Name: JIan Wang **Course: Computational Finance** **Fsuid: Jw09r**

0. Executive Summary

In this problem, I calculated the value of a European put by solving the Black-Scholes equation with a second order finite difference method.
I used K=$10, r=$0.05/yr, σ =0.20/yr, T=0.5/yr with accuracy to nearest penny, the results of the value of put option are as follows with the S belongs to [0,20]

S	V	S	V
S=1	8.75	S=11	0.16
S=2	7.75	S=12	0.05
S=3	6.75	S=13	0.01
S=4	5.75	S=14	0.00
S=5	4.75	S=15	0.00
S=6	3.75	S=16	0.00
S=7	2.76	S=17	0.00
S=8	1.8	S=18	0.00
S=9	0.99	S=19	0.00
S=10	0.44	s=20	0.00

The result is displayed in the nearest cent.
Besides, if we choose S=8:
Then according to the above table
V=$1.80 (accurate to the nearest penny).

$\Delta = = \frac{u_{j+1} - u_{j-1}}{2\Delta x}$ =-0.908

I also check the model with a know solution PDE program, the result shows that the model is feasible.

I. Statement of Problem

Compute the value of the European put by solving the Black-Scholes equation with a second order finite difference method. I used the Crank-Nicolson method to solve this partial differential equation. Besides, I also calculate the sensitivity of the value of the option to the interest rate. Finally, I used a known function to test the accuracy of my model.

II. Description of The Mathematics

Firstly, we already know the initial-boundary-value (IBVP) for the European put is

$$\frac{\partial V^*}{\partial t^*} + \frac{1}{2}\sigma^{*2} S^{*2} \frac{\partial^2 V^*}{\partial S^{*2}} + r^* S^* \frac{\partial V^*}{\partial S^*} = r^* V^*$$

$$V(S^*, T^*) = \max(K^* - S^*, 0)$$

$$V(0, t^*) = K^* e^{-r^*(T^* - t^*)}$$

$$V(S^*, t^*) = 0 \quad as S^* \to \infty$$

We scaled this by letting:

$$x = \frac{S^*}{K^*},\ U = \frac{V^*}{K^*},\ \tau^* = r_0^*(T^* - t^*),\ \sigma = \sigma^* / \sigma_0^*, r = r^* / r_0^*,\ \frac{1}{R} = \frac{1}{2}\frac{(\sigma_0^*)^2}{r^*_0}$$

then we have equations:

$$\frac{\partial V}{\partial S} = \frac{\partial U}{\partial x},\ \frac{\partial^2 V}{\partial S^2} = \frac{1}{K} * \frac{\partial^2 U}{\partial x^2},\ \frac{\partial V}{\partial t} = K\frac{\partial U}{\partial \tau}$$

So the scaled IBVP is:

$$\frac{\partial U}{\partial \tau} - rx\frac{\partial U}{\partial x} = \frac{1}{2}\sigma^2 x^2 \frac{\partial^2 U}{\partial x^2} - rU$$

$$U(x, 0) = \max(1 - x, 0)$$

$$U(0, \tau) = e^{-r\tau}$$

$$U(x, \tau) = 0 \quad as\ x \to \infty$$

Secondly,

we used Crank-Nicolson approximation to find the numerical solution to this PDE problem. The formula is as follows:

$$\frac{\partial^2 U}{\partial x^2} = \delta_x^+ \delta_x^- x + O(\Delta x^2)$$

$$\frac{\partial U}{\partial x} = \delta_x^0 x + O(\Delta x^2)$$

Then

$$\delta_x^+ \delta_x^- u_j = \frac{u_{j+1} - 2u_j + u_{j-1}}{\Delta x^2} + O(\Delta x^2)$$

$$\delta_x^0 u_j = \frac{u_{j+1} - u_{j-1}}{2\Delta x} + O(\Delta x^2)$$

Now we use Trapezoidal Rule to integrate through τ from 0 to T, we get following equations:

$$\delta_\tau^+ u_j^n = \{\frac{\sigma_j^2 x_j^2}{2}\delta_x^+ \delta_x^- + r_j x_j \delta_x^0 - r_j\}\frac{[u_j^{n+1} + u_j^n]}{2}$$

Where

$$\delta_\tau^+ u_j^n = \frac{u_j^{n+1} - u_j^n}{\Delta\tau}$$

So we can represent the whole problem as:

$$M_1 \vec{u}^{n+1} = M_2 \vec{u}^n + \Delta\tau g$$

Where: $\vec{u}^n = \begin{bmatrix} u_1 \\ u_2 \\ . \\ . \\ u_{N_x-1} \end{bmatrix}^n$, $g = \begin{bmatrix} a_1(\frac{u_0^n + u_0^{n+1}}{2}) \\ 0 \\ . \\ 0 \\ c_{N_x-1}(\frac{u_\infty^n + u_\infty^{n+1}}{2}) \end{bmatrix}$

And:

$$M_1 = diag(\frac{-\Delta\tau}{2}a, 1 - \frac{\Delta\tau}{2}b, \frac{-\Delta\tau}{2}c)$$

$$M_2 = diag(\frac{\Delta\tau}{2}a, 1 + \frac{\Delta\tau}{2}b, \frac{\Delta\tau}{2}c)$$

$$a_j = \frac{\sigma_j^2 x_j^2}{2\Delta x^2} - \frac{r_j x_j}{2\Delta x}$$

$$b_j = \frac{-\sigma_j^2 x_j^2}{2\Delta x^2} - r_j$$

$$c_j = \frac{\sigma_j^2 x_j^2}{2\Delta x^2} + \frac{r_j x_j}{2\Delta x}$$

Thirdly, form the following formula, we can get the error.

$$u_{1j}^n = u_j^n + c\Delta x^2$$

$$u_{22j}^{2n} = u_{2j}^{2n} + c\Delta x^2$$

Then the error is as follows:

$$u_{1j}^n - u_{22j}^{2n} = \frac{c\Delta x^2}{E_{1j}^n}(1 - \frac{1}{4})$$

$$E_{1j}^n = \frac{4}{3}(u_{1j}^n - u_{22j}^{2n})$$

Finally,

I used a known solution PDE problem to check the accuracy of the model. If we add a

forcing term $(1-x-\frac{x^2}{2})e^x$ to the right of the initial problem, and let $\sigma = r = 1$. Then the exact solution of the following problem is $e^x + e^{-\Gamma}$

$$\frac{\partial U}{\partial \tau} - rx\frac{\partial U}{\partial x} = \frac{1}{2}\sigma^2 x^2 \frac{\partial^2 U}{\partial x^2} - rU + (1-x-\frac{x^2}{2})e^x$$
$$u(x,0) = e^x + 1$$
$$u(0,\tau) = 1 + e^{-\tau}$$
$$u(2,\tau) = e^2 + e^{-\tau}$$

III. Description of the Algorithm

Get $\vec{u}^0$ form the initial function of S

Input N_x, N_τ

For n=0 **To** N_τ -1

Compute $\vec{R}^n = M_2\vec{u}^n + \Delta\tau g$

Solve $M_1\vec{u}^{n+1} = \vec{R}^n$ for $\vec{u}^{n+1}$

Next n

IV. Results

The Value of European put calculated by C-N procedure for K=\$10, r=\$0.05/yr, σ =0.20/yr, T=0.5/yr with accuracy to nearest penny is as follows:

S	V	S	V
S=1	8.75	S=11	0.16
S=2	7.75	S=12	0.05
S=3	6.75	S=13	0.01
S=4	5.75	S=14	0.00
S=5	4.75	S=15	0.00
S=6	3.75	S=16	0.00
S=7	2.76	S=17	0.00
S=8	1.8	S=18	0.00
S=9	0.99	S=19	0.00
S=10	0.44	s=20	0.00

By running the procedure two times with $\Delta x = \Delta\tau = 0.01$ and $\Delta x = \Delta\tau = 0.005$,

we can get the following output in which the absolute error was calculated by the equation $E_{1j}^{n} = \frac{4}{3}(u_{1j}^{n} - u_{22j}^{2n})$. The results are as follows:

Stock Price	Option value	Abs Error	Rel Error
1	8.7531	5.07974E-09	5.80336E-10
2	7.7531	5.07974E-09	6.55188E-10
3	6.7531	5.07972E-09	7.52206E-10
4	5.7531	5.03924E-09	8.75917E-10
5	4.7531	-6.68012E-08	-1.40542E-08
6	3.75319	-5.21828E-06	-1.39036E-06
7	2.75687	3.74006E-05	-1.35663E-05
8	1.79871	8.50182E-06	4.72663E-06
9	0.987804	0.000237598	0.000240531
10	0.441628	0.000344371	0.000779777
11	0.160418	0.000219242	0.00136669
12	0.0482722	7.21586E-05	0.00149483
13	0.0123743	-6.74375E-06	5.44981E-05
14	0.00278088	-6.03358E-06	-0.00216966
15	0.00056237	-4.1625E-06	-0.0074017
16	0.00010467	-1.67051E-06	-0.0159594
17	1.8275E-05	-5.23964E-07	-0.0286705
18	3.0397E-06	-1.40641E-07	-0.0462674
19	4.7713E-07	-3.24423E-08	-0.0679954

2.

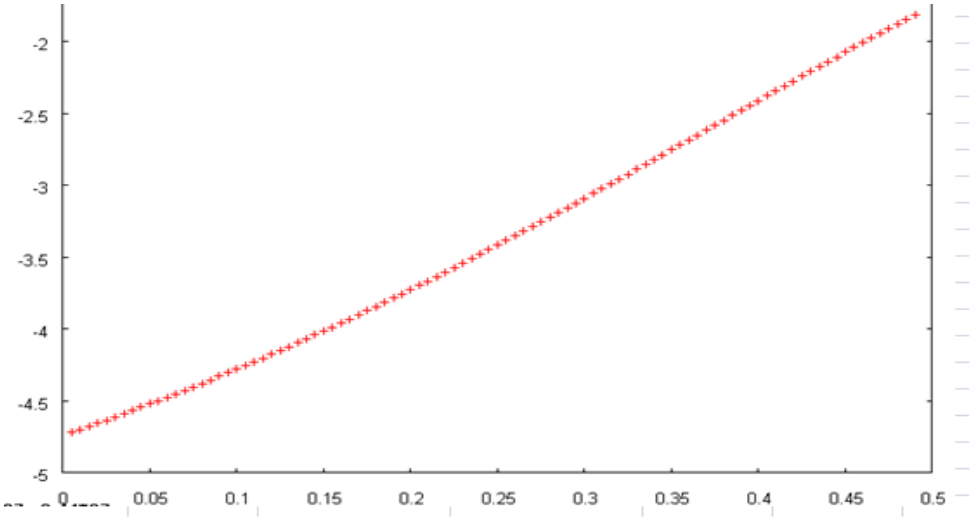
Since $x = \frac{S}{K}$, $U = \frac{V}{K}$, then $\Delta = \frac{\partial V}{\partial S} = \frac{\partial U}{\partial x}$ .So choose S=8, V=$1.80.

$\Delta = \frac{\partial V}{\partial S} = \frac{\partial U}{\partial x} = \frac{u_{j+1} - u_{j-1}}{2\Delta x}$ =-0.908

3.

To test the sensitivity of the value of the option to the interest rate, I calculate the value of the greek $\rho = \frac{\partial V}{\partial r}$

I choose the stock price as 8 and the following graph of is the ρ value where R change from 0 to 0.5. The following chart showed the result of the sensitivity.



V. Conclusions

I choose a known PDE problems to test the accuracy of the model.

The PDE formula is as follows:

$$\frac{\partial U}{\partial \tau} - rx\frac{\partial U}{\partial x} = \frac{1}{2}\sigma^2 x^2 \frac{\partial^2 U}{\partial x^2} - rU + (1 - x - \frac{x^2}{2})e^x$$

$$u(x,0) = e^x + 1$$

$$u(0,\tau) = 1 + e^{-\tau}$$

$$u(2,\tau) = e^2 + e^{-\tau}$$

And choose x from 0 to 2, τ from 0 to 1, $\sigma = r = 1$, then the exact solution will be $e^x + e^{-\tau}$. The theory tells us:

1. The error = $O(\Delta x^2, \Delta t^2)$.

   In this case, since $X_{\max} = 2 * T_{\max}$. If we choose $N_x = 2N_t$, then $\Delta t = \Delta x$ and the error calculated should be reduced at the level of 10^{-2}, when $\Delta t = \Delta x$ reduced at the level of 10^{-1}. The results are as follows:

Calculated value	Absolute Error	Delta t
2.59577	2.35E-03	0.1
2.59344	2.35E-05	0.01
2.59342	2.35E-07	0.001

The absolute error is the numerical solution calculated by C-N procedure minus the exact solution $e^{x}+e^{-\tau}$ at the point x=0.8 and τ =1, which equals to 2.59342037. Form the above table, we can see that the theory that when $\Delta t=\Delta x$ reduced to 1/10, the error reduced 1/100.

If we do the Crank-Nicholson procedure twice and choose $\Delta t_1 = 2 * \Delta t_2$. The error should satisfy the following equation $E_{1j}^{\ n}=\frac{4}{3}(u_{1j}^{\ n}-u_{22j}^{\ 2n})$, I have run the procedure twice by choosing $\Delta t_1=\Delta x_1$=0.005 and $\Delta t_2=\Delta x_2$=0.0025. The real error is calculated by calculated solution minus exact solution $e^{x}+e^{-\tau}$ at x, and the estimated error is calculated by the equation $E_{1j}^{\ n}=\frac{4}{3}(u_{1j}^{\ n}-u_{22j}^{\ 2n})$, and these two columns match.

The following is the results:

The value of x	Real error	Estimated Error
0.1	-1.41531E-07	-1.41531E-07
0.2	7.8563E-07	7.8563E-07
0.3	1.80654E-06	1.80654E-06
0.4	2.80978E-06	2.80978E-06
0.5	3.74098E-06	3.74098E-06
0.6	4.5719E-06	4.5719E-06
0.7	5.28646E-06	5.28646E-06
0.8	5.8743E-06	5.8743E-06
0.9	6.32755E-06	6.32755E-06
1	6.63925E-06	6.63925E-06
1.1	6.80244E-06	6.80244E-06
1.2	6.80962E-06	6.80962E-06
1.3	6.65251E-06	6.65251E-06
1.4	6.32174E-06	6.32174E-06
1.5	5.80679E-06	5.80679E-06
1.6	5.09579E-06	5.09579E-06
1.7	4.17541E-06	4.17541E-06
1.8	3.03072E-06	3.03072E-06
1.9	1.64509E-06	1.64509E-06

We can see that the real error and the estimated error are just same, which shows that the formula and the results are right.

VI. Code Listing

<main.cpp>

```
#include "Quadrature.h"
#include <iostream>
#define _USE_MATH_DEFINES
#include <math.h>
#include <fstream>
using namespace std;

void TriDiagonal( double *A, double *B, double *C, double *R, double *U, int number)
{
for(int k=2;k<number;k++)
{
B[k]=B[k]-C[k-1]/B[k-1]*A[k];
R[k]=R[k]-R[k-1]/B[k-1]*A[k];
}
U[number-1]=R[number-1]/B[number-1];
for(int k=number-2;k>=1;k--)
{
U[k]=(R[k]-C[k]*U[k+1])/B[k];
}

}

void Payoff(double *f1,double dx, int number)
{
for(int k=0;k<number;k++)
{
if((1-k*dx)>0) f1[k]=1-k*dx;
else f1[k]=0;
}
}

double ForcingTerm(double x)
{
return 0;
}

void solve(int x, int t,double r,double *f1)
{
double Smax=20.0;
```

```
double Tmax=0.5;
double StrikePrice=10.0;
double RatioRate=r;
double Volatility=0.2;

int Nx=x;
int Nt=t;

double Xmax=Smax/StrikePrice;
double dx=Xmax/Nx;
double dt=Tmax/Nt;

double* U=new double[Nx];
double* A=new double[Nx];
double* B=new double[Nx];
double* C=new double[Nx];
double* R=new double[Nx];
double* F=new double[Nx];
double Xj=0;
double Tj=0;

Payoff(U, dx, Nx);

for(int p=0;p<Nt;p++)
{
Tj=p*dt;
for(int k=1;k<Nx;k++)
{
Xj=k*dx;
A[k]=(pow(Volatility,2.0)*pow(Xj,2.0)/2.0/pow(dx,2.0)-RatioRate*Xj/2.0/dx)*dt/2;
B[k]=1+((-1)*pow(Volatility,2.0)*pow(Xj,2.0)/pow(dx,2.0)-RatioRate)*dt/2;
C[k]=(pow(Volatility,2.0)*pow(Xj,2.0)/2.0/pow(dx,2.0)+RatioRate*Xj/2.0/dx)*dt/2;
F[k]=ForcingTerm(Xj);
}

for(int k=2;k<Nx-1;k++)
{
R[k]=U[k-1]*A[k]+U[k]*B[k]+U[k+1]*C[k];
}

R[1]=2*A[1]*(exp((-1)*RatioRate*p*dt)+exp((-1)*RatioRate*(p+1)*dt))/2+U[1]*B[1]+U[2]*C[
1];
R[Nx-1]=U[Nx-2]*A[Nx-1]+U[Nx-1]*B[Nx-1];
```

```
for(int k=1;k<Nx;k++)
{
R[k]=R[k]+dt*F[k];
}

for(int k=1;k<Nx;k++)
{
A[k]=(-1)*A[k];
B[k]=(-1)*B[k]+2;
C[k]=(-1)*C[k];
}

TriDiagonal(A, B, C, R, U, Nx);
}


for(int k=1;k<Nx;k++)
{
f1[k]=U[k];
}


delete [] A;
delete [] B;
delete [] C;
delete [] R;
delete [] U;
}



void main()
{
ofstream out;
out.open("out.txt");

}
```