

[Help](#)

```
#include "dup1d_std.h"

static double Call_OrnsteinUhlenbeck(double s,
    double k,double t,double r,double divid,double sigma)

{
    double v,y1,y2,val;

    v=sigma*sqrt((exp(-2*divid*t)-(exp(-2*r*t)))/(2
        *(r-divid)));
    y1=(s*exp(-divid*t)-k*exp(-r*t))/v;
    y2=(-s*exp(-divid*t)-k*exp(-r*t))/v;

    /*Price*/
    val=(s*exp(-divid*t)-k*exp(-r*t))*N(y1)+
        (s*exp(-divid*t)+exp(-r*t)*k)*N(y2)+
        v*(nd(y1)-nd(y2));

    /* printf("%f\n",(exp(-divid*t))*N(y2));*/

    return val;
}

/*tridiagonal matrices*/
struct tridiag{
    int size;
    double *subdiag; /*size-1*/
    double *diag;    /*size*/
    double *updiag;  /*size-1*/
};

/*bidiagonal matrices*/
struct bidiag{
    int size;
    double *subdiag; /*size-1*/
    double *diag;    /*size*/
};

/*tridiagonal system Tx=b */
```

```

struct tridiagSystem{
    int size;
    struct tridiag *T;
    double *b;
};

/*bidiagonal system Tx=b */
struct bidiagSystem{
    int size;
    struct bidiag *T;
    double *b;
};

/*equals the tridiag matrix B to the tridiag matrix A when they already have the same size*/
static void affectOperator(struct tridiag *A, struct tridiag *B){

    int i;

    for (i=0;i<=B->size-2;i++){
        B->subdiag[i] = A->subdiag[i];
        B->diag[i] = A->diag[i];
        B->updiag[i] = A->updiag[i];
    }
    B->diag[B->size-1] = A->diag[B->size-1];
}

static int find_index(double x, double *grid, int size, int start){
    /* OUTPUT
    - returns the index i such that grid[i]<=x<grid[i+1]
    INPUT
    - x (x = y or T)
    - grid = grid of discretized values of x, grid = x_0,...,x_n (x = y or T)
    - size = size of grid - 1 (M for T_grid, N for y_grid)

```

```

        - start = index at which we start the search of i */

int k;

if (grid[size] <= x)
    return size-1;
else{
    k=start;
    while (k<size && grid[k]<=x)
        k++;
    return k-1;
}
}

/*changes the tridiagonal system to a triangular system*/
static void tridiagToBidiagSyst(struct bidiagSystem *U, struct tridiagSystem *S){

    int size,i;

    size = S->size;
    U->T->diag[size-1] = S->T->diag[size-1];
    U->b[size-1] = S->b[size-1];

    for (i=size-2;i>=0;i--){
        U->T->diag[i] = S->T->diag[i] - S->T->updiag[i]*S->T->subdiag[i]/(U->T->diag[i+1]);
        U->b[i] = S->b[i] - S->T->updiag[i]*U->b[i+1]/(U->T->diag[i+1]);
        U->T->subdiag[i] = S->T->subdiag[i];
    }

}

/*changes a tridiagonal matrix to a bidiagonal one*/
static void tridiagToBidiagMat(struct bidiag *B,
```

```

    struct tridiag *T){

int size,i;

size = T->size;
B->diag[size-1] = T->diag[size-1];

for (i=size-2;i>=0;i--){
    B->diag[i] = T->diag[i] - T->updiag[i]*T->subdiag[i]/(B->diag[i+1]);
    B->subdiag[i] = T->subdiag[i];
}

}

/*solves the bidiagonal system at each time step*/
/
static void solveSyst(double *sol, struct bidiag
    System *S){

int size,i;

size = S->size;
sol[0] = (S->b)[0]/(S->T->diag[0]);

for (i=1;i<=size-1;i++){
    sol[i] = (S->b[i] - S->T->subdiag[i-1]*sol[i-1])/
        (S->T->diag[i]);
}

static void tridiagMatrixInv(double **inv, struct
    tridiag *T){

struct bidiagSystem *U;
struct tridiagSystem *V;
double *x_j;
int i,j,size;

size = T->size;

```

```

/*memory allocation */

/* memory allocation for the tridiagonal system
   m V*/
V = (struct tridiagSystem *) malloc(sizeof(struct
   tridiagSystem));
V->T = (struct tridiag *) malloc(sizeof(struct
   tridiag));
V->T->size = size;
V->T->subdiag = (double *) malloc((size-1)*sizeof(double));
V->T->diag = (double *) malloc(size*sizeof(double));
V->T->updiag = (double *) malloc((size-1)*sizeof(double));
V->b = (double *) malloc(size*sizeof(double));
V->size = size;

/* memory allocation for the bidiagonal system
   U*/
U = (struct bidiagSystem *) malloc(sizeof(struct
   bidiagSystem));
U->T = (struct bidiag *) malloc(sizeof(struct
   bidiag));
U->T->subdiag = (double *) malloc((size-1)*sizeof(double));
U->T->diag = (double *) malloc(size*sizeof(double));
U->T->size = size;
U->b = (double *) malloc(size*sizeof(double));
U->size = size;

/*memory alloc for x_j*/
x_j = (double *) malloc(size*sizeof(double));

/*initialization of V->T*/
affectOperator(T,V->T);

for (j=0;j<size;j++){
    /*initialisation of V->b */

```

```

    for(i=0;i<j;i++)
        V->b[i] = 0;
    V->b[j] = 1;
    for(i=j+1;i<size;i++)
        V->b[i] = 0;
    /*computation of the jth column of the inverse */
    tridiagToBidiagSyst(U,V);
    solveSyst(x_j,U);
    for (i=0;i<size;i++){
        inv[i][j] = x_j[i];
    }
}

/* memory desallocation */
free(V->T->subdiag);
V->T->subdiag = NULL;
free(V->T->diag);
V->T->diag = NULL;
free(V->T->updiag);
V->T->subdiag = NULL;
free(V->T);
V->T = NULL;
free(V->b);
V->b = NULL;
free(V);
V = NULL;

free(U->T->subdiag);
U->T->subdiag = NULL;
free(U->T->diag);
U->T->diag = NULL;
free(U->T);
U->T = NULL;
free(U->b);
U->b = NULL;
free(U);
U = NULL;

free(x_j);
x_j = NULL;

```

```

}
```

```

static void tridiagMatMult(double **res, struct
    tridiag *A, double **B){
    /* res = B*A */
    int i,j,size;

    size = A->size;
    for (i=0;i<size;i++){
        res[i][0] = A->diag[0]*B[i][0] + A->subdiag[0]
            *B[i][1];
        for (j=1;j<size-1;j++)
            res[i][j] = A->updiag[j-1]*B[i][j-1] + A->
            diag[j]*B[i][j] + A->subdiag[j]*B[i][j+1];
        res[i][size-1] = A->updiag[size-2]*B[i][size-
            2] + A->diag[size-1]*B[i][size-1];
    }
}
```

```

}
```

```

static void discretizeSigma(double (*volatility)(
    double,double,int), double **sigmaCoarseGrid, int n,
    int m, double *y_coarseGrid, double *T_coarseGrid,
    int sigma_type){
    /* OUTPUT
        - sigmaCoarseGrid = coarse grid of discretiz
        ed values of sigma
        INPUTS
        - volatility = function defining the volatil
        ity
        - n,m = size of the coarse grid
        - y_coarseGrid = array of dicretized values
        y (for the coarse grid)
        - T_coarseGrid = array of dicretized values
        T (for the coarse grid) */

    int i,j;
    double y_min,y_max,t_0,T_max;
```

```

y_min = y_coarseGrid[0];
y_max = y_coarseGrid[n];
t_0 = T_coarseGrid[0];
T_max = T_coarseGrid[m];

for (i=0;i<=n;i++)
    for(j=0;j<=m;j++)
        sigmaCoarseGrid[i][j] = volatility(T_coars
        eGrid[j],exp(y_coarseGrid[i]),sigma_type);
}

static double invtanh(double x, double a, double
    b){
    /*fonction invtanh (inverse de a*tanh(y-b)) */
    /* returns y so that a*tanh(y-b) = x          */
    return b+.5*log((a+x)/(a-x));
}

static void buildFineGrid(double *y_fineGrid,
    double *T_fineGrid, int N, int M, double t_0, double
    T_max, double y_min, double y_max, double S_0,
    int gridType){
    /* builds the arrays of discretized values of y
    and T for the fine grid
    OUTPUTS:
    - y_fineGrid = array of discretized values
    of y for the fine grid
    - T_fineGrid = array of discretized values
    of y for the fine grid
    - (N,M) = size of the fine grid
    - t_0 = time origin
    - T_max = time horizon
    - y_min = log(S_min)
    - y_max = log(S_max)
    - gridType = type of the y_fineGrid (0 for
    regular, 1 for tanh)
    */

```



```

double K,H, valInvtnh;
int i,l;

K = (T_max-t_0)/M; /* size of time step*/
for (i=0;i<M+1;i++)
    T_fineGrid[i] = t_0+K*i;

/* steps for the fine grid : */
H = (y_max-y_min)/N;

for (i=0;i<N+1;i++)
    y_fineGrid[i] = y_min+H*i;

if (gridType){ /* tanh grid */
    i=1;
    valInvtnh = invtnh(y_min+H,y_max,log(S_0));
    while (i<N && valInvtnh <= y_min){
        i = i+1;
        valInvtnh = invtnh(y_min+i*H,y_max,log(S_0));
    }

    /* the i first values (from 0 to i-1) of inv
    tanh(y_min+i*H,y_max,log(S_0)) */
    /* are below y_min so for the i first values
    of y_tanhGrid[.], we do a uniform */
    /* discretization so that y_coarseGrid[0] = y
    _min */
    for (l=i-1;l>=0;l--)
        y_fineGrid[l] = valInvtnh - (i-l)*(valInv
        tanh - y_min)/i;

    while (i<N && valInvtnh < y_max){
        y_fineGrid[i] = valInvtnh;
        i = i+1;
        valInvtnh = invtnh(y_min+i*H,y_max,log(S_0));
    }
}

```

```

    }
    /* once the image of  $y_{\min}+iH$  is greater than  $y_{\max}$ , we do a uniform */
    /* discretization until N, so that  $y_{\text{coarseGrid}}[N] = y_{\max}$  */
    for (l=i; l<=N; l++)
         $y_{\text{fineGrid}}[l] = y_{\text{fineGrid}}[i-1] + (l-i+1) * (y_{\max} - y_{\text{fineGrid}}[i-1]) / (N-i+1);$ 

}

}

static double f(int optionType, double y, double S_0){
    /* condition for  $T=t_0$ 
    OUTPUT :
    - returns  $f(y, S_0)$  (condition for  $T=t_0$ 
    INPUTS :
    - optionType = 1 for a call, 0 for a put
    - y : log of the price of the asset
    - S_0 : price of the asset at  $t_0$  */

    if (optionType==1)
        return  $(S_0 - \exp(y) >= 0) ? S_0 - \exp(y) : 0;$ 
    else
        return  $(\exp(y) - S_0 >= 0) ? \exp(y) - S_0 : 0;$ 
}

static void buildOperator(struct tridiag *A,
    double r, double q, double S_0, int j, int N, double *
    *sigmaFineGrid, double *y_fineGrid){
    /* builds the discretized operator A

    OUTPUT:
    - A : discretized operator (tridiagonal
    matrix) of size N-1
    INPUTS:

```

```

- r : RF rate
- q : dividends
- S_0 : price of the asset at t_0
- j : index of time step
- N : number of space (price) steps for the fine grid
- sigmaFineGrid : fine grid of the values of sigma
- y_fineGrid : discretized values of y (for the fine grid) */

double x,h_i,h_i_1;
int i;

i=1;
h_i = y_fineGrid[i+1]- y_fineGrid[i];
h_i_1 = y_fineGrid[i] - y_fineGrid[i-1];
x = .5*(r-q+pow(sigmaFineGrid[i][j],2)/2);
A->diag[i-1] = pow(sigmaFineGrid[i][j],2)/(h_i+h_i_1)*(1/h_i + 1/h_i_1) + x*(1/h_i_1 - 1/h_i) + q;
A->updiag[i-1] = x/h_i - pow(sigmaFineGrid[i][j],2)/((h_i+h_i_1)*h_i);
for (i=2;i<=N-2;i++){
    h_i = y_fineGrid[i+1]- y_fineGrid[i];
    h_i_1 = y_fineGrid[i] - y_fineGrid[i-1];
    x = .5*(r-q+pow(sigmaFineGrid[i][j],2)/2);
    A->subdiag[i-2] = -x/h_i_1 - pow(sigmaFineGrid[i][j],2)/((h_i+h_i_1)*h_i_1);
    A->diag[i-1] = pow(sigmaFineGrid[i][j],2)/(h_i+h_i_1)*(1/h_i + 1/h_i_1) + x*(1/h_i_1 - 1/h_i) + q;
    A->updiag[i-1] = x/h_i - pow(sigmaFineGrid[i][j],2)/((h_i+h_i_1)*h_i);
}

/* i=N-1 */
h_i = y_fineGrid[N]- y_fineGrid[N-1];
h_i_1 = y_fineGrid[N-1] - y_fineGrid[N-2];
x = .5*(r-q+pow(sigmaFineGrid[N-1][j],2)/2);

```

```

A->subdiag[N-3] = -x/h_i_1 - pow(sigmaFineGrid[
    N-1][j],2)/((h_i+h_i_1)*h_i_1);
A->diag[N-2] = pow(sigmaFineGrid[N-1][j],2)/(h_
    i+h_i_1)*(1/h_i + 1/h_i_1) + x*(1/h_i_1 - 1/h_i)
    + q;

}

static void buildTridiagSystem(int optionType,
    struct tridiagSystem *S, struct tridiag *A, struct
    tridiag *A_prev, double *u_prev, double k,
    double theta, double condLim){
/* builds the data of the tridiagonal system S
    (Mat*X = b, with Mat tridiagonal matrix, b rig
    ht hand side vector)
    OUTPUT:
    - S : tridiagonal system
    INPUTS:
    - optionType: type of the option (1 for
    call, 0 for put)
    - A : current discretized operator

    - A_prev : previous discretized operator

    - u_prev : vector of prices computed in the
    previous iteration
    - k : size of time step

    - theta : parameter of the finite differenc
    e scheme
    - condLim : limit condition (depending on th
    e time of the option) */

int size,i;

size = A->size;

```

```

S->b[0] = (1-theta*k*A_prev->diag[0])*u_prev[0]
        - theta*k*A_prev->updiag[0]*u_prev[1];

if (optionType == 1) /* call option */
    S->b[0] = S->b[0] + condLim;

S->T->subdiag[0] = (1-theta)*k*A->subdiag[0];
S->T->diag[0] = 1+(1-theta)*k*A->diag[0];
S->T->updiag[0] = (1-theta)*k*A->updiag[0];

for (i=1;i<size-1;i++){
    S->b[i] = (-theta*k*A_prev->subdiag[i-1])*u_
        prev[i-1] + (1-theta*k*A_prev->diag[i])*u_prev[i]
        - theta*k*(A_prev->updiag[i])*u_prev[i+1];
    S->T->subdiag[i] = (1-theta)*k*A->subdiag[i];
    S->T->diag[i] = 1+(1-theta)*k*A->diag[i];
    S->T->updiag[i] = (1-theta)*k*A->updiag[i];
}

S->T->diag[size-1] = 1+(1-theta)*k*A->diag[siz
    e-1];
S->b[size-1] = -theta*k*A_prev->subdiag[size-2]
    *u_prev[size-2] + (1-theta*k*A_prev->diag[size-1
    ])*u_prev[size-1];
if (optionType == 0) /*put option*/
    S->b[size-1] = S->b[size-1] + condLim;
}

static void solve(int optionType, double **res,
    double S_0, int N, int M, double r, double q, double
    theta, double (*f)(int,double,double), double **
    sigmaFineGrid, double *y_fineGrid, double *T_fine
    Grid){
/* solves the EDP using the finite difference
method
OUTPUT :
- res : fine grid of prices to be computed (
size N*M)
INPUTS :
- optionType : type of the option (1 for

```

```

call, 0 for put)
- S_0 : price of the asset at t_0

- N : number of space steps of the fine grid
- M : number of time steps of the fine grid

- r : RF rate

- q : dividends

- theta : parameter of the finite difference scheme
- sigma : fine grid of values of sigma

- y_fineGrid : discretized values of y (for the fine grid)
- T_fineGrid : discretized values of t (for the fine grid)      */

/* declarations */
double *u_prev,*u_next;
double h_i, h_i_1;
int i,j;
struct tridiag *op, *op_prev;
struct tridiagSystem *S1;
struct bidiagSystem *S2;
double x,h_0,h_1,alpha,alpha2,gamma,gamma2,condLim,k;
double t_0,T_max,y_min,y_max;

y_min = y_fineGrid[0];
y_max = y_fineGrid[N];
t_0 = T_fineGrid[0];
T_max = T_fineGrid[M];

/* memory allocation for the operator (of type struct tridiag *) */
op = (struct tridiag *) malloc(sizeof(struct

```

```

    tridiag));
op->subdiag = (double *) malloc((N-2)*sizeof(
    double));
op->diag = (double *) malloc((N-1)*sizeof(
    double));
op->updiag = (double *) malloc((N-2)*sizeof(
    double));
op->size = N-1;

/* memory allocation for the operator (of type
    struct tridiag *) */
op_prev = (struct tridiag *) malloc(sizeof(stru
    ct tridiag));
op_prev->subdiag = (double *) malloc((N-2)*siz
    eof(double));
op_prev->diag = (double *) malloc((N-1)*sizeof(
    double));
op_prev->updiag = (double *) malloc((N-2)*sizeo
    f(double));
op_prev->size = N-1;

/* memory allocation for the tridiagonal syste
    m S1*/
S1 = (struct tridiagSystem *) malloc(sizeof(str
    uct tridiagSystem));
S1->T = (struct tridiag *) malloc(sizeof(struct
    tridiag));
S1->T->subdiag = (double *) malloc((N-2)*sizeo
    f(double));
S1->T->diag = (double *) malloc((N-1)*sizeof(
    double));
S1->T->updiag = (double *) malloc((N-2)*sizeof(
    double));
S1->T->size = N-1;
S1->b = (double *) malloc((N-1)*sizeof(double))
    ;
S1->size = N-1;

/* memory allocation for the bidiagonal system
    S2*/
S2 = (struct bidiagSystem *) malloc(sizeof(stru

```

```

    ct bidiagSystem));
S2->T = (struct bidiag *) malloc(sizeof(struct
    bidiag));
S2->T->subdiag = (double *) malloc((N-2)*sizeof
    f(double));
S2->T->diag = (double *) malloc((N-1)*sizeof(
    double));
S2->T->size = N-1;
S2->b = (double *) malloc((N-1)*sizeof(double))
    ;
S2->size = N-1;

```

```

/* memory allocation for u_prev and u_next */
u_prev = (double *) malloc((N-1)*sizeof(double)
    );
u_next = (double *) malloc((N-1)*sizeof(double)
    );

```

```

/* initialization of u */

```

```

if (optionType == 1)
    res[0][0] = S_0;
else
    res[0][0] = 0;

for (i=1;i<N;i++){
    u_prev[i-1] = f(optionType,y_fineGrid[i],S_0)
        ;
    res[i][0] = u_prev[i-1];
}
res[N][0] = f(optionType,y_max,S_0);

```

```

/*builds the initial discretized operator*/
buildOperator(op_prev,r,q,S_0,0,N,sigmaFineGrid,
    y_fineGrid);
for (j=1;j<=M;j++){
    buildOperator(op,r,q,S_0,j,N,sigmaFineGrid,y_

```



```

fineGrid); /*builds the discretized operator*/
/* initial condition for y=y_min*/
h_0 = y_fineGrid[1]-y_fineGrid[0];
h_1 = y_fineGrid[2]-y_fineGrid[1];
k = T_fineGrid[j]-T_fineGrid[j-1];
x = .5*(r-q+pow(sigmaFineGrid[1][j-1],2)/2);
alpha = -x/h_0 - pow(sigmaFineGrid[1][j-1],2)/
((h_1+h_0)*h_0);
gamma = x/h_0 - pow(sigmaFineGrid[1][j-1],2)/
((h_1+h_0)*h_0);
x = .5*(r-q+pow(sigmaFineGrid[1][j],2)/2);
alpha2 = -x/h_0 - pow(sigmaFineGrid[1][j],2)/
((h_1+h_0)*h_0);
gamma2 = x/h_0 - pow(sigmaFineGrid[1][j],2)/
((h_1+h_0)*h_0);
if (optionType==1) /*call*/
    condLim = - S_0*k*(theta*alpha*exp(-q*(T_fi
neGrid[j-1]-t_0)) + (1-theta)*alpha2*exp(-q*(T_fi
neGrid[j]-t_0)))+ exp(y_min)*k*(theta*alpha*exp(-
r*(T_fineGrid[j-1]-t_0)) + (1-theta)*alpha2*exp(-
r*(T_fineGrid[j]-t_0)));
else /*put*/
    condLim = - k*theta*gamma*(-S_0*exp(-q*(T_
fineGrid[j-1]-t_0))+exp(y_max)*exp(-r*(T_fineGri
d[j-1]-t_0))) + k*(1-theta)*gamma2*(-S_0*exp(-q*(
T_fineGrid[j]-t_0))+exp(y_max)*exp(-r*(T_fineGri
d[j]-t_0)));

buildTridiagSystem(optionType,S1,op,op_prev,u
_prev,k,theta,condLim); /*builds the tridiagonal
system*/

tridiagToBidiagSyst(S2,S1); /* changes the
tridiag system to a bidiagonal one */
solveSyst(u_next,S2); /* solves the bidiagon
al system*/

/*we update u_prev and stock the results in
the matrix res */

```

```

if (optionType == 1){ /*call*/
    res[0][j] = S_0*exp(-q*(T_fineGrid[j]-t_0))
; /* = limit condition for y = y_min for a call
option*/
    res[N][j] = 0; /* = limit condition for y =
y_max for a call option*/
}else{
    res[0][j] = 0; /* = limit condition for y =
y_min for a put option*/
    res[N][j] = -S_0*exp(-q*(T_fineGrid[j]-t_0))
+exp(y_max)*exp(-r*(T_fineGrid[j]-t_0)); /* =
limit condition for y = y_max for a put option*/
}

for (i=1;i<N;i++){
    h_i = y_fineGrid[i+1]-y_fineGrid[i];
    h_i_1 = y_fineGrid[i]-y_fineGrid[i-1];
    res[i][j] = u_next[i-1];
    u_prev[i-1] = u_next[i-1];
}

/* update of op_prev */
affectOperator(op,op_prev);

}

/*free memory space*/
free(S1->T->subdiag);
free(S1->T->diag);
free(S1->T->updiag);
free(S1->T);
free(S1->b);
free(S1);

free(S2->T->subdiag);
free(S2->T->diag);
free(S2->T);
free(S2->b);
free(S2);

```

```

    free(op->subdiag);
    free(op->diag);
    free(op->updiag);
    free(op);

    free(op_prev->subdiag);
    free(op_prev->diag);
    free(op_prev->updiag);
    free(op_prev);

    free(u_prev);
    free(u_next);

}

/* Main Procedure */
static int Implicit(NumFunc_1 *p,double S_0,
    double T,double r,double q,int sigma_type,int n,int m,
    double *ptprice,double *ptdelta)
{
    /*declarations*/
    int i,optionType,gridType;
    double y_max,y_min,T_max,t_0,theta,K,temp;
    double **res; /* res contains the values of the
        option today */
    /*for different strikes and maturities
        */
    double **grid; /* discretized values of sigma */
    /
    double *y_grid, *T_grid; /* grids of the uniformly
        discretized values of y and t used for the
        small grid of sigma*/

    double priceph,pricenh,inc=0.00001;

    /* Resolution of DUPIRE EDP */
    if((p->Compute) == &Call)
        optionType=1;
    else if((p->Compute) == &Put)
        optionType=0;

```

```

/* gridType = type of the y_fineGrid (0 for re
   gular, 1 for tanh) */
gridType=1;
K=p->Par[0].Val.V_DOUBLE;

/*Transformation to BLACK-SCHOLES EDP*/
temp=S_0;
S_0=K;
K=temp;

temp=r;
r=q;
q=temp;

if(optionType==1)
    optionType=0;
else optionType=1;

/*Cranck-Nicholson*/
theta = .5;
y_max = log(10.*S_0); /*S_max=exp(y_max)*/
y_min = -y_max; /*S_max=exp(y_min)*/
T_max = T;
t_0 = 0; /* time origin */

/*memory allocation for y_grid and t_grid*/
y_grid = (double *) malloc((n+1)*sizeof(double)
    );
T_grid = (double *) malloc((m+1)*sizeof(double)
    );

/* memory allocation for the matrix sigma = gri
   d*/
grid = (double **) malloc((n+1)*sizeof(double *
    ));
for (i=0;i<=n;i++)
    grid[i] = (double *) malloc((m+1)*sizeof(
        double));

```

```

/* memory allocation for the matrix res*/
res = (double **) malloc((n+1)*sizeof(double *))
);
for (i=0;i<=n;i++)
    res[i] = (double *) malloc((m+1)*sizeof(
double));

/*initialization of y_grid and t_grid*/
buildFineGrid(y_grid,T_grid,n,m,t_0,T_max,y_mi
n,y_max,S_0,gridType);

/* discretization of the analytical function si
gma_func */
discretizeSigma(volatility,grid,n,m,y_grid,T_g
rid,sigma_type);

/* solves Dupire PDE */
solve(optionType,res,S_0,n,m,r,q,theta,f,grid,y
_grid,T_grid);

/* gives the option price for K and T*/
i = find_index(log(K),y_grid,n,0);

/*for(i=0;i<=n;i++)
    printf("%f %f {n",exp(y_grid[i]),res[i][m]-
Call_OrnsteinUhlenbeck(S_0,exp(y_grid[i]),T,r,q,15.)
);*/

/* linear linterpolation */
*ptprice= res[i+1][m]-(y_grid[i+1]-log(K))*(re
s[i+1][m]-res[i][m])/(y_grid[i+1]-y_grid[i]);

/*printf("%f %f %f{n",exp(y_grid[i]),res[i][m],
Call_OrnsteinUhlenbeck(S_0,K,T,r,q,15.));*/

/*Delta*/
i= find_index(log(K)+inc,y_grid,n,0);
priceph=res[i+1][m]-(y_grid[i+1]-(log(K)+inc))*
(res[i+1][m]-res[i][m])/(y_grid[i+1]-y_grid[i]);

```

```

i= find_index(log(K)-inc,y_grid,n,0);
pricenh=res[i+1][m]-(y_grid[i+1]-(log(K)-inc))*
    (res[i+1][m]-res[i][m])/(y_grid[i+1]-y_grid[i]);

*ptdelta=(priceph-pricenh)/(2.*inc*K);

    /*****
    *****/
/*          memory desallocation
          */
/*****/
    *****/

for (i=0;i<=n;i++)
    free(grid[i]);
free(grid);

free(T_grid);
free(y_grid);

for (i=0;i<=n;i++)
    free(res[i]);
free(res);

return OK;
}

int CALC(FD_Implicit)(void *Opt,void *Mod,Pricing
    Method *Met)
{
    TYPEOPT* ptOpt=( TYPEOPT*)Opt;
    TYPEMOD* ptMod=( TYPEMOD*)Mod;
    double r,divid;

    r=log(1.+ptMod->R.Val.V_DOUBLE/100.);
    divid=log(1.+ptMod->Divid.Val.V_DOUBLE/100.);

    return Implicit(ptOpt->PayOff.Val.V_NUMFUNC_1,
        ptMod->S0.Val.V_PDOUBLE,

```

```

        ptOpt->Maturity.Val.V_DATE-ptMod->T.Val.
V_DATE,r,divid,ptMod->Sigma.Val.V_INT, Met->Par[0
].Val.V_INT,Met->Par[1].Val.V_INT,
    &(Met->Res[0].Val.V_DOUBLE),&(Met->Res[1]
.Val.V_DOUBLE));
}

int CHK_OPT(FD_Implicit)(void *Opt, void *Mod)
{
    Option* ptOpt=(Option*)Opt;
    TYPEOPT* opt=(TYPEOPT*)(ptOpt->TypeOpt);

    if ((strcmp( ((Option*)Opt)->Name,"CallEuro")==
0)||strcmp( ((Option*)Opt)->Name,"PutEuro")==0)
    )
        return OK;

    return WRONG;
}

static int MET(Init)(PricingMethod *Met)
{
    static int first=1;

    if (first)
    {
        Met->Par[0].Val.V_INT2=1000;
        Met->Par[1].Val.V_INT2=100;

        first=0;
    }

    return OK;
}

PricingMethod MET(FD_Implicit)=
{
    "FD Dupire",
    {"Space StepNumber",INT2,100,ALLOW},{"Time Ste
pNumber",INT2,100,ALLOW}, {" ",END,0,FORBID}},
    CALC(FD_Implicit),

```

```
{{"Price",DOUBLE,100,FORBID},{ "Delta",DOUBLE,10
  0,FORBID},{ " ",END,0,FORBID}},
CHK_OPT(FD_Implicit),
CHK_ok,
MET(Init)
};
```

References