

# Pricing European & American Options using Crank- Nicolson finite difference method.

Biswaroop Chatterjee

Department of Computer Science, Cornell University, Ithaca NY 14850.

## ***European Calls and Puts:***

For this part I implemented the Crank Nicolson Finite Differences method in the function BLSPRICE2 (see MATLAB code in Appendix). At first, I calculated the axis of the stock prices divided by the number of steps and the Time in the number of time steps (the number of steps are provided as input). These were put into two vectors  $T_m$  and  $S_n$ . Then I started converting the inputs into the Forward equation/Heat Equation domain. I did this with the following steps:

1. Get the value of  $N_{-\infty}$  and  $N_{+\infty}$  from by transforming  $S_{\min}$  and  $S_{\max}$  in terms of  $x$
2. Convert  $t$  from the time vector mentioned earlier in terms of  $\tau$ .

Subsequently I calculated  $dx$  and  $d\tau$  using the formulas given, and then from these, I calculated alpha ( $\alpha$ ).

These substitutions will enable us to convert the vectors in the original domain  $S_n$  and  $T_m$  into the new domain for the Heat equation which correspond to  $x_n$  and  $\tau_m$ . To convert  $S_n$  to  $x_n$ , I use the  $N_{-\infty}$  and  $N_{+\infty}$  as the boundaries and increment by a step corresponding to  $dx$ . Similarly  $\tau_m$  is converted.

Now we have to calculate the boundaries (marked in light in our new domain which will have the known values.

We convert the boundary conditions for Puts and Calls in the following manner in one step to the Heat equation:

## ***European Calls:***

$$\lim_{x \rightarrow -\infty} C_u(\tau, x) = 0$$

$$\lim_{x \rightarrow \infty} C_u(\tau, x) = e^{0.5(k-1)x^2 + 0.25(k+1)^2\tau} (e^x - e^{-k\tau})$$

$$C_u(0, x) = \max(e^{0.5(k+1)x} - e^{0.5(k-1)x}, 0)$$

## ***European Puts:***

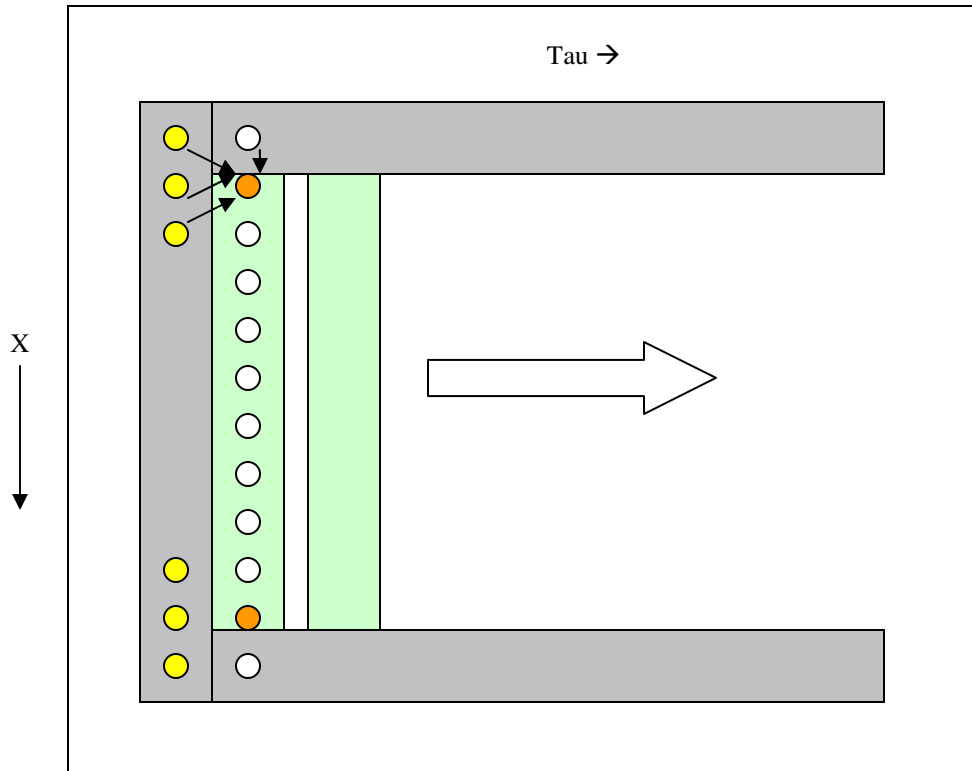
$$\lim_{x \rightarrow -\infty} P_u(\tau, x) = e^{0.5(k-1)x^2 + 0.25(k+1)^2\tau} (e^x - e^{-k\tau})$$

$$\lim_{x \rightarrow \infty} P_u(\tau, x) = 0$$

$$P_u(0, x) = \max(e^{0.5(k-1)x} - e^{0.5(k+1)x}, 0)$$

The interesting point to note is that the direction of time gets reverse as in the original domain the boundary conditions are for  $t = T$  i.e. at expiration and this corresponds to  $\tau = 0$  in the transformed domain.

Here we have a schematic of the transformed domain and how the calculations progress through in our algorithm. So calculating the values at the boundary conditions using our transformed vectors  $\tau_{\text{aum}}$  and  $X_n$ , we get the grey rectangles marked in the figure below which correspond to the known values in the domain.



Then for each timestep ( $\tau$ ) we get a vector from index 2 to index  $(N_x - 1)$  which consist of the unknowns at that timestep.

Here we apply the Crank-Nicolson method shown in the equations below:

$$-\frac{1}{2}\alpha f_{n-1}^{m+1} + (1 + \alpha)f_n^{m+1} - \frac{1}{2}\alpha f_{n+1}^{m+1} = \underbrace{\frac{1}{2}\alpha f_{n-1}^m + (1 - \alpha)f_n^m + \frac{1}{2}\alpha f_{n+1}^m}_{Z_n^m}$$

$$\underbrace{\begin{bmatrix} 1 + \alpha & -\frac{1}{2}\alpha & 0 & \dots & 0 & 0 \\ -\frac{1}{2}\alpha & 1 + \alpha & -\frac{1}{2}\alpha & \dots & 0 & 0 \\ 0 & -\frac{1}{2}\alpha & 1 + \alpha & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 1 + \alpha & -\frac{1}{2}\alpha \\ 0 & 0 & 0 & \dots & -\frac{1}{2}\alpha & 1 + \alpha \end{bmatrix}}_{M_{CN}} F^{m+1} = \underbrace{\begin{bmatrix} Z_1^m \\ Z_2^m \\ Z_3^m \\ \dots \\ Z_{N_x-2}^m \\ Z_{N_x-1}^m \end{bmatrix}}_{b_{CN}^m} + \frac{1}{2}\alpha \begin{bmatrix} f_0^m \\ 0 \\ 0 \\ \dots \\ 0 \\ f_{N_x}^m \end{bmatrix}$$

We set up  $M_{CN}$  by dividing it into its constituent LU matrices. In the U matrix we have the values for (1,1) which is  $1 + \alpha$ . We calculate the rest from this sequentially to get the whole diagonal. The upper diagonal of the U matrix is a constant  $(-\alpha/2)$ .

In the L matrix we have a main diagonal of just 1's starting from index (1, 1). The Lower diagonal is related to the main diagonal of the U matrix.

Once we have set this up, for each tau increment, we calculate the  $b_{CN}^M$  matrix of the RHS in the above equation. Through our boundary conditions we have the values for  $Z_1^m$  and  $Z_{N_x-1}^m$  (3 terms each corresponding to the yellow circles in the diagram) and  $f_0^m$  &  $f_{N_x}^m$ . For the middle values we have the Z's from the lower boundary which we add together in triples and store in our  $b_{CN}^M$  matrix.

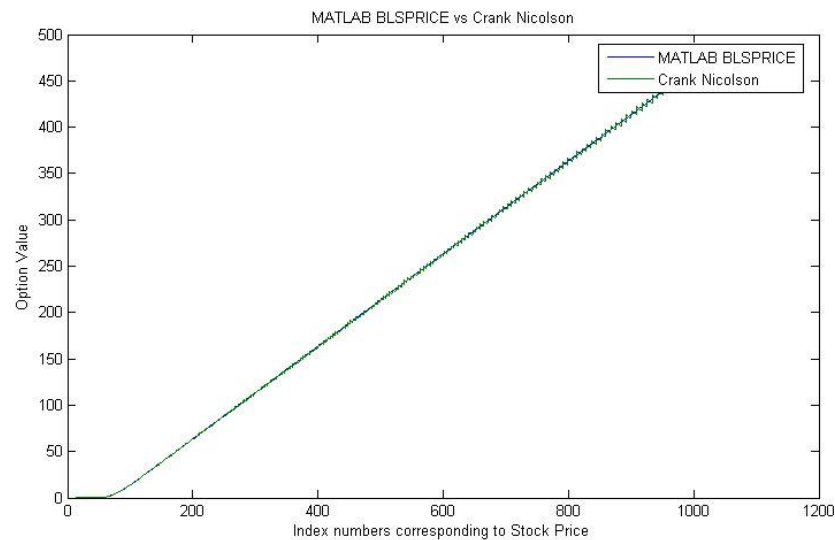
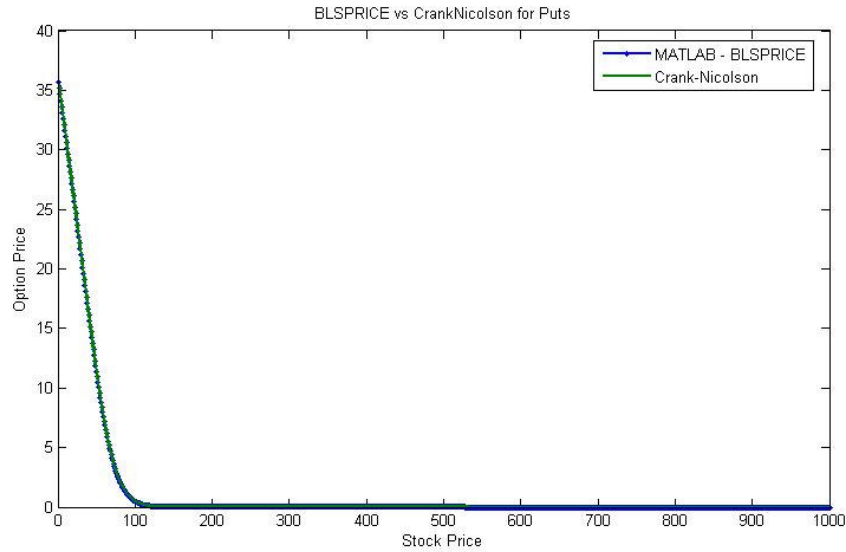
Now we are in a position to solve by first forward substitution in one pass. We know the value of  $v^M(1)$  which is the same as  $b^M(1)$ . For the others we use the lower diagonal L, the value of  $b^M$  at that index and the  $v^M$  from the previous index. When we solve for F, we are doing the backward substitution. Here we start from the last index and then work back to the first.

$$\begin{aligned}
 L_M v^m &= b^m \\
 U_M F^{m+1} &= v^m
 \end{aligned}$$

This enables us to get all the values in our transformed domain, but we still have to convert them back into our original domain now. We do this by locating each stock by converting it into the transformed domain and performing nearest neighbour, linear interpolation. And then we calculate the time to expiration for each taustep and reverse substitute to get back our final Option prices.

The validity of my Matlab function can be seen by comparison with the reference Matlab BLSPRICE function. The parameters which were passed as input were:

**$\sigma = 25$ ;  $\leftarrow 25\%$**   
 **$T = 1$ ;  $\leftarrow 1$  year**  
 **$K = 40$ ;  $r = 0.1$ ;  $\leftarrow 10\%$**   
 **$S_{max} = 500$ ;  $S_{min} = 0.05$ ;**  
 **$N_x = 1000$ ;  $N_t = 2000$ ;**



So here we can see that for both European puts and calls, the Crank-Nicolson method converges to the corresponding option prices given by Matlab's BLSPRICE function. The convergence is better when we have a large number of time steps and space steps. If we have very few steps then the option prices given by Crank Nicolson oscillates around the BLSPRICE

### ***American Put using LCF with Crank Nicholson method***

In this part, I am trying to simulate the American Put using the Linear Complementarity Formulation (LCF) with the Crank Nicholson method using the Projected Successive Over-Relaxation (PSOR) technique.

We start off by again forming vectors for the Stock price  $S_n$  and Time  $T_m$  based on the number of steps. We also calculate the values of  $k$ ,  $\tau$ ,  $d\tau$ ,  $dx$ ,  $\alpha$  using the conversion into the Heat equation domain.

$$\begin{cases} S = Ke^x \\ t = T - \frac{1}{\frac{1}{2}\sigma^2}\tau \\ u(\tau, x) = e^{\frac{1}{2}(k-1)x + \frac{1}{4}(k+1)^2\tau} \frac{P(t, S)}{K} \end{cases}$$

Then we form the new boundaries using  $N_{plus}$  and  $N_{minus}$  and use it to get our transformed domain  $x_n$  and our transformed time domain  $\tau_m$ . Using these two vectors we create our boundary conditions and initial condition which will be our known values ( $u_n^0 = g_n^0$ ,  $0 \leq n \leq N_x$ ).

Then we create our  $M_{CN}$  vector which is simpler in this case because we can directly put in the values and do not have to conduct the LU decomposition.

Now calculate for each  $\tau$  step. Here we first set up  $b_{CN}^M$  in the same manner as previously encountered. The boundary values at each  $\tau$ -step will not change  $u_0^{M+1} = g_0^{M+1}$  and  $u_{N_x}^{M+1} = 0$ . Here we first make an initial guess and call the PSOR solver which will make subsequent guesses based on the following scheme:

$$\begin{cases} y_n^{m+1, k+1} = \frac{1}{1+\alpha} \left[ Z_n^m + \frac{1}{2}\alpha \left( u_{n-1}^{m+1, k+1} + u_{n+1}^{m+1, k} \right) \right] \\ u_n^{m+1, k+1} = u_n^{m+1, k} + \omega (y_n^{m+1, k+1} - u_n^{m+1, k}) \end{cases},$$

This keeps taking place until the tolerance is reached. In this function the  $\omega$  values starts off from the midpoint between 1 and 2 and is changed based on the performance of the PSOR function.

**Selection of  $\omega$ :** The  $\omega$  value (relaxation parameter) depends on the size of the matrices involved. Here in the algorithm, I start off with a value of 1.5 as suggested in the assignment. To find the optimal  $\omega$ , what I do is have a  $d\_omega$  parameter which will change the value of  $\omega$  if the number of loops required by the PSOR function increases. In the example which I had, the  $\omega$  value finally settled at around 1.1. This is quite a good way of going about the solution because the relaxation parameter can change if the solver is taking too long to converge. This method was mentioned in the Wilmott book.

Once we have found the solution for the transformed problem we again bring back our answer which is in terms of  $x$  and  $\tau$  in the heat equation domain back into the financial domain.

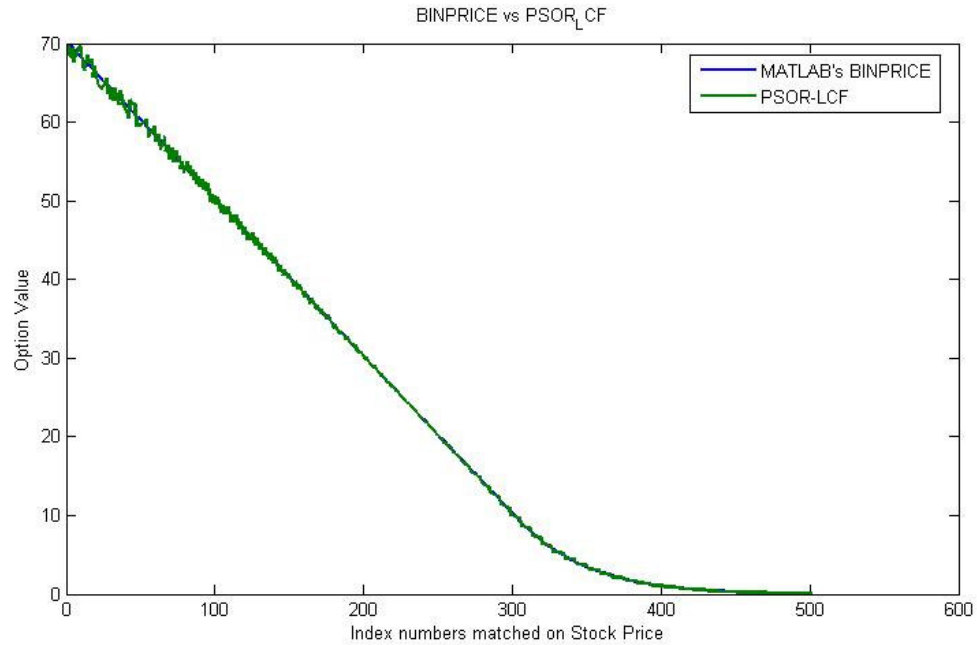
The validity of the PutAM function (see MATLAB code in the Appendix) can be seen by comparing with MATLAB's BINPRICE function which was run for small timesteps to provide good accuracy.

The option parameters I used were:

$$\begin{aligned} S_{max} &= 100; \\ S_0 &= 50; \end{aligned}$$

$K = 70;$   
 $r = 0.10;$   
 $\sigma = 0.20;$   
 $T = 1;$   
 $\epsilon = 1E-8;$   
 $S_{min} = 0.001;$   
 $N_t = 500;$   
 $N_x = 500;$

for binprice I used timestep = 0.01



Here we can see that the function follows very closely. There is an initial disturbance at first when the PSOR function oscillates. The PSOR-LCF method is ofcourse slower than the earlier method but it is superior because it can handle American options as well as European options.

## Appendix A:

### Eu Calls and Puts:

```
function [Onm, Tm, Sn, taum, xn] = blsprice2(type, K, r, sigma, T, Nt, Smin, Smax, Nx)
```

```
Sn = [Smin:(Smax-Smin)/Nx:Smax]; % axis of the Stock
```

```
Tm = [0:T/Nt:T]; % evolution of the time
```

```
% rewriting conditions to correspond to the new variables
```

```
k = r / (0.5 * sigma^2); % getting the value of k
```

```
Nminusinf = log(Smin / K);
```

```
Nplusinf = log(Smax / K);
```

```
% obtaining the transformed bounds of the log steps of the Stock price evolution
```

```
dtau = 0.5 * sigma^2 * T / Nt;
```

```
dx = (Nplusinf - Nminusinf)/Nx;
```

```
alpha = dtau/((dx)^2);
```

```
% these substitutions enable us to obtain xn and taum in the transformed
```

```
% dimension
```

```
xn = [Nplusinf-dx:Nminusinf]; % vectors for x
```

```
taum = [0:dtau:dtau*Nt]; % vectors for tau
```

```
% now we calculate the boundaries
```

```
% calculating the Nplusinf boundary:
```

```
if (type == 0)
```

```
    % here x= Nplusinf for calls
```

```
    V(1,2:Nt+1) = (exp(0.5*(k-1).* Nplusinf.*ones(Nt,1) +  
0.25.*(k+1)^2.*taum(2:Nt+1)).*(exp(Nplusinf.*ones(Nt,1)) - exp(-1.*k.*taum(2:Nt+1)))));
```

```
    %V(1,2:Nt+1) = (exp(0.5*(k+1).* Nplusinf.*ones(Nt,1) +  
0.25.*(k+1)^2.*taum(2:Nt+1)))'; % for calls
```

```
else
```

```
    V(1,2:Nt+1) = zeros(1,Nt); % for puts
```

```
end
```

```
% calculating the Nminusinf boundary:
```

```
if (type == 0)
```

```
    % for calls the boundaries will be all zero
```

```
    V(Nx+1,2:Nt+1) = zeros(1,Nt);
```

```
else
```

```
    % here x= Nminusinf for puts, therefore the boundaries will be
```

```
    V(Nx+1,2:Nt+1) = (exp(0.5*(k-1).*Nminusinf.*ones(Nt,1) + 0.25.*(k-  
1)^2.*taum(2:Nt+1)).*(exp(Nplusinf.*ones(Nt,1)) - exp(-1.*k.*taum(2:Nt+1)))));
```

```
    %V(Nx+1,2:Nt+1) = (exp(0.5*(k-1).*Nminusinf.*ones(Nt,1) + 0.25.*(k-  
1)^2.*taum(2:Nt+1)))';
```

```
end
```

```
% calculating the lower boundary (with the stock prices)
```

```

if (type == 0)
    V(1:Nx+1,1) = (max(exp(0.5.*(k+1).*xn) - exp(0.5.*(k-1).*xn),0))';
else
    V(1:Nx+1,1) = (max(exp(0.5.*(k-1).*xn) - exp(0.5.*(k+1).*xn),0))';
end

% create the LU matrix
% create the central diagonals for L and U
L_diag = ones(Nx-1);
U_diag(1) = 1 + alpha;
for i = 2:Nx-1
    U_diag(i) = 1 + alpha - (alpha/2)^2 / U_diag(i-1);
    % main diagonal in upper depends on previous value of diagonal
    U_upperdiag(i-1) = -0.5 * alpha; % upper diagonal (in U)
    L_lowerdiag(i-1) = -0.5 * alpha / U_diag(i-1);
end
%calculation for each tau
for j = 2:(Nt+1)
    %extract the endpoints of each column
    % here we calculate bm_CN = Z_m_1 (3 terms) + f_m_0
    b_m(1) = 0.5*alpha * (V(1,j) + V(1,j-1) + V(3,j-1)) + (1-alpha)*V(2,j-1);
    % here we calculate bm_CN = Z_m_Nx-1 (3 terms) + f_m_Nx
    b_m(Nx-1) = 0.5*alpha*(V(Nx+1,j) + V(Nx+1,j-1) + V(Nx-1,j-1)) + (1-alpha)*V(Nx,j-1);
    % now calculate for all the other points in the array
    for i = 2:(Nx-2)
        b_m(i) = 0.5*alpha * (V(i,j-1) + V(i+2,j-1)) + (1-alpha) * V(i+1,j-1);
    end
    v_m(1) = b_m(1);
    %do the forward substitution
    for i = 2:(Nx-1)
        v_m(i) = b_m(i) - v_m(i-1)*L_lowerdiag(i-1);
    end
    V(Nx,j) = v_m(Nx-1)/U_diag(Nx-1);
    %do the backward substitution
    for i = (Nx-1) : -1 : 2
        V(i,j) = (v_m(i-1) + 0.5*alpha * V(i+1,j))/U_diag(i-1);
    end
end

for i = 1:Nx+1 %10:10
    currentS = Sn(i);
    %convert to x
    currentV = log(currentS/K);
    closestX = interp1((xn(1,:))', V(:,Nt+1), currentV,'nearest');
    for j = 1:Nt+1
        currentT = Tm(j);
        currentTau = 0.5*sigma^2*(T-currentT);
    end
end

```

```

        Onm(i,j) = exp(-0.125*(((k-1)^2)+4*k)* 2 * currentTau)*(currentS ^ (0.5*(1-k)))*
        K^(0.5*(1+k))* closestX;
    end
end

```

```

%converting vector back into the correct order
xn = [Nminusinf:dx:Nplusinf]; % vectors for x
disp('Function put Blsprice2 terminated successfully!');

```

## ***Appendix A (contd.):***

### **Part 2 American Puts:**

```
function [Onm, Tm, Sn, taum, xn] = putAM(S0, K, r, sigma, T, Nt, Smin, Smax, Nx, eps)
```

```
oldloops = 10000;
```

```
omega = 1.5;
```

```
domega = 0.05;
```

```
%%rewriting conditions to correspond to the new variables
```

```
k = r / (0.5 * sigma^2); % getting the value of k
```

```
Sn = [Smin:(Smax-Smin)/Nx:Smax]; %evolution of the Stock
```

```
Tm = [0:T/Nt:T]; %evolution of the time
```

```
Nminusinf = log(Smin/K);
```

```
Nplusinf = log(Smax/K);
```

```
tau = 0.5*sigma^2*T;
```

```
x0 = log(S0/K);
```

```
% obtaining the transformed bounds of the log steps of the Stock price evolution
```

```
dtau = 0.5 * sigma^2 * T / Nt;
```

```
dx = (Nplusinf - Nminusinf)/Nx;
```

```
alpha = dtau/((dx)^2);
```

```
Nplus = (Nt+1)*dx;
```

```
Nminus = -(Nt+1)*dx;
```

```
N = round((Nplus-Nminus)/dx);
```

```
% vectors for tau and for x
```

```
taum = [0:dtau:dtau*Nt]; % vectors for tau
```

```
xn = x0 + [Nplus:-dx:Nminus]';
```

```
% set up the boundary conditions boundary condition for Nplus and Nminus
```

```
v(1,2:Nt+1) = zeros(1,Nt);
```

```
v(N+1,2:Nt+1) = (exp(0.5*(k-1)*xn(N+1) + 0.25*((k+1)^2+4*k).*taum(2:Nt+1)'));
```

```
%initial condition
```

```
v(1:N+1,1) = max(exp(0.5*(k-1).*xn)-exp(0.5*(k+1).*xn),0);
```

```
% set up matrix M_CN with the three diagonals
```

```
% in this case we dont have to do LU factorization
```

```
M_CN = diag((1 + alpha)*ones(N-1,1),0) + diag((-0.5*alpha)*ones(N-2,1),1) + diag((-0.5*alpha)*ones(N-2,1),-1);
```

```
% Calculate v for each tau
```

```
for j = 2 : Nt+1
```

```
    %b_m = zeros(N-1,1);
```

```
    % here we calculate b_m_CN =Z_m_1 (3 terms) + f_m_0
```

```
    b_m(1)= alpha*0.5*((g_fn(xn(1),taum(j-1),k) + g_fn(xn(1),taum(j),k)- 2*v(2,j-1)+ v(3,j-1))) + v(2,j-1) ;
```

```
    b_m(N-1)= alpha*0.5*((g_fn(xn(N+1),taum(j-1),k)- 2*v(N,j-1) + g_fn(xn(N+1),taum(j),k) + v(N-1,j-1))) + v(N,j-1);
```

```
    for i = 2:N-2
```

```

    b_m(i) = alpha*0.5*(v(i+2,j-1) - 2*v(i+1,j-1) + v(i,j-1)) + v(i+1,j-1) ;
end;
%now do the psor relaxation to get the value of v
[v(2:N,j), newloops] = psor(M_CN, alpha, N-1, b_m, g_fn(xn(2:N),taum(j),k), omega, eps
, g_fn(xn(2:N),taum(j),k));
if (newloops > oldloops)
    domega = domega * -1.0;
end
omega = omega + domega;
oldloops = newloops;
end;

for i = 1:Nx+1 %10:10
    currentS = Sn(i);
    %convert to x
    currentV = log(currentS/K);
    closestX(i) = interp1((xn(:,1))', v(:,Nt+1), currentV, 'nearest');
    for j = 1:Nt+1
        currentT = Tm(j);
        currentTau = 0.5*sigma^2*(T-currentT);
        Onm(i,j) = exp(-0.125*(((k-1)^2)+4*k)* 2 * currentTau)*(currentS ^ (0.5*(1-k)))*
K^(0.5*(1+k))* closestX(i);
    end
end
xn = xn(1:N-1)';
xn = fliplr(xn);
disp('Function putAM terminated successfully!');

```

```

function [v, loops] = psor(M_CN, alpha, n, b_m,initial_est,w,eps ,transpayoff);
loops = 0;
initguess= initial_est+ 1;
nextguess = initial_est;
count = 1; % Number of iterations;
done = 0;
while (done ==0)
    initguess= nextguess;
    for i = 1:n
        %y_m+1,k+1_n = 1/(1+alpha) * [Zm_n - (-0.5*alpha) * (u_m+1,k+1_n-1
        % + u_m+1,k_n+1)
        y(i) = 1/(1+alpha)*(b_m(i) - M_CN(i,1:i-1)*nextguess(1:i-1) -
M_CN(i,i+1:n)*initguess(i+1:n));
        %u_m+1,k+1_n = max(u_m+1,k_n) + omega* (y_m+1,k+1_n - u_m+1,k_n),
        %g_m+1,n
        nextguess(i) = max(initguess(i) + w*(y(i) - initguess(i,1)), transpayoff(i) );
    end;
    if (max(abs( nextguess-initguess)) < eps)
        done = 1;
    end;
end;

```

```
else
    done = 0;
end
loops = loops + 1;
end;
v = nextguess;
```