

```
#include <stdlib.h>
#include "cirpp1d_std.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2007+2) //The "#els
static int CHK_OPT(FD_GaussZBO)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(FD_GaussZBO)(void *Opt, void *Mod, PricingMethod *Met)
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else

/* defined in premia_obj.c */
extern char premia_data_dir[MAX_PATH_LEN];
extern char *path_sep;

/*////////////////////////////////////// DONNEES ////////////////////////////////////////
////////////////////////////////////// DONNEES ////////////////////////////////////////
////////////////////////////////////// DONNEES ////////////////////////////////////////

/*////////////////////////////////////// Donnees communes des modèles ////////////////////////////////////////

static char init[] = "initialyield.dat";
static FILE *Entrees;
static double *tm;
static double *Pm;

static int Nvalue;

/*////////////////////////////////////// Donnees propres au modèle CIR ++ ////////////////////////////////////////

static double a;
static double b;
static double rx0;
static double sigma;
```

```
static double FM;
```

```
/*////////////////////////////////////////// Donnees propres pour l'edp //////////////////////////////////
```

```
static struct EDP Edp;
```

```
/*//////////////////////////////////////////  
////////////////////////////////////////// Fin des DONNEES //////////////////////////////////  
//////////////////////////////////////////
```

```
/*//////////////////////////////////////////  
////////////////////////////////////////// Fonctions de CIR++ //////////////////////////////////  
//////////////////////////////////////////
```

```
static int lecture()  
{
```

```
    int i;  
    char ligne[20];  
    char *pligne;  
    double p, tt;  
    char data[MAX_PATH_LEN];
```

```
    sprintf(data, "%s%s%s", premia_data_dir, path_sep, init);  
    Entrees = fopen(data, "r");
```

```
    if (Entrees == NULL)  
    {  
        printf("Le FICHER N'A PU ETRE OUVERT. VERIFIER LE CHEMIN\ n");  
    }  
    else {}
```

```
    i = 0;  
    pligne = ligne;  
    Pm = malloc(100 * sizeof(double));  
    tm = malloc(100 * sizeof(double));  
    /* printf("OUVERTURE\ n");*/
```

```
while (1)
{
    pligne = fgets(ligne, sizeof(ligne), Entrees);
    if (pligne == NULL) break;
    else
    {
        sscanf(ligne, "%lf t=%lf", &p, &tt);

        Pm[i] = p;
        tm[i] = tt;
        i++;
    }
}

fclose(Entrees);

Nvalue = i;
return i;
}

int SetTimegrid_EDP(struct EDP *Meth, int n, double T)
{
    int i;

    Meth->Ngrid = n;
    Meth->Tf = T;

    Meth->t = malloc((Meth->Ngrid + 1) * sizeof(double));

    for (i = 0; i < Meth->Ngrid + 1; i++)
    {
        Meth->t[i] = i * Meth->Tf / Meth->Ngrid;
    }

    return 1;
}
```

```
static double mu_r(double s, double r)
{
    double mu;
    mu = a * (b - r);
    return mu;
}

static double sigma_r(double s, double r)
{
    double sigm;
    sigm = sigma * sqrt(r);
    return sigm;
}

static double bond(double T)
{
    /* in the cir++ model, the read bond price */
    double POT;
    int i = 0;

    if (T > 0)
    {
        if (FM > 0)
        {
            POT = exp(-FM * T);
        }
        else
        {
            while (tm[i] < T && i < Nvalue)
            {
                i = i + 1;
            }

            if (i == 0)
            {
                POT = 1 * (1 - T / tm[0]) + Pm[0] * (T / tm[0]);
            }
            else
            {
                POT = 1 * (1 - T / tm[i]) + Pm[i] * (T / tm[i]);
            }
        }
    }
    else
    {
        POT = 1;
    }
}
```

```

        {
            if (i < Nvalue)
            {
                POT = Pm[i - 1] * (tm[i] - T) / (tm[i] - tm[i - 1]) + Pm[i] *
            }
            else
            {
                POT = Pm[i - 1] + (T - tm[i - 1]) * (Pm[i - 1] - Pm[i - 2]) /
            }
        }
    }
else
{
    POT = 1;
}
/*printf("P(0,%lf)=%lf\ n", T, POT);*/
return POT;
}

static double Shift(double s)
{
    double alpha;
    double x, y, c;
    double fm;

    c = sqrt(a * a + 2 * sigma * sigma);

    if (s - 0.5 * INC > 0)
    {
        fm = (log(bond(s - 0.5 * INC)) - log(bond(s + 0.5 * INC))) / INC;
    }
    else
    {
        fm = -log(bond(INC)) / INC;
    }

    x = exp(s * c);
    y = 2 * c + (a + c) * (x - 1);

    alpha = 2 * a * b * (x - 1) / y + rx0 * 4 * c * c * x / (y * y);

```

```

    alpha = fm - alpha;

    return alpha;
}

/*static int DeleteMod(void)
{

    return 1;
}*/

/*//////////////////////////////////// Fin des fonctions de CIR++ //////////////////////////////////////
//////////////////////////////////// Fonctions de l'edp //////////////////////////////////////

static int indiceTime(struct EDP *Meth, double s)
{
    int i = 0;

    if (Meth->t == NULL)
    {
        printf("FATALE ERREUR, PAS DE GRILLE DE TEMPS !");
    }
    else
    {
        while (Meth->t[i] <= s && i <= Meth->Ngrid)
        {
            i++;
        }
    }
    return i - 1;
}

/*static int add(struct EDP *Meth, double T)
{
    int i, j, boo;
    double* tmp;
    i=0;
    tmp= malloc((Meth->Ngrid+1)*sizeof(double));

```

```

    boo=1;
    for(j=0; j<Meth->Ngrid+1; j++){tmp[j]=Meth->t[j];}
    if(Meth->t==NULL){boo=0;}
    else
    {

    i=0;

    while(Meth->t[i]<T)
    {
    i++;
    if(Meth->t[i]==T){boo=0;}
    }

    if(boo==1)
    {
    Meth->Ngrid=Meth->Ngrid+1;
    free(Meth->t);
    Meth->t= malloc((Meth->Ngrid+1)*sizeof(double));
    for(j=0; j<i; j++){Meth->t[j]=tmp[j];}
    Meth->t[i]=T;
    for(j=i+1; j<Meth->Ngrid+1; j++){Meth->t[j]=tmp[j-1];}
    }
    }
    free(tmp);
    return i;

}*/

/*static int supp(struct EDP *Meth, double T)
{
    int boo;
    int i, j;
    double* tmp;
    i=0;
    tmp= malloc((Meth->Ngrid+1)*sizeof(double));
    boo=0;
    if(Meth->t==NULL){boo=0;} else {

```

```

    for(j=0; j<Meth->Ngrid+1; j++){tmp[j]=Meth->t[j];}

    i=0;

    while(Meth->t[i]<T)
    {
        i++;
        if(Meth->t[i]==T){boo=1;}
    }

    if(boo==1)
    {
        Meth->Ngrid=Meth->Ngrid-1;
        free(Meth->t);
        Meth->t= malloc((Meth->Ngrid+1)*sizeof(double));
        for(j=0; j<i; j++){Meth->t[j]=tmp[j];}

        for(j=i; j<Meth->Ngrid+1; j++){Meth->t[j]=tmp[j+1];}
    }
    free(tmp);
    return boo;

}

*/

/* static int DeleteTimegrid(struct EDP *Meth)
* {
*   free(Meth->t);
*   return 1;
* } */

double OPTIONr_EDP(struct EDP *Meth, double r, double s, double T0)
{
    int i, n0, ns;

```



```

double xs, theta, dr, price;
double int_alpha = 0;

dr = Meth->dx;
ns = indiceTime(Meth, s);
n0 = indiceTime(Meth, T0);
i = 0;

if (Meth->t == NULL)
{
    printf("FATAL ERROR IN OPTIONr(), IL FAUT INITIALISER TIMEGRID AVEC SetTim
}

while (ns + i < n0)
{
    int_alpha = int_alpha + (Shift(Meth->t[ns + i])) * (Meth->t[ns + i + 1] -
    i++;
}
int_alpha = exp(-int_alpha);

i = 1;

xs = r - Shift(s);

if (xs < 0)
{
    printf("x(s) IS NON POSITIVE, OUT OF RANGE OF EDP COMPUTION, rx0 MUST BE C
}
if (xs > Meth->Rm)
{
    printf("x(s) IS TOO BIG, OUT OF RANGE OF EDP COMPUTION, rx0 MUST BE CHANGE
}

while (i * dr < xs && i < Meth->nx - 1)
{
    i++;
}
theta = i - xs / dr;
if (theta > 1)
{
    theta = 1;
}

```

```

    }

    price = int_alpha * (theta * Meth->Payoffunc[ns][i - 1] + (1 - theta) * Meth->Payoffunc[ns][i]);

    return price;
}

static double OPTION(struct EDP *Meth, double T0)
{
    double dr, theta, int_alpha = 0;
    int i, n0;
    double Price;

    dr = Meth->dx;
    n0 = indiceTime(Meth, T0);

    if (Meth->t == NULL)
    {
        Price = -1;
        printf("FATAL ERROR IN OPTION(), IL FAUT INITIALISER TIMEGRID AVEC SetTimeGrid\n");
    }

    int_alpha = 0;

    for (i = 0; i < n0; i++)
    {
        int_alpha = int_alpha + Shift(Meth->t[i]) * (Meth->t[i + 1] - Meth->t[i]);
    }
    int_alpha = exp(-int_alpha);

    i = 0;

    while (i * dr < rx0 && i < Meth->nx - 1)
    {
        i++;
    }
    theta = i - rx0 / dr;

    Price = int_alpha * (theta * Meth->Payoffunc[0][i - 1] + (1 - theta) * Meth->Payoffunc[0][i]);

    return Price;
}

```

```

}
```

```

static void resolutionPayoff(struct EDP *Meth, double s, double T0, int am)
{
    double *X;
    double *Y;
    int i, j, n0, nx;
    char *sorties = "Solution.dat";
    FILE *fich;

    fich = fopen(sorties, "w");

    n0 = indiceTime(Meth, T0);
    nx = Meth->nx;
    X = malloc(nx * sizeof(double));
    Y = malloc(nx * sizeof(double));

    for (j = 0; j < nx; j++)
    {
        X[j] = Meth->Payofffunc[n0][j];
    }

    j = 0;
    while (Meth->t[n0 - j] > s)
    {
        multiplytridiag(Meth->M2, X, Y, nx);

        tridiagsolve(Meth->M1, X, Y, nx);

        for (i = 1; i < 11; i++)
        {
            fprintf(fich, "%f ", X[i * (nx / 10) - 1]);
        }
        fprintf(fich, "\n");
        /*American Case*/
        if (am)
        {

```

```

        for (i = 0; i < nx; i++)
        {
            X[i] = MAX(X[i], Meth->Payofffunc[n0 - j - 1][i]);
        }
    }

    for (i = 0; i < nx; i++)
    {
        Meth->Payofffunc[n0 - j - 1][i] = X[i];
    }

    j++;
}

free(X);
fclose(fich);
}

static void assembleMat(struct EDP *Meth)
{
    double x, dt, dx, dd;
    int i, j, nx;
    nx = Meth->nx;

    Meth->Payofffunc = malloc((Meth->Ngrid + 1) * sizeof(double *));
    for (i = 0; i <= Meth->Ngrid; i++)
    {
        Meth->Payofffunc[i] = malloc(nx * sizeof(double));
    }
    for (i = 0; i <= Meth->Ngrid; i++)
    {
        for (j = 0; j < nx; j++)
        {
            Meth->Payofffunc[i][j] = 0;
        }
    }

    Meth->M1 = malloc(nx * sizeof(double *));
    Meth->M2 = malloc(nx * sizeof(double *));

```

```

for (i = 0; i < nx; i++)
{
    Meth->M1[i] = malloc(nx * sizeof(double));
}
for (i = 0; i < nx; i++)
{
    Meth->M2[i] = malloc(nx * sizeof(double));
}

```

```

for (i = 0; i < nx; i++)
{
    for (j = 0; j < nx; j++)
    {
        Meth->M1[i][j] = 0.;
    }
}
for (i = 0; i < nx; i++)
{
    for (j = 0; j < nx; j++)
    {
        Meth->M2[i][j] = 0.;
    }
}

```

```

dt = Meth->t[1] - Meth->t[0];
x = 0;
dx = Meth->dx;
dd = dt / dx;

```

```

Meth->M1[0][0] = 1.0 - 0.5 * (- mu_r(0, 0) * dd + 0.5 * sigma_r(0, 0) * sigma_r(0, 0) * dd)
Meth->M1[0][1] = -0.5 * (mu_r(0, 0) * dd - sigma_r(0, 0) * sigma_r(0, 0) * dd)
Meth->M1[0][2] = 0.5 * (0.5 * sigma_r(0, 0) * sigma_r(0, 0) * dd / dx);

```

```

Meth->M2[0][0] = 1.0 + 0.5 * (- mu_r(0, 0) * dd + 0.5 * sigma_r(0, 0) * sigma_r(0, 0) * dd)
Meth->M2[0][1] = 0.5 * (mu_r(0, 0) * dd - sigma_r(0, 0) * sigma_r(0, 0) * dd)
Meth->M2[0][2] = 0.5 * (0.5 * sigma_r(0, 0) * sigma_r(0, 0) * dd / dx);

```

```

for (i = 1; i < nx - 1; i++)
{

```

```

x = x + dx;

Meth->M1[i][i - 1] = 0.5 * (-0.5 * sigma_r(0, x) * sigma_r(0, x) * dd / dx
Meth->M1[i][i] = 1. + 0.5 * (sigma_r(0, x) * sigma_r(0, x) * dd / dx + x *
Meth->M1[i][i + 1] = 0.5 * (-0.5 * sigma_r(0, x) * sigma_r(0, x) * dd / dx

Meth->M2[i][i - 1] = 0.5 * (0.5 * sigma_r(0, x) * sigma_r(0, x) * dd / dx
Meth->M2[i][i] = 1. - 0.5 * (sigma_r(0, x) * sigma_r(0, x) * dd / dx + x *
Meth->M2[i][i + 1] = 0.5 * (0.5 * sigma_r(0, x) * sigma_r(0, x) * dd / dx

}
x = x + dx;

Meth->M1[nx - 1][nx - 1] = 1;
Meth->M1[nx - 1][nx - 2] = -0;
Meth->M2[nx - 1][nx - 1] = 0;
Meth->M2[nx - 1][nx - 2] = 0;

/*
Meth->M1[nx-1][nx-1]=1. + 0.5*( sigma_r(0,x)*sigma_r(0,x)*dd/dx );
Meth->M1[nx-1][nx-2]=0.5*( -0.5*sigma_r(0,x)*sigma_r(0,x)*dd/dx + 0.5*mu_r(0,
Meth->M2[nx-1][nx-1]=1. - 0.5*( sigma_r(0,x)*sigma_r(0,x)*dd/dx );
Meth->M2[nx-1][nx-2]=0.5*( 0.5*sigma_r(0,x)*sigma_r(0,x)*dd/dx - 0.5*mu_r(0,

printf("MATRICE ASSEMBLEE\ n");*/
}

/*static void DeleteEDP(struct EDP* Meth)
{
DeleteTimegrid(Meth);
}*/

/*//////////////////////////////////// Fin des fonctions de l'edp //////////////////////////////////////

/*//////////////////////////////////// Fonctions des differents payoff //////////////////////////////////////

void initPayoff1_EDP(struct EDP *Meth, double T0)

```

```

{
    int i, j, n0, N;

    N = Meth->nx;
    n0 = indiceTime(Meth, T0);

    for (i = 0; i < n0; i++)
    {
        for (j = 0; j < N; j++)
        {
            Meth->Payofffunc[i][j] = 0;
        }
    }
    for (j = 0; j < N; j++)
    {
        Meth->Payofffunc[n0][j] = 1.;
    }
}

static void initPayoffZB0(struct EDP *Meth, double T0, NumFunc_1 *p)
{
    int i, j, n0, N;
    double int_alpha = 0;

    N = Meth->nx;
    n0 = indiceTime(Meth, T0);
    i = 0;

    if (Meth->t == NULL)
    {
        printf("FATAL ERROR IN OPTIONr(), IL FAUT INITIALISER TIMEGRID AVEC SetTim
    }

    while (n0 + i < Meth->Ngrid)
    {
        int_alpha = int_alpha + (Shift(Meth->t[n0 + i])) * (Meth->t[n0 + i + 1] -
        i++;
    }
    int_alpha = exp(-int_alpha);

```

```

initPayoff1_EDP(Meth, Meth->Tf);
resolutionPayoff(Meth, 0, Meth->Tf, 0);
for (i = 0; i <= n0; i++)
{
    for (j = 0; j < N; j++)
    {
        Meth->Payofffunc[i][j] = (p->Compute)(p->Par, int_alpha * Meth->Payoffu
    }
}

}

/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Fin des fonctions %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

static int zbo_cirpp1d(int flat_flag, double a0, double b0, double t0, double si
{
    int i, n_price;

    a = a0;
    sigma = sigma0;
    b = b0;
    rx0 = rc;

    Edp.Rm = 2;
    Edp.nx = Ns;
    Edp.dx = (Edp.Rm / Edp.nx);

    SetTimegrid_EDP(&Edp, Nt, S0);
    assembleMat(&Edp);

    if (flat_flag == 0)
    {
        FM = rc;
    }
    else
    {
        FM = -1;
    }
}

```



```

    n_price = lecture();
    if (S0 > tm[n_price - 1])
    {
        printf("\ nError : time bigger than the last time value entered in ini
        exit(EXIT_FAILURE);
    }
}

initPayoffZB0(&Edp, T0, p);
resolutionPayoff(&Edp, t0, T0, am);

if (t0 == 0)
{
    *ptprice = OPTION(&Edp, T0);
}
else
{
    *ptprice = OPTIONr_EDP(&Edp, rc, t0, T0);
}

/**ptdelta=0;*/

for (i = 0; i <= Nt; i++)
{
    free(Edp.Payofffunc[i]);
}
free(Edp.Payofffunc);
for (i = 0; i < Ns ; i++)
{
    free(Edp.M1[i]);
}
free(Edp.M1);
for (i = 0; i < Ns ; i++)
{
    free(Edp.M2[i]);
}
free(Edp.M2);
free(Edp.t);

return OK;
}

```

```

int CALC(FD_GaussZB0)(void *Opt, void *Mod, PricingMethod *Met)
{
    TYPEOPT *ptOpt = (TYPEOPT *)Opt;
    TYPEMOD *ptMod = (TYPEMOD *)Mod;

    return zbo_cirpp1d(ptMod->flat_flag.Val.V_INT, ptMod->a.Val.V_DOUBLE, ptMod->b.Val.V_DOUBLE,
        ptMod->Sigma.Val.V_PDOUBLE, MOD(GetYield)(ptMod), ptOpt->BM, ptOpt->OMaturity.Val.V_DATE,
        ptOpt->PayOff.Val.V_NUMFUNC_1,
    }

static int CHK_OPT(FD_GaussZB0)(void *Opt, void *Mod)
{
    if ((strcmp(((Option *)Opt)->Name, "ZeroCouponCallBondEuro") == 0) || (strcmp(
        return OK;
    else
        return WRONG;
}

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

        Met->Par[0].Val.V_INT2 = 300;
        Met->Par[1].Val.V_INT2 = 300;
        Met->Par[2].Val.V_RGDOUBLE = 0.5;

    }
    return OK;
}

PricingMethod MET(FD_GaussZB0) =
{

```

```

"FD_Cirpp1d_ZB0",
{ {"SpaceStepNumber", INT2, {100}, ALLOW }, {"TimeStepNumber", INT2, {100}, A
  {" ", PREMIA_NULLTYPE, {0}, FORBID}
},
CALC(FD_GaussZB0),
{"Price", DOUBLE, {100}, FORBID}/*,{"Delta",DOUBLE,{100},FORBID} */ , {" ", PR
CHK_OPT(FD_GaussZB0),
CHK_ok,
MET(Init)
} ;

```