Lab12_sol

November 21, 2017

1 Sample solutions of exercises of Lab 12

Suppose we want to find the current prices of an European option so that the error at the current stock price $S(0) = S_0$ was less that ε. When using a finite difference method for the untransformed equation, we have to fix only one artificial boundary; a good starting point is to take $s_{max} = 2 \cdot S_0$ (if σ is large or the time period is long, it may make sense to take larger value for s_{max}). In the case of solving the transformed equation, we also have to introduce another boundary for which we may take $s_{min} = \frac{S_0}{2}$ and define $x_{min} = \ln s_{min}$, $x_{max} = \ln s_{max}$. Our procedure is as follows:

1. Fix a starting value n_0 and choose m_0 (in the case of the explicit method choose it from the stability constraint) and define $z = 1$.
2. Solve the problem with a finite difference method (starting with $n = z \cdot n_0$ and $m = m_0$) and estimate the error by Runge's method, until the (estimated) finite difference discretization error is less than $\frac{\varepsilon}{2}$.
3. multiply s_{max} by 2, divide s_{min} by 2, increase z by one in the case of the transformed equation or multiply it by two in the case of the untransformed equation and solve the problem with the same method (starting with $n = z \cdot n_0$, $m = m_0$ again until the (estimated) finite difference discretization error is less than $\frac{\varepsilon}{2}$. If the answer changes by more than $\frac{\varepsilon}{2}$ then repeat the step. Otherwise we assume that we have obtained the solution with the desired accuracy.

1.1 Exercise 1

Use the procedure above to compute the price of the call option with accuracy 0.02 by using the explicit finite difference method for the transformed problem and simple boundary conditions in the case $S_0 = 51$, $E = 50$, $r = 0.03$, $D = 0$, $T = 1$, $\sigma = 0.5$. Find the actual error of the final answer.

Solution Of cause the solution procedure can (and finally should) be fully automated as a function that does all needed compuations and returns the final answer. But at leas first time it is reasonable to follow the procedure step by step. So here only step by step solution is presented.

We are going to use the function explicit_solver from lab9 (see the sample solutions of Lab9)

```
In [1]: import numpy as np
        from scipy import linalg
        import sys
        sys.path.append("h:/compfin_labs")
        import numpy as np
        def explicit_solver(n,rho,r,D,S0,T,sigma,p,phi1,phi2):
            """sigma is assumed to be a constant
            phi1,phi2 are functions of xmin,t and xmax,t
```

```
    p is a function of stock price only
    """
    xmax=np.log(S0*rho)
    xmin=np.log(S0/rho)
    delta_x=(xmax-xmin)/n
    #find m from the stability condition
    m=T*(sigma**2/delta_x**2+r)
    m=np.int64(np.ceil(m)) #has to be an integer
    delta_t=T/m
    #define values of x_i
    x=np.linspace(xmin,xmax,n+1)
    #define matrix U with dimension (n+1)x(m+1)
    U=np.zeros(shape=(n+1,m+1))
    #fill in the final condition
    U[:,m]=p(np.exp(x))
    #define a,b,c
    alpha=sigma**2/2
    beta=r-D-alpha
    a=delta_t/delta_x**2*(alpha-beta*delta_x/2)
    b=1-2*delta_t/delta_x**2*alpha-r*delta_t
    c=delta_t/delta_x**2*(alpha+beta*delta_x/2)
    #compute all other values
    i=np.arange(1,n)
    t=np.linspace(0,T,m+1)
    for k in range(m,0,-1): #backward iteration, k=m,m-1,...
        #boundary conditions
        U[0,k-1]=phi1(xmin,t[k-1])
        U[n,k-1]=phi2(xmax,t[k-1])
        #all other values
        U[i,k-1]=a*U[i-1,k]+b*U[i,k]+c*U[i+1,k]
    return [U[:,0],np.exp(x)]

## Exercise 1
S0 = 51; E = 50; r = 0.03; D = 0; T = 1; sigma = 0.5
def p_call(s):
    return np.maximum(s-E,0)
def phi1_const(xmin,t):
    return p_call(np.exp(xmin))
def phi2_const(xmax,t):
    return p_call(np.exp(xmax))
```

Let us start the procedure. For this set the starting parameters. For n_0 usually a relatively small value (like 10) is chosen

```
In [2]: n0=10
        z=1
        rho=2
        total_error=0.02
```

Computations for the first value of rho:

```
In [3]: n=z*n0
        answer1=explicit_solver(n,rho,r,D,S0,T,sigma,p_call,phi1_const,phi2_const)[0][n//2]
        n=n*2
        answer2=explicit_solver(n,rho,r,D,S0,T,sigma,p_call,phi1_const,phi2_const)[0][n//2]
        error_estimate=np.abs(answer2-answer1)/3
        print(answer1,answer2,error_estimate<total_error/2)
```

```
11.0511493963 11.0289819061 True
```

As the estimate of the discretization error is smaller than half of the total error, we have found the final answer for this *rho*. If it the estimate were larger than half of the total error, we would continue computing by setting answer1=answer2, multiplying n by 2, computing a new value for answer2 and estimating the discretization error again and so on.

Now we save the final answer for this *rho* in a variable and compute the final answer for the next value of rho

```
In [4]: answer_rho1=answer2
        ##computations for the second rho
        rho=rho*2
        z=z+1 #transformed equation
        n=z*n0
        answer1=explicit_solver(n,rho,r,D,S0,T,sigma,p_call,phi1_const,phi2_const)[0][n//2]
        n=n*2
        answer2=explicit_solver(n,rho,r,D,S0,T,sigma,p_call,phi1_const,phi2_const)[0][n//2]
        error_estimate=np.abs(answer2-answer1)/3
        print(answer1,answer2,error_estimate<total_error/2)
```

```
11.1366781126 11.1315760525 True
```

As the the estimate of the discretization error is small enough, we have found the answer for the second value of ρ. So we save this answer in a variable and check if the truncation error is small enough

```
In [5]: answer_rho2=answer2
        print(np.abs(answer_rho1-answer_rho2)<total_error/2)
```

```
False
```

So changing the value of ρ had significant effect on the answer we got and therefore the truncation error is not small enough. We have to repeat the computations with larger value of ρ. For a new computation the last result answer_rho2 becomes the less acurate result answer_rho2, we increase the value of rho and z and compute a new answer_rho2

```
In [6]: rho=rho*2
        z=z+1
        answer_rho1=answer_rho2
        n=z*n0
        answer1=explicit_solver(n,rho,r,D,S0,T,sigma,p_call,phi1_const,phi2_const)[0][n//2]
        n=n*2
        answer2=explicit_solver(n,rho,r,D,S0,T,sigma,p_call,phi1_const,phi2_const)[0][n//2]
        error_estimate=np.abs(answer2-answer1)/3
        print(error_estimate<total_error/2)
        #True - finished for this rho
        answer_rho2=answer2
        print(np.abs(answer_rho1-answer_rho2)<total_error/2)
        #True - we are finished
        print("final answer:",answer_rho2)
        #check with true answer
        from BSformulas import Call
        exact_price=Call(S0,E,T,r,sigma,D)
        print(exact_price)
```

```
True
True
final answer: 11.1322483109
11.1100117809
```

So we see, that the final answer is computed with error that is very close to the allowed error 0.02 but slightly larger. So we have to understand that such practical procedures do not guarantee completele that we obtain the answer with desired accuracy, but usually they work reasonably well. There are several reasons why the error estimates we are using may be slightly wrong - the estimates are of limiting nature (for large enough value of n, the error is reduced approximately 4 times when n is multiplied by 2) and the estimates are valid when the solution of the PDE is smooth enough (has 4 derivatives with respect to it's variables). Actually, the later assumption is not valid for the pricing function of the Call option - since the derivative of the payoff function is not continuos, the pricing function is not 4 times differentiable at $t = T$, so that in Runge's estiamte it is safe not to divide by 3, but only with 1 if it is very important to compute the answer with given accuracy.

1.2 Exercise 2

Find the price of the European option with payoff

$$p(s) = \begin{cases} 40 - \frac{s}{2}, & s \leq 80, \\ 0, & 80 < s \leq 110, \\ \frac{3s}{2} - 165, & s > 110 \end{cases}$$

with maximal error 0.01 for current stock price $S0 = 105$ in the case, $r = 0.05$, $D = 0$, $T = 0.5$ and nonconstant volatility

$$\sigma(s,t) = 0.4 + \frac{0.3}{1 + 0.01(s-90)^2}.$$

Use Crank-Nicolson method for the untransformed equation, the exact boundary condition

$$\phi_1(t) = p(0)e^{-r(T-t)}$$

at the boundary $x_{min} = 0$ and the boundary condition corresponding to a special (linear in s) solution at the boundary $x = s_{max}$ to obtain the answer.

Solution

Define a solver for untransformed BS equation using Crank-Nicolson method

```
In [7]: def CN_untransformed(m,n,xmax,r,D,T,sigma,p,phi2):
            """sigma is assumed to be a function of s and t
            phi2 is a functions of xmax and t
            solver for untransformed problem, x is equal to s
            return prices for t=0
            """
            xmin=0
            delta_x=(xmax-xmin)/n
            x=np.linspace(xmin,xmax,n+1)
            #define the function alpha
            #for transformed BS equation
            def alpha(x,t):
                return sigma(x,t)**2*x**2/2
            delta_t=T/m
            #define matrix U with dimension (n+1)x(m+1)
            U=np.zeros(shape=(n+1,m+1))
            #fill in the final condition
            U[:,m]=p(x)
            #compute all other values
            i=np.arange(1,n)
            t=np.linspace(0,T,m+1)
            #define matrix M
            M=np.zeros(shape=(n+1,n+1))
            M[0,0]=1
            M[n,n]=1
            #define vector F
            F=np.zeros(n+1)
            for k in range(m-1,-1,-1):
                #compute the coefficients
                alpha_vec=alpha(x[i],t[k]+delta_t/2)
                beta=(r-D)*x[i]
                a=1/2*delta_t/delta_x**2*(-alpha_vec+beta*delta_x/2)
                b=1+delta_t/delta_x**2*alpha_vec+1/2*r*delta_t
                c=-1/2*delta_t/delta_x**2*(alpha_vec+beta*delta_x/2)
                d=-a
                e=1-delta_t/delta_x**2*alpha_vec-1/2*r*delta_t
                f=-c
                #Fill M with right values
                M[i,i-1]=a
                M[i,i]=b
```

```
            M[i,i+1]=c
            #Fill F
            F[0]=p(0)*np.exp(-r*(T-t[k]))#exact formula
            F[n]=phi2(xmax,t[k])
            F[i]=d*U[i-1,k+1]+e*U[i,k+1]+f*U[i+1,k+1]
            #solve the system
            U[:,k]=linalg.solve(M,F)
        return U[:,0] #option prices for t=0
```

Define the data for the exercise

```
In [8]: def p(s):
            return (40-s/2)*(s<=80)+(3*s/2-165)*(s>110)
        def sigma(s,t):
            return 0.4+0.3/(1+0.01*(s-90)**2)
        def phi2_spec(xmax,t):
            return 3/2*np.exp(-D*(T-t))*xmax-165*np.exp(-r*(T-t))
        r=0.05;D=0;T=0.5;S0=105;total_error=0.01;n0=10;m0=5;z=1;rho=2
```

Computations for the first ρ

```
In [9]: n=z*n0
        m=m0
        answer1=CN_untransformed(m,n,rho*S0,r,D,T,sigma,p,phi2_spec)[n//rho]
        n=n*2
        m=m*2
        answer2=CN_untransformed(m,n,rho*S0,r,D,T,sigma,p,phi2_spec)[n//rho]
        error_estimate=np.abs(answer2-answer1)/3
        print(answer1,answer2,error_estimate<total_error/2)
```

```
21.8698941708 22.4290083919 False
```

```
In [10]: answer1=answer2
         n=n*2
         m=m*2
         answer2=CN_untransformed(m,n,rho*S0,r,D,T,sigma,p,phi2_spec)[n//rho]
         error_estimate=np.abs(answer2-answer1)/3
         print(answer1,answer2,error_estimate<total_error/2)
```

```
22.4290083919 22.361372997 False
```

```
In [11]: answer1=answer2
         n=n*2
         m=m*2
         answer2=CN_untransformed(m,n,rho*S0,r,D,T,sigma,p,phi2_spec)[n//rho]
         error_estimate=np.abs(answer2-answer1)/3
         print(answer1,answer2,error_estimate<total_error/2)
```

```
22.361372997 22.3986305716 False
```

```
In [12]: answer1=answer2
         n=n*2
         m=m*2
         answer2=CN_untransformed(m,n,rho*S0,r,D,T,sigma,p,phi2_spec)[n//rho]
         error_estimate=np.abs(answer2-answer1)/3
         print(answer1,answer2,error_estimate<total_error/2)
```

```
22.3986305716 22.4045541645 True
```

Now we have finished with the first value of rho. We have to do the same for the next value of rho. Of cause it is possible to avoid copyng the same lines over and over again by using a while cycle

```
In [13]: answer_rho1=answer2
         rho*=2
         z*=2 #untransformed equation
         n=z*n0
         m=m0
         answer1=CN_untransformed(m,n,rho*S0,r,D,T,sigma,p,phi2_spec)[n//rho]
         error_estimate=total_error#to force the cycle to start
         while(error_estimate>total_error/2):
             n=n*2
             m=m*2
             answer2=CN_untransformed(m,n,rho*S0,r,D,T,sigma,p,phi2_spec)[n//rho]
             error_estimate=np.abs(answer2-answer1)/3
             answer1=answer2
         answer_rho2=answer2
         truncation_error=np.abs(answer_rho2-answer_rho1)
         print(answer_rho1,answer_rho2,truncation_error<total_error/2)
```

```
22.4045541645 22.4045826212 True
```

Changing ρ did not change the answer significantly, so the final answer is 22.4045826212