

[Help](#)

```
#include <stdlib.h>
#include "
href../../mod/dup1d/dup1d_std/dup1d_std_h_src.pdfdup1d_std.h"

#if 0 /* Unused */
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)) * cdf_nor(y1) +
          (s * exp(-divid * t) + exp(-r * t) * k) * cdf_nor(y2) +
          v * (pn1_normal_density(y1) - pn1_normal_density(y2));

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

    return val;
}
#endif

/*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
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*/
#if 0
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];
    }
}
#endif

/*solves the bidiagonal system at each time step*/
static void solveSyst(double *sol, struct bidiagSystem *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]);
}

#if 0 /* Unused */
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 V*/
V = malloc(sizeof(struct tridiagSystem));
V->T = malloc(sizeof(struct tridiag));
V->T->size = size;
V->T->subdiag = malloc((size - 1) * sizeof(double));
V->T->diag = malloc(size * sizeof(double));
V->T->updiag = malloc((size - 1) * sizeof(double));
V->b = malloc(size * sizeof(double));
V->size = size;

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

/*memory alloc for x_j*/
x_j = 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;
}
#endif

#if 0 /* Unused */
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->s
    res[i][size - 1] = A->updiag[size - 2] * B[i][size - 2] + A->diag[size - 1]
}
}
#endif

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

    int i, j;

    for (i = 0; i <= n; i++)
        for (j = 0; j <= m; j++)
            sigmaCoarseGrid[i][j] = volatility(T_coarseGrid[j], exp(y_coarseGrid[i]),

}

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,
{
    /* 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, valInvtanh;
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;
    valInvtanh = invtanh(y_min + H, y_max, log(S_0));
    while (i < N && valInvtanh <= y_min)
    {
        i = i + 1;
        valInvtanh = invtanh(y_min + i * H, y_max, log(S_0));
    }

    /* the i first values (from 0 to i-1) of invtanh(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 uni
    /* discretization so that y_coarseGrid[0] = y_min
    for (l = i - 1; l >= 0; l--)

```



```

        y_fineGrid[l] = valInvtanh - (i - 1) * (valInvtanh - y_min) / i;

while (i < N && valInvtanh < y_max)
{
    y_fineGrid[i] = valInvtanh;
    i = i + 1;
    valInvtanh = invtanh(y_min + i * H, y_max, log(S_0));
}
/* once the image of y_min+i*H is greater than y_max, we do a uniform */
/* discretization until N, so that y_coarseGrid[N] = y_max */
for (l = i; l <= N; l++)
    y_fineGrid[l] = y_fineGrid[i - 1] + (l - i + 1) * (y_max - y_fineGrid[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 N)
{
    /* 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);
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);
    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);
}

static void buildTridiagSystem(int optionType, struct tridiagSystem *S, struct t
{
    /* builds the data of the tridiagonal system S (Mat*X = b, with Mat tridiagonal)
    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 difference scheme
- condLim : limit condition (depending on the 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->
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 - the
    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[size - 1];
S->b[size - 1] = -theta * k * A_prev->subdiag[size - 2] * u_prev[size - 2] + (
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

```

```

{
/* 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;
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, y_min, y_max;

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

/* memory allocation for the operator (of type struct tridiag *) */
op = malloc(sizeof(struct tridiag));
op->subdiag = malloc((N - 2) * sizeof(double));
op->diag = malloc((N - 1) * sizeof(double));
op->updiag = malloc((N - 2) * sizeof(double));
op->size = N - 1;

/* memory allocation for the operator (of type struct tridiag *) */
op_prev = malloc(sizeof(struct tridiag));
op_prev->subdiag = malloc((N - 2) * sizeof(double));

```

```

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

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

/* memory allocation for the bidiagonal system S2*/
S2 = malloc(sizeof(struct bidiagSystem));
S2->T = malloc(sizeof(struct bidiag));
S2->T->subdiag = malloc((N - 2) * sizeof(double));
S2->T->diag = malloc((N - 1) * sizeof(double));
S2->T->size = N - 1;
S2->b = malloc((N - 1) * sizeof(double));
S2->size = N - 1;

/* memory allocation for u_prev and u_next */
u_prev = malloc((N - 1) * sizeof(double));
u_next = 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
    /* 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_fineGrid[j - 1] - t_0)
    else /*put*/
        condLim = - k * theta * gamma * (-S_0 * exp(-q * (T_fineGrid[j - 1] - t_0)

    buildTridiagSystem(optionType, S1, op, op_prev, u_prev, k, theta, condLim)

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

```

```

    }

    for (i = 1; i < N; i++)
    {
        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, in
{
    /*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

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 regular, 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 = malloc((n + 1) * sizeof(double));
T_grid = malloc((m + 1) * sizeof(double));

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

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

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

/* discretization of the analytical function sigma_func */
discretizeSigma(premia_local_vol, grid, n, m, y_grid, T_grid, 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

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

/*printf("%f %f %f \n",exp(y_grid[i]),res[i][m],Call_OrnsteinUhlenbeck(S_0,K,T

```

```

/*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] -

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

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

/*****
/*                                memory deallocation
/*****

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, PricingMethod *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, pt
                    &(Met->Res[0].Val.V_DOUBLE), &(Met->Res[1].Val.V_DOUBLE));
}

```

```

}

static 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
        return OK;

    return WRONG;
}

static int MET(Init)(PricingMethod *Met, Option *Opt)
{
    if (Met->init == 0)
    {
        Met->init = 1;

        Met->Par[0].Val.V_INT2 = 1000;
        Met->Par[1].Val.V_INT2 = 100;

    }

    return OK;
}

PricingMethod MET(FD_Implicit) =
{
    "FD_Dupire",
    {"Space StepNumber", INT2, {100}, ALLOW}, {"Time StepNumber", INT2, {100}, AL
    CALC(FD_Implicit),
    {"Price", DOUBLE, {100}, FORBID}, {"Delta", DOUBLE, {100}, FORBID}, {" ", PRE
    CHK_OPT(FD_Implicit),
    CHK_ok,
    MET(Init)
};

```