, S0, K, r, q, T, mod, alpha_vec, eta_vec, n_vec)
        end_time = time()
        print('Elapsed time (seconds): ' + str(end_time - start_time))
```

```
In [ ]: # print results in table form
        print('Model = ' + mod)
        print('eta\tN\talpha\tput')
        for i in range(num_prices):
            print('%.2f\t2^%i\t%.2f\t%.4f' % (bms_matrix[i,0], bms_matrix[i,1], bms_matrix[i,2
```

Question 13 Input manually on the quiz page.

After inspecting the numerical results for different values of α, η, n, what is the price of the European put using the BMS model?

```
In [ ]: # solution
```

Heston

```
In [ ]: # model-specific parameters
        mod = 'Heston'
        kappa = 2.
        theta = 0.05
        lda = 0.3
        rho = -0.7
        v0 = 0.04
        params = [kappa, theta, lda, rho, v0]
```

```
In [ ]: start_time = time()
        heston_matrix = price_all_puts(params, S0, K, r, q, T, mod, alpha_vec, eta_vec, n_vec)
        end_time = time()
        print('Elapsed time (seconds): ' + str(end_time - start_time))
```

```
In [ ]: # print results in table form
        print('Model = ' + mod)
        print('eta\tN\talpha\tput')
        for i in range(num_prices):
            print('%.2f\t2^%i\t%.2f\t%.4f' % (heston_matrix[i,0], heston_matrix[i,1], heston_ma
```

Question 14 Input manually on the quiz page.

After inspecting the numerical results for different values of α, η, n, what is the price of the European put using the Heston model?

```
In [ ]: # solution
```

VG

```
In [ ]: # model-specific parameters
        mod = 'VG'
        sigma = 0.3
        nu = 0.5
        theta = -0.4
        params = [sigma, nu, theta]
```

```
In [ ]: start_time = time()
        vg_matrix = price_all_puts(params, S0, K, r, q, T, mod, alpha_vec, eta_vec, n_vec)
        end_time = time()
        print('Elapsed time (seconds): ' + str(end_time - start_time))
```

```
In [ ]: # print results in table form
        print('Model = ' + mod)
        print('eta\tN\talpha\tput')
        for i in range(num_prices):
            print('%.2f\t2^%i\t%.2f\t%.4f' % (vg_matrix[i,0], vg_matrix[i,1], vg_matrix[i,2],
```

Question 15 Input manually on the quiz page.

After inspecting the numerical results for different values of α, η, n, what is the price of the European put using the VG model?

```
In [ ]: # solution
```