

[Help](#)

```
#include <stdlib.h>
#include "bs1d_std.h"
#include "error_msg.h"

/* Inversion LU matrice tridiagonale */

static void Initinv_Crout(double *A_inf, double *A_diag, double *A_up, int n_0,
{

    int i;

    L_diag[0] = A_diag[n_0 + 0];
    U_up[0] = A_up[n_0 + 0] / L_diag[0];

    for (i = 1; i <= dim - 2; i++)
    {
        L_inf[i - 1] = A_inf[n_0 + i];
        L_diag[i] = A_diag[n_0 + i] - L_inf[i - 1] * U_up[i - 1];
        U_up[i] = A_up[n_0 + i] / L_diag[i];
    }
    L_inf[dim - 2] = A_inf[n_0 + dim - 1];
    L_diag[dim - 1] = A_diag[n_0 + dim - 1] - L_inf[dim - 2] * U_up[dim - 2];
}

static void inv_Crout(double *v, double *sol, double *A_inf, double *A_diag, dou
{

    int i;

    Initinv_Crout(A_inf, A_diag, A_up, n_0, dim, L_inf, L_diag, U_diag, U_up);

    /* On resout d'abord L sol = v */
    /* descente */
    sol[n_0] = v[n_0] / L_diag[0];
    for (i = 1; i < dim; i++)
        sol[n_0 + i] = (v[n_0 + i] - L_inf[i - 1] * sol[n_0 + i - 1]) / L_diag[i];

    /* Puis on resout U sol(new) = sol */
```

```
/* remontee */
for (i = dim - 2; i >= 0; i--)
    sol[n_0 + i] = (sol[n_0 + i] - U_up[i] * sol[n_0 + i + 1]);

}

/* Construction des matrices */
static void Construit_Matrices_CN(double *Mn_inf, double *Mn_diag, double *Mn_up)
{
    double val1, val2, facmul;
    int i;

    /* Initialisation des matrices */

    /*
        Mn          =( -(1/dt + r/2) Id + 0.5 * A + 0.5 * B )
        Mn_plus_1=( -(1/dt + r/2) Id - 0.5 * A - 0.5 * B )
    */

    /* Mn */
    for (i = 0; i < I + 1; i++)
    {
        Mn_inf[i] = 0.;
        Mn_diag[i] = 0.;
        Mn_up[i] = 0.;
    }

    /* M(n+1) */
    for (i = 0; i < I + 1; i++)
    {
        Mn_plus_1_inf[i] = 0.;
        Mn_plus_1_diag[i] = 0.;
        Mn_plus_1_up[i] = 0.;
    }

    /* Mn */
    facmul = 0.5 * (sigma * sigma / 2);
    for (i = 0; i < I + 1; i++)
    {
        val1 = facmul * (i - 0.5) * (i - 0.5);
```

```
        val2 = facmul * (i + 0.5) * (i + 0.5);
        Mn_inf[i] += val1;
        Mn_diag[i] += -val2 - val1;
        Mn_up[i] += val2;
    }

/* M(n+1) */
facmul = -0.5 * (sigma * sigma / 2);
for (i = 0; i < I + 1; i++)
{
    val1 = facmul * (i - 0.5) * (i - 0.5);
    val2 = facmul * (i + 0.5) * (i + 0.5);
    Mn_plus_1_inf[i] += val1;
    Mn_plus_1_diag[i] += -val2 - val1;
    Mn_plus_1_up[i] += val2;
}

/* Mn */
facmul = 0.5 * ((taux_d_interet - divid - sigma * sigma) / 2);
for (i = 0; i < I + 1; i++)
{
    val1 = facmul * (i - 0.5);
    val2 = facmul * (i + 0.5);
    Mn_inf[i] += -val1;
    Mn_diag[i] += -facmul;
    Mn_up[i] += val2;
}

/* M(n+1) */
facmul = -0.5 * ((taux_d_interet - divid - sigma * sigma) / 2);
for (i = 0; i < I + 1; i++)
{
    val1 = facmul * (i - 0.5);
    val2 = facmul * (i + 0.5);
    Mn_plus_1_inf[i] += -val1;
    Mn_plus_1_diag[i] += -facmul;
    Mn_plus_1_up[i] += val2;
}

/* Mn et M(n+1) */
for (i = 0; i < I + 1; i++)
```

```
    {  
        Mn_diag[i] += -1. / dt - taux_d_interet / 2;  
        Mn_plus_1_diag[i] += -1. / dt + taux_d_interet / 2;  
    }  
}
```

```
static int FixedPoint(double s, NumFunc_1 *p, double tt, double r, double divid  
{  
    double EPSILON = 1.e-10;  
  
    /* Matrices */  
    double *Mn_inf, * Mn_diag, * Mn_up;  
    double *Mn_plus_1_inf, * Mn_plus_1_diag, * Mn_plus_1_up;  
    double *L_diag, * L_inf, * U_up, *U_diag;  
  
    /* Vecteur */  
    double *Fold, * Fnew;  
    double *scd_membre;  
  
    /* Parametres financiers */  
    double T = tt;  
    double S0 = s;  
  
    /* Parametres numeriques */  
    int N = nt; /* nbre de pas de temps */  
    int I = nesp; /* nbre de pas d'espaces */  
    double dt = T / N;  
    double K = p->Par[0].Val.V_DOUBLE;  
    double xmax = 5 * K;  
    double dx = xmax / I;  
  
    /* dans les itérations en temps */  
    int timestep;  
    double CL;  
    int ind_frontiere, dim;  
    int CV;  
    double G;  
  
    /* Calcul du prix */
```

```
double x;
int i;
double a, b, c, d;
double aprime, bprime, cprime, dprime;

/*Memory Allocation*/
L_diag = malloc((I + 1) * sizeof(double));
if (L_diag == NULL)
    return MEMORY_ALLOCATION_FAILURE;
L_inf = malloc((I + 1) * sizeof(double));
if (L_inf == NULL)
    return MEMORY_ALLOCATION_FAILURE;
U_diag = malloc((I + 1) * sizeof(double));
if (U_diag == NULL)
    return MEMORY_ALLOCATION_FAILURE;
U_up = malloc((I + 1) * sizeof(double));
if (U_up == NULL)
    return MEMORY_ALLOCATION_FAILURE;
Mn_inf = malloc((I + 1) * sizeof(double));
if (Mn_inf == NULL)
    return MEMORY_ALLOCATION_FAILURE;
Mn_diag = malloc((I + 1) * sizeof(double));
if (Mn_diag == NULL)
    return MEMORY_ALLOCATION_FAILURE;
Mn_up = malloc((I + 1) * sizeof(double));
if (Mn_up == NULL)
    return MEMORY_ALLOCATION_FAILURE;
Mn_plus_1_inf = malloc((I + 1) * sizeof(double));
if (Mn_plus_1_inf == NULL)
    return MEMORY_ALLOCATION_FAILURE;
Mn_plus_1_diag = malloc((I + 1) * sizeof(double));
if (Mn_plus_1_diag == NULL)
    return MEMORY_ALLOCATION_FAILURE;
Mn_plus_1_up = malloc((I + 1) * sizeof(double));
if (Mn_plus_1_up == NULL)
    return MEMORY_ALLOCATION_FAILURE;
Fold = malloc((I + 1) * sizeof(double));
if (Fold == NULL)
    return MEMORY_ALLOCATION_FAILURE;
Fnew = malloc((I + 1) * sizeof(double));
if (Fnew == NULL)
```

```

    return MEMORY_ALLOCATION_FAILURE;
scd_membre = malloc((I + 1) * sizeof(double));
if (scd_membre == NULL)
    return MEMORY_ALLOCATION_FAILURE;

/* Construction des matrices */
Construit_Matrices_CN(Mn_inf, Mn_diag, Mn_up, Mn_plus_1_inf, Mn_plus_1_diag, M

/* Condition finale */
for (i = 0; i < I + 1; i++)
    Fnew[i] = (p->Compute)(p->Par, i * dx);

/* indice de la frontiere d'exercice : initialement sur K */
ind_frontiere = (int)floor(K / dx);

/* Iterations en temps */

for (timestep = N - 1; timestep >= 0; timestep--)
{
    for (i = 0; i < I + 1; i++)
        Fold[i] = Fnew[i];

    CV = 0; /* test de convergence */

    while (!CV)
    {

        CV = 1;

        dim = I - ind_frontiere - 1; /* dimension du système à résoudre */

        /* second membre */
        for (i = ind_frontiere + 1; i < I; i++)
            scd_membre[i] = Fold[i - 1] * Mn_plus_1_inf[i] + Fold[i] * Mn_plus_1

        /* Definition de la condition aux limites de Dirichlet (à gauche) */
        CL = (p->Compute)(p->Par, ind_frontiere * dx);
        scd_membre[ind_frontiere + 1] -= CL * Mn_inf[ind_frontiere + 1];

        /* calcul de la valeur au temps tn */

```

```

    inv_Crout(scd_membre, Fnew, Mn_inf, Mn_diag, Mn_up, ind_frontiere + 1,

/* on complète à gauche */
for (i = 0; i <= ind_frontiere; i++)
    Fnew[i] = (p->Compute)(p->Par, i * dx);

/* mouvement de la frontiere d'exercice */

if (Fnew[ind_frontiere + 1] < (p->Compute)(p->Par, (ind_frontiere + 1)
{
    CV = 0;
    ind_frontiere++;
    /*    printf("+1 \ n"); */
}

i = ind_frontiere;
G = Fnew[i - 1] * Mn_inf[i] + Fnew[i] * Mn_diag[i] + Fnew[i + 1] * Mn_
G -= Fold[i - 1] * Mn_plus_1_inf[i] + Fold[i] * Mn_plus_1_diag[i] + Fo

if (G > EPSILON && ind_frontiere > 1 && CV)
{
    CV = 0;
    ind_frontiere--;
    /*    printf("-1 \ n"); */
}

} /* de while */

} /* de for (timestep */

/* Calcul du prix et du delta */
for (i = 0; i < I + 1; i++)
    Fnew[i] = Fold[i];

x = S0;
i = (int)floor(x / dx);

/* Interpolation d'ordre 1 */
/*a=((i+1)*dx-x)/dx;
b=(x-i*dx)/dx;*/

```

```

a = (i * dx - x) * ((i + 1) * dx - x) * ((i + 2) * dx - x) / (dx * 2 * dx * 3
b = (x - (i - 1) * dx) * ((i + 1) * dx - x) * ((i + 2) * dx - x) / (dx * dx *
c = (x - (i - 1) * dx) * (x - i * dx) * ((i + 2) * dx - x) / (2 * dx * dx * dx
d = (x - (i - 1) * dx) * (x - i * dx) * (x - (i + 1) * dx) / (3 * dx * 2 * dx

aprime = ((-1.) * ((i + 1) * dx - x) * ((i + 2) * dx - x) + (i * dx - x) * (-1
bprime = ((1.) * ((i + 1) * dx - x) * ((i + 2) * dx - x) + (x - (i - 1) * dx)
cprime = ((1.) * (x - i * dx) * ((i + 2) * dx - x) + (x - (i - 1) * dx) * (1.)
dprime = ((1.) * (x - i * dx) * (x - (i + 1) * dx) + (x - (i - 1) * dx) * (1.)

/*Price*/
*ptprice = a * Fnew[i - 1] + b * Fnew[i] + c * Fnew[i + 1] + d * Fnew[i + 2];

/*Delta */
*ptdelta = (aprime * Fnew[i - 1] + bprime * Fnew[i] + cprime * Fnew[i + 1] + d

free(Mn_inf);
free(Mn_diag);
free(Mn_up);
free(Mn_plus_1_inf);
free(Mn_plus_1_diag);
free(Mn_plus_1_up);
free(L_diag);
free(L_inf);
free(U_up);
free(U_diag);
free(Fold);
free(Fnew);
free(scd_membre);

return OK;
}

int CALC(FD_FixedPoint)(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 FixedPoint(ptMod->S0.Val.V_PDOUBLE, ptOpt->PayOff.Val.V_NUMFUNC_1, ptOp
}

static int CHK_OPT(FD_FixedPoint)(void *Opt, void *Mod)
{
    if ((strcmp(((Option *)Opt)->Name, "CallAmer") == 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 = 1000;

    }

    return OK;
}

PricingMethod MET(FD_FixedPoint) =
{
    "FD_FixedPoint",
    {"SpaceStepNumber", INT2, {100}, ALLOW    }, {"TimeStepNumber", INT2, {100},
    CALC(FD_FixedPoint),
    {"Price", DOUBLE, {100}, FORBID}, {"Delta", DOUBLE, {100}, FORBID} , {" ", PR
    CHK_OPT(FD_FixedPoint),
    CHK_fdiff,
    MET(Init)
};

```