FE621 Assignment3

Yuzhu Jiang

November 1, 2020

Part 1

For the assignment, the data were retrieved from Yahoo Finance using Python.
let $x = lnS$, $v = r - div - \sigma^2/2$

$$\frac{\partial C}{\partial t} = \sigma^2 \frac{\partial^2 C}{2 * \partial^2 x^2} + v \frac{\partial C}{\partial x} - rC$$

This is an equation with constant coefficient, means that they do not depends on x or t.
For call option, when S is large, $\frac{\partial C}{\partial S} = 1$
For call option, when S is small, $\frac{\partial C}{\partial S} = 0$
Explicit Finite Difference:
We use a forward difference for $\frac{\partial C}{\partial x}$, central difference for $\frac{\partial^2 C}{\partial x^2}$ and $\frac{\partial C}{\partial x}$. Therefore, in terms of the grid, we obtain:

$$-\frac{C_{i+1,j} - C_{i,j}}{\delta_t} = \frac{\sigma^2}{2} * \frac{C_{i+1,j+1} - 2C_{i+1,j} + C_{i+1,i-1}}{\Delta_x^2} + v * \frac{C_{i+1,j+1} - C_{i+1,j-1}}{2\Delta X} - rC_{i+1,j}$$

which can be written as:

$$C_{i,j} = pu * C_{i+1,j+1} + pm * C_{i+1,j} + pd * C_{i+1,j-1}$$

$$pm = (1 - \Delta_t(\frac{\sigma^2}{\Delta x^2}) - r\Delta_t)$$

$$pu = \Delta_t(\frac{\sigma^2}{2\Delta x^2} + \frac{v}{\Delta x})$$

$$pd = \Delta_t(\frac{\sigma^2}{2\Delta x^2} - \frac{v}{2\Delta x})$$

Implicit Finite Difference:

$$C_{i+1,j} = pu * C_{i,j+1} + pm * C_{i,j} + pd * C_{i,j-1}$$

$$pm = 1 + \Delta_t(\frac{\sigma^2}{\Delta x^2}) + r\Delta_t$$

$$pu = -0.5\Delta_t(\frac{\sigma^2}{\Delta x^2} + \frac{v}{\Delta x})$$

$$pd = -0.5\Delta_t(\frac{\sigma^2}{\Delta x^2} - \frac{v}{2\Delta x})$$

$$\lambda_U = S_{i,Nj} - S_{i,Nj-1}$$

$$\lambda_L = 0$$

It can be written in matrix form:

$$\begin{bmatrix} \lambda_U \\ C_{i+1,Nj-1} \\ C_{i+1,Nj-2} \\ ... \\ \lambda_L \end{bmatrix} = \begin{bmatrix} 1 & -1 & ... & ... & ... & ... & 0 \\ pu & pm & pd & 0 & ... & ... & 0 \\ 0 & pu & pm & pd & 0 & ... & 0 \\ ... & ... & ... & ... & ... & ... & ... \\ 0 & ... & ... & ... & ... & 1 & -1 \end{bmatrix} \begin{bmatrix} C_{i,Nj} \\ C_{i,Nj-1} \\ C_{i,Nj-2} \\ ... \\ C_{i,-Nj} \end{bmatrix}$$

Crank Nicolson Finite Difference:

$$-\frac{C_{i+1,j} - C_{i,j}}{\Delta_t} = 0.5\sigma^2(\frac{C_{i+1,j+1} - 2C_{i+1,j-1} + C_{i+1,j-1} + C_{i,j+1} - 2C_{i,j} + C_{i,j-1}}{2\Delta x^2}) + v(\frac{C_{i+1,j+1} - C_{i+1,j-1} + C}{4\Delta x}$$

$$-pu * C_{i+1,j+1} - (pm-2)C_{i+1,j} - pdC_{i+1,j-1}$$

$$pu = -0.25\Delta t(\sigma^2/\Delta x^2 + v/\Delta x)$$

$$pm = 1 + \Delta t(\sigma^2/\Delta x^2 + v/\Delta x)$$

$$pd = -0.25\Delta t(\sigma^2/\Delta x^2 - v/\Delta x)$$

It can be written in matrix form:

$$\begin{bmatrix} 1 & -1 & ... & ... & ... & ... & 0 \\ -pu & -pm+2 & -pd & 0 & ... & ... & 0 \\ 0 & -pu & -pm+2 & -pd & 0 & ... & 0 \\ ... & ... & ... & ... & ... & ... & ... \\ 0 & ... & ... & ... & ... & 1 & -1 \end{bmatrix} \begin{bmatrix} \lambda_U \\ C_{i+1,Nj-1} \\ C_{i+1,Nj-2} \\ ... \\ C_{\lambda_L} \end{bmatrix} = \begin{bmatrix} 1 & -1 & ... & ... & ... & ... & 0 \\ pu & pm & pd & 0 & ... & ... & 0 \\ 0 & pu & pm & pd & 0 & ... & 0 \\ ... & ... & ... & ... & ... & ... & ... \\ 0 & ... & ... & ... & ... & 1 & -1 \end{bmatrix}$$

The Implementation in Python as following:

```
import numpy as np
from numpy import exp
def explict_finite(S,K,T,sig,r,div,N,Nj,dx, CP, AE):
    #precompute consts
    dt = T/N
    if CP == "call":
        nu = r - div - sig**2/2
    elif CP == "put":
        nu = r + div - sig**2/2
    edx = exp(dx)

    pu = 0.5*dt*( (sig/dx)**2 + nu/dx )
    pd = 0.5*dt*( (sig/dx)**2 - nu/dx )
    pm = 1-dt*(sig/dx)**2 - r*dt
```

```
    #initialize asset prices at maturity
    stock = np.zeros((2*Nj+1,1))
    option = np.zeros((2*Nj+1, N))
    stock[2*Nj][0] = S*exp(-Nj*dx)
    for j in range(2*Nj-1, -1, -1):
        stock[j][0] = stock[j+1][0]*edx

    #initialize option value at maturity
    for i in range(2*Nj, -1,-1):
        if CP == "call":
            option[i][-1] = max(0, stock[i][0] - K)
        elif CP == "put":
            option[i][-1] = max(0, K - stock[i][0])

    #step back through lattice
    for x in range(N-2,-1,-1):
        for y in range(1,2*Nj,1):
            option[y][x] = pu*option[y-1][x+1] + pm*option[y][x+1]+ pd*option[y+1][x+1]

    #compute boundary condition
        if CP == "call":
            option[0,x],option[2*Nj,x] = option[1,x]+stock[0][0] - stock[1][0],0
        elif CP == "put":
            option[2*Nj,x],option[0,x] = option[2*Nj-1,x]+stock[2*Nj-1][0] - stock[2*Nj]
        #apply early exercise for American Put
        if AE == "A" and CP == "put":
            for j in range(2*Nj-1, -1,-1):
                option[j][x] = max(option[j][x], K - stock[j][0])
    return option[Nj][0]

def implict_finite(S,K,T,sig,r,div,N,Nj,dx, CP, AE):
    #precompute consts
    dt = T/N
    if CP == "call":
        nu = r - div - sig**2/2
    elif CP == "put":
        nu = r + div - sig**2/2
    edx = exp(dx)
    pu = -0.5*dt*( (sig/dx)**2 + nu/dx )
    pd = -0.5*dt*( (sig/dx)**2 - nu/dx )
    pm = 1 + dt*(sig/dx)**2 + r*dt

    #initialize asset prices at maturity
    stock = np.zeros((2*Nj+1,1))
```

```
    option = np.zeros((2*Nj+1, N))
    stock[2*Nj][0] = S*exp(-Nj*dx)

    for j in range(2*Nj-1, -1, -1):
        stock[j][0] = stock[j+1][0]*edx

    #initialize option value at maturity
    for j in range(2*Nj, -1,-1):
        if CP == "call":
            option[j][N-1] = max(0, stock[j][0]-K)
        elif CP == "put":
            option[j][N-1] = max(0, K-stock[j][0])

    M = np.zeros((2*Nj+1, 2*Nj+1))
    M[0][0] = M[-1][-2] = 1
    M[0][1] = M[-1][-1] = -1
    for i in range(1, 2*Nj):
        for j in range(i-1, i+2):
            if j == i-1:
                M[i,j] = pu
            elif j == i:
                M[i,j] = pm
            elif j == i+1:
                M[i,j] = pd
    M_inv = np.linalg.inv(M)
    #compute boundary condition
    if CP == "call":
        lambda_U,lambda_L = (stock[0][0] - stock[1][0]),0
    elif CP == "put":
        lambda_L,lambda_U = (stock[2*Nj-1][0] - stock[2*Nj][0]),0
    for i in range(N-1,0,-1):
        option[0, i] = lambda_U
        option[-1,i] = lambda_L
        option[:,i-1] = np.dot(M_inv, option[:,i])
        if AE == "A" and CP == "put":
            for j in range(2*Nj-1, -1,-1):
                option[j][i-1] = max(option[j][i-1], K - stock[j][0])
    return option[Nj][0]

def CN_finite(S,K,T,sig,r,div,N,Nj,dx, CP, AE):
    #precompute consts
    dt = T/N
    if CP == "call":
        nu = r - div - 0.5*sig**2
    elif CP == "put":
```

```
    nu = r + div - 0.5*sig**2
edx = exp(dx)
pu = -0.25*dt*( (sig/dx)**2 + nu/dx )
pd = -0.25*dt*( (sig/dx)**2 - nu/dx  )
pm = 1 + 0.5*dt*(sig/dx)**2 + r*dt/2
#initialize asset prices at maturity
stock = np.zeros((2*Nj+1,1))
option = np.zeros((2*Nj+1, N))
stock[2*Nj][0] = S*exp(-Nj*dx)
for j in range(2*Nj-1, -1, -1):
    stock[j][0] = stock[j+1][0]*edx
#initialize option value at maturity
for j in range(2*Nj, -1,-1):
    if CP == "call":
        option[j][N-1] = max(0, stock[j][0]-K)
    elif CP == "put":
        option[j][N-1] = max(0, K-stock[j][0])
#compute boundary condition
if CP == "call":
    lambda_U,lambda_L = -(stock[0][0] - stock[1][0]),0
elif CP == "put":
    lambda_L,lambda_U = -(stock[2*Nj-1][0] - stock[2*Nj][0]),0
M = np.zeros((2*Nj+1, 2*Nj+1))
MA = np.zeros((2*Nj+1, 2*Nj+1))
M[0][0] = M[-1][-2] = MA[0][0] = MA[-1][-2] = 1
M[0][1] = M[-1][-1] = MA[0][0] = MA[-1][-2] = -1
for i in range(1, 2*Nj):
    for j in range(i-1, i+2):
        if j == i-1:
            M[i,j] = pu
            MA[i,j] = -pu
        elif j == i:
            M[i,j] = pm
            MA[i,j] = -pm +2
        elif j == i+1:
            M[i,j] = pd
            MA[i,j] = -pd
M_inv = np.linalg.inv(M)
for i in range(N-1,0,-1):
    option[0, i] = lambda_U
    option[-1,i] = lambda_L
    option[:,i-1] = np.dot(np.dot(M_inv, MA),option[:,i])
    if AE == "A" and CP == "put":
        for j in range(2*Nj-1, -1,-1):
            option[j][i-1] = max(option[j][i-1], K - stock[j][0])
```

```
    return option[Nj][0]
```

The accuracy of these two Finite Difference Method is $O(\Delta x^2 + \Delta t)$. Yet, unlike Explicit Finite Method, Implicit Finite Difference Method is unconditionally stable and convergent. So we can increase the accuracy by making step smaller without sacrificing the speed.
For Explicit Finite Difference Method, $\Delta x \leq \sigma\sqrt{3\Delta t}$. We let delta x equal to the boundary to larger the accuracy. The corresponding Δt is $\frac{error}{3\sigma^2+1}$. A reasonable range of asset price at maturity date of the option is 3 standard deviation either side of the mean and a reasonable number of asset price value would be 2Nj+1. In the example given. For Explicit Method: The goal is to get the boundary condition for dx and dt when

$$dx^2 + dt <= 0.0001$$

Knowing that

$$dx >= \sigma\sqrt{3dt}$$

we have

$$0.12dt^2 + dt \leq 0.0001$$

Solving the equations we have the following boundaries:

$$dx >= 0.010955621149237727$$

$$dt <= 0.0010002136230468747$$

$$Nj = 55$$

$$N = 1000$$

```
bisection(0,0.3,lambda x: 12*x**2 + 100*x - 0.1, times = 10000)
dx = 0.2*np.sqrt(3*dt)

Output:
dt is  0.0010002136230468747
dx is  0.010955621149237727
```

All variables can be expressed as Nj as the following (Assuming nsd = 6):

$$N = \frac{(2Nj+1)^2}{12}$$

$$dt = \frac{1}{N}$$

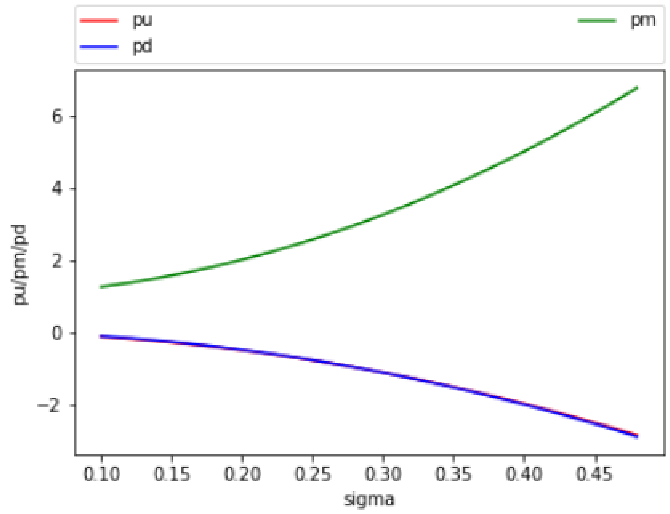
$$dx = \sigma\sqrt{3dt}$$

For part (f), here is the thresholds I got from iteration:

Type	Nj	N	dt	dx
Explicit-Call	508	86191	1.1602173087019198e-05	0.00118
Explicit-Put	306	31519	3.172714653975808e-05	0.00195
Implicit-Call	530	93810	1.0659834896923839e-05	0.00113
Implicit-Put	542	98102	1.0193463441568095e-05	4.496602883132097
CN-Call	422	59502	1.6806134238997232e-05	0.00142
CN-Put	430	61777	1.6187319663141876e-05	0.00139

```
def refine_para(method,CP):
    Nj = 550
    sig,AE = 0.2,"E"
    S,K,T = 100,100,1
    r,div =0.06,0.02
    error = 0.0001
    dx = 6*sig/(2*Nj+1)
    dt = (dx/sig)**2/3
    N = int(1/dt)
    dx = 0.2*np.sqrt(3*dt)
    if CP == "put":
        BS = vanilla_Euro_option(100, 100, 1, 0.06, 0.2, CP)
    elif CP == "call":
        BS = vanilla_Euro_option(100, 100, 1, 0.06, 0.2, CP)
    if method == "EFD":
        res = explict_finite(100,100,1,0.2,0.06,0.0,N,Nj,dx, "call", "E")
        while abs(res -BS) >0.0001:
            Nj += 10
            res = explict_finite(100,100,1,0.2,0.06,0.0,N,Nj,dx, "call", "E")
    elif method == "IFD":
        res = implict_finite(100,100,1,0.2,0.06,0.0,N,Nj,dx, "call", "E")
        while abs(res -BS) >0.0001:
            Nj += 10
            res = implict_finite(100,100,1,0.2,0.06,0.0,N,Nj,dx, "call", "E")
    elif method == "CNFD":
        res = CN_finite(100,100,1,0.2,0.06,0.0, N,Nj,dx,"call", "E")
        while abs(res -BS) > 0.0001:
            Nj += 10
            res = CN_finite(100,100,1,0.2,0.06,0.0,N,Nj,dx, "call", "E")
    return Nj
refine_para("EFD","call")
```

For part (g), the graph is as following:




```
import matplotlib.pyplot as plt
def get_pupmpd(S,K,T,sig,r,div,N,Nj,dx, CP, AE):
    #precompute consts
    dt = T/N
    if CP == "call":
        nu = r - div - sig**2/2
    elif CP == "put":
        nu = r + div - sig**2/2
    edx = exp(dx)
    pu = -0.5*dt*( (sig/dx)**2 + nu/dx )
    pd = -0.5*dt*( (sig/dx)**2 - nu/dx )
    pm = 1 + dt*(sig/dx)**2 + r*dt
    return sig, pu, pm, pd

res_u, res_d, res_m = [],[],[]
x_ray = [i for i in np.arange(0.1,0.5,0.02)]
for i in x_ray:
    res_u.append(get_pupmpd(100,100,1,i,0.06,0.02,1000,55,0.012, "call", "E")[1])
    res_m.append(get_pupmpd(100,100,1,i,0.06,0.02,1000,55,0.012, "call", "E")[2])
    res_d.append(get_pupmpd(100,100,1,i,0.06,0.02,1000,55,0.012, "call", "E")[3])
plt.plot(x_ray, res_u, color = "red", label = "pu")
plt.plot(x_ray, res_d, color = "blue", label = "pd")
plt.plot(x_ray, res_m, color = "green", label = "pm")
plt.xlabel("sigma")
plt.ylabel("pu/pm/pd")
plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc='lower left',
           ncol=2, mode="expand", borderaxespad=0.)
```

For part (h), the code for these 3 methods calculation were attached in the previous part. And the result is shown below:

P/C	Explicit	Implicit	Crank-Nicolson
Call	9.720284696907019	9.715286980309013	9.713903754874911
Put	4.501455042248694	4.495821560765051	4.496602883132097

```
Code:
print("Explict Method Call, ",explict_finite(100,100,1,0.2,0.06,0.02,1000,55,0.013, "cal
print("Implict Method Call, ",implict_finite(100,100,1,0.2,0.06,0.02,1000,55,0.017, "cal
print("CN Method Call, ",CN_finite(100,100,1,0.2,0.06,0.02,1000,55,0.02, "call", "E"))
print("BS Model Call, ", vanilla_Euro_option(100, 100, 1, 0.04, 0.2, option = 'call'))
print("Explict Method Put, ",explict_finite(100,100,1,0.2,0.06,0.02,1000,55,0.015, "put"
print("Implict Method Put, ",implict_finite(100,100,1,0.2,0.06,0.02,1000,55,0.02, "put",
print("CN Method Put, ",CN_finite(100,100,1,0.2,0.06,0.02,1000,55,0.02, "put", "E"))
print("BS Model Call, ", vanilla_Euro_option(100, 100, 1, 0.08, 0.2, option = 'put'))
```

The result generated from these three methods are quite similar. Yet, Crank Nicolson is slower than the other two methods.
For part (i), the results are below:

```
def get_greek(S,K,T,sig,r,div,N,Nj,dx, CP, AE,greek):
    edx = exp(dx)
    S0 = S
    S1 = S/edx
    S2 = S*edx
    dt = T/N
    C0 = explict_finite(S,K,T,sig,r,div,N,Nj,dx, CP, AE)
    C1 = explict_finite(S/edx,K,T,sig,r,div,N,Nj,dx, CP, AE)
    C2 = explict_finite(S*edx,K,T,sig,r,div,N,Nj,dx, CP, AE)
    Ct = explict_finite(S,K,T-dt,sig,r,div,N,Nj,dx, CP, AE)
    Cv = explict_finite(S,K,T,sig+.001*sig,r,div,N,Nj,dx, CP, AE)
    Cr = explict_finite(S,K,T,sig,r+0.001,div,N,Nj,dx, CP, AE)
    if greek == "delta": return (C2-C1)/(S*edx - S/edx)
    if greek == "gamma": return ((C2-C0)/(S2-S0)-(C0-C1)/(S0-S1))/(0.5*(S2-S1))
    if greek == "theta": return -(C0-Ct)/dt
    if greek == "vega": return (Cv-C0)/(0.001*sig)
    if greek == "rho": return (Cr - C0)/0.001
print("Delta is ", get_greek(100,100,1,0.2,0.06,0.02,300,675,0.02, "call", "E", "delta")
print("Gamma is ",get_greek(100,100,1,0.2,0.06,0.02,300,675,0.02, "call", "E", "gamma"))
print("Theta is ", get_greek(100,100,1,0.2,0.06,0.02,300,675,0.02, "call", "E", "theta")
print("Vega is ",get_greek(100,100,1,0.2,0.06,0.02,300,675,0.02, "call", "E", "vega"))
# print("Rho is ",get_greek(100,100,1,0.2,0.06,0.02,300,675,0.02, "call", "E", "rho"))
```

```
Output:
Delta is  0.6056341693501248
Gamma is  0.018724159330690455
Theta is  -5.578108393562431
Vega is  37.39551467992541
```

Part II

Under the risk neutral probability measure,

$$dS = rSdt + \sigma Sdz$$

$$dlnS = (r - \frac{1}{2}\sigma^2)dt + \sigma dz$$

let $st = ln(S)$, we have $st \sim \mathcal{N}(lnS_0 + t(r - 0.5\sigma^2), \sigma^2 t)$

$$\phi_u = \int e^{ius} q_T(s) ds = E[\exp ius t]$$

Thus the characteristic function of st is:

$$\phi_u = \exp\left(iu(lnS_0 + t(r - 0.5\sigma^2)) - \frac{1}{2}u^2\sigma^2 t\right)$$

To be consistent with the lecture, we choose:

$$\alpha = 1.75$$

$$\Phi_T(v) = \frac{e^{-rT}\phi_T(v - (\alpha+1)i)}{\alpha^2 + \alpha - v^2 + i(2\alpha+1)v}$$

For η, we need to choose a number that could satisfy $\eta \le \alpha/N$

$$\lambda = \frac{2\pi}{N}$$

$$b = 0.5N\lambda$$

$$K_u = -b + \lambda(u - 1)$$

We first find the Ku which is most close to ln(K), and then noting that $V_j = (j-1)\eta$, we have

$$C_T(K_u) = \exp -\alpha k_u/\pi * \Sigma_{j=1}^{N} \exp -i\lambda\eta(j-1)(u-1) * \exp ibV_j * \Phi(V_j) * \eta$$

With Simpon's Rule, we can write the equation as:

$$C_T(K_u) = \exp -\alpha k_u/\pi * \Sigma_{j=1}^{N} \exp -i\lambda\eta(j-1)(u-1) * \exp ibV_j * \Phi(V_j) * \eta/3[3 + (-1)^j - \omega_{j-1}]$$

```
import numpy as np
from numpy import complex, exp, log, pi,round
al = 1.75
def PHI(mu,r,t,S0,sig):
    def phi(u):
        return exp(complex(-0.5*u**2*sig**2*t,u*(log(S0)+t*(r-sig**2/2)) ))
    al = 1.75
    u = exp(-r*t) * phi(complex(mu, - (al+1)))
    d = complex(al**2+al-mu**2, mu*(2*al+1))
```

```
    return u/d

def FFT_Option(S0,K,r,T,sig, N = 5000):
    k = np.log(K)
    a = 1.75
    eta = a/N + 0.5
    lam = 2*pi/N/eta
    b = N*lam/2
    li_k = []
    def generator(v):
        return exp(np.complex(0,b*v))*PHI(v,r,T,S0,sig)*eta
    for u in range(1, N+1):
        ku = -b+lam*(u-1)
#         print(ku)
        li_k.append(ku)
        A = exp(-a*ku)/pi
        print(ku)
        if round(ku,2) == round(k,2):
            mark = u
            break
    inp = []
    for j in range(1,N+1):
        inp.append(generator(eta*(j-1)))

    out = np.fft.fft(inp)[mark].real*A
    return out

FFT_Option(30,30,0.05,1,0.1, N = 5000)
#FFT Model
vanilla_Euro_option(30, 30, 1, 0.05, 0.1, option = 'call')
#BS Model
vanilla_Euro_option(30, 30, 1, 0.05, 0.1, option = 'call')

Output:
FFT Model  2.030044369967576
BS Model  2.041487312646641
```