

[Help](#)

```
#include <stdlib.h>
#include "
href../../mod/cirpp1d/cirpp1d_std/cirpp1d_std_h_src.pdfcirpp1d_std.h"

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

/*MAJOR COMMENTS ARE IN hwtree1dincludes.h - READ FRIST */
/*MAJOR COMMENTS ARE IN hwtree1dincludes.h - READ FRIST */
/*MAJOR COMMENTS ARE IN hwtree1dincludes.h - READ FRIST */

/*////////////////////////////////////// DATAS ////////////////////////////////////////
//////////////////////////////////////

////////////////////////////////////// Commons datas of shift models ////////////////////////////////////////
/*Name of the file where to read P(0, T) of the market.*/
static FILE *Entrees;           /*File variable of the code*/
static double *tm;              /*Times T of maturities read in the fil
static double *Pm;              /*Values of P(0,tm) read in the file*/
static int Nvalue;              /*Number of value read for Pm*/

////////////////////////////////////// Datas specific to CIR++ ////////////////////////////////////////

static double a;                /*Speed of mean reversion of the CIR++
static double b;
```

```

static double rx0;
static double sigma;           /*Volatility of the CIR++ model*/
static double FM;              /*Constant rate, > 0 if no initial marke

/*//////////////////////////////////// Tree data //////////////////////////////////////

struct Tree Tr;                /* The unique tree variable create by Premia fo

/*////////////////////////////////////
//////////////////////////////////// END OF THE DATAS //////////////////////////////////////
////////////////////////////////////

/*////////////////////////////////////
//////////////////////////////////// Functions of HullWhite //////////////////////////////////////
////////////////////////////////////

static double VarTree(double s)
{
    return s * s;
}

int lecture(char *curve)
{

    int i, etat;
    char ligne[20];
    char *pligne;
    double p, tt;

    Entrees = fopen(curve, "r");

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

```

```

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++;
    }
}

etat = fclose(Entrees);

return i;
}

```

```

/*Conditional variation of the short rate at time s knowing the starting rate x0
/*static double Var( double s)
{

double x, V;
x=exp(-a*s);
V=sigma*sqrt( rx0/a*(1-x)*x + b*pow( (1-x)/(2*a), 2));
return V;
}*/

```

```

static double Var_y(double s)
{
    /*Variation of the variable tree y at time s (must be independent of a variable)
    double V;

    V = sigma * sqrt(s) / 2.0;
    return V;
}

/*Expectation at time s starting from the present time*/
/*static double Expect( double s)
{

    double x, E;
    x=exp(-a*s);
    E=rx0*exp(-a*s) + a*b*(1-x);
    return E;
}*/

/*Conditional expectation of the short rate at time s knowing the starting short rate
/*static double ExpectCond( double x0, double s)
{

    double x, E;
    x=exp(-a*s);
    E=x0*exp(-a*s) + a*b*(1-x);
    return E;

} */

static double ExpectCond_y(double x0, double s)
{
    /*Conditional expectation of variable y used in tree at time s starting from time 0
    double E, x00;
    x00 = 0.5 * sqrt(s * (4 * a * b - sigma * sigma) / (2 - a * s));

    E = x0 + ((a * b / 2 - sigma * sigma / 8) / x0 - a * x0 / 2.0) * s;

```

```

    if (x0 < x00)
    {
        E = x00 + ((a * b / 2 - sigma * sigma / 8) / x00 - a * x00 / 2.0) * s;
    }

    return E;
}

```

```

/*static double mu_r( double s, double r)
{

    double mu;
    mu=a*(b-r);
    return mu;
}*/

```

```

/*static double sigma_r( double s, double r)
{

    return sigma*sqrt(r);
}*/

```

```

static double bond(double T)
{
    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
    {
        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;
}

```

```

/* the shift rate of the cir++ model */
/*static double Shift( double s)
{

double alpha;
double x, y, c, delta;

```



```

if (Meth->t == NULL)
{
    printf("FATALE ERREUR, PAS DE GRILLE DE TEMPS !");
}
else
{
    while (i <= Meth->Ngrid && Meth->t[i] <= s)
    {
        i++;
    }
}
return i - 1;
}
/*
static int MatchGrid(struct Tree *Meth, double T)
{
    double rest=1.;
    double i=1.;
    int n=1;
    int nmax=600;

    int N=Meth->Ngrid;

    while(rest>0 && n<=nmax)
    {
        rest=T*n/Meth->Tf - floor(T*n/Meth->Tf);
        n=n+1;
    }
    rest=1.;
    while(rest>0 && n<=nmax)
    {
        rest=N*i/n - floor(N*i/n);
        i=i+1.;
    }
    n=(int)N*i;

    return n;
}*/

```



```

/*static int add(struct Tree *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 Tree *Meth, double T)

```

```

{
int i, j, boo;
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 SetTimegrid(struct Tree *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 int DeleteTimegrid(struct Tree *Meth)
{
    free(Meth->t);
    return 1;
}

static void SetTree(struct Tree *Meth)
{
    int jmin, jmax, jminprev, jmaxprev;
    double x, xi;
    int h, i, j, k, nv;

    double M, sigmai, mujk, Mij, dx;
    if (Meth->t == NULL)
    {
        printf("FATAL ERROR IN SetTree(), SetTimegrid must be used before SetTree\n");
    }

    jmin = 0;
    jmax = 0;
    xi = 0;
    nv = 1;
    /* Allocation of all the tree variable*/
    Meth->pLRij = malloc((Meth->Ngrid + 1) * sizeof(double *));
    Meth->pLPDo = malloc((Meth->Ngrid) * sizeof(double *));

```

```

Meth->pLPMi = malloc((Meth->Ngrid) * sizeof(double *));
Meth->pLPUp = malloc((Meth->Ngrid) * sizeof(double *));
Meth->pLRef = malloc((Meth->Ngrid) * sizeof(int *));
Meth->TSize = malloc((Meth->Ngrid + 1) * sizeof(int));

Meth->pLRij[0] = malloc(sizeof(double));

Meth->pLRij[0][0] = xi;
Meth->TSize[0] = 1;

/* one step backward translation of the tree, there are 3 point in rank 0 for
{
    jmin = -1;
    jmax = +1;
    xi = 0;
    nv = 3;
    free(Meth->pLRij[0]);
    Meth->pLRij[0] = malloc(3 * sizeof(double));
    Meth->pLRij[0][0] = -sqrt(3.) * Var_y(Meth->t[1]);
    Meth->pLRij[0][1] = xi;
    Meth->pLRij[0][2] = +sqrt(3.) * Var_y(Meth->t[1]);
    Meth->TSize[0] = 3;
}

/* iteration on the time step */
for (i = 1; i <= Meth->Ngrid; i++)
{
    sigmai = Var_y(Meth->t[i] - Meth->t[i - 1]);
    dx = sqrt(3.) * sigmai;
    xi = ExpectCond_y(xi, Meth->t[i] - Meth->t[i - 1]);
    jminprev = jmin;
    jmaxprev = jmax;

    M = ExpectCond_y(Meth->pLRij[i - 1][0], Meth->t[i] - Meth->t[i - 1]);
    jmin = pnl_iround((M - xi) / dx) - 1;
    M = ExpectCond_y(Meth->pLRij[i - 1][nv - 1], Meth->t[i] - Meth->t[i - 1]);
    jmax = pnl_iround((M - xi) / dx) + 1;

    Meth->pLPDo[i - 1] = malloc(nv * sizeof(double));
    Meth->pLPMi[i - 1] = malloc(nv * sizeof(double));

```

```

Meth->pLPUp[i - 1] = malloc(nv * sizeof(double));
Meth->pLRef[i - 1] = malloc(nv * sizeof(int));

nv = jmax - jmin + 1;
Meth->>TSize[i] = nv;

Meth->pLRij[i] = malloc(nv * sizeof(double));

for (k = jmin; k <= jmax; k++)
{
    j = k - jmin;

    x = k * dx + xi;
    Meth->pLRij[i][j] = x;

}
for (k = jminprev; k <= jmaxprev; k++)
{
    j = k - jminprev;
    Mij = ExpectCond_y(Meth->pLRij[i - 1][j], Meth->t[i] - Meth->t[i - 1])
    h = pnl_iround((Mij - xi) / dx);

    mujk = Mij - h * dx - xi;

    Meth->pLPUp[i - 1][j] = 1. / 6. + pow(mujk / dx, 2) / 2. + mujk / (2.*
    Meth->pLPMi[i - 1][j] = 2. / 3. - pow(mujk / dx, 2);
    Meth->pLPDo[i - 1][j] = 1. / 6. + pow(mujk / dx, 2) / 2. - mujk / (2.*
    Meth->pLRef[i - 1][j] = h - jmin;

    if (h <= jmin)
    {
        printf("ERROR FATAL JMIN JMAX IN SetTree(), ExpectCond_y() MUST BE
    }
    if (h >= jmax)
    {
        printf("ERROR FATAL JMIN JMAX IN SetTree(), ExpectCond_y() MUST BE
    }

}

```

```

    }

    /*printf("FIN de la construction de l'arbre des taux\ n");*/
}

static void TranslateTree(struct Tree *Meth)
{
    int k, i, j, comp;
    double alpha, sum, eps;

    if (Meth->t == NULL)
    {
        printf("FATAL ERROR IN TranslateTree(), SetTimegrid() and SetTree() must b
    }
    if (Meth->pLRij == NULL)
    {
        printf("FATAL ERROR IN TranslateTree(), SetTimegrid() and SetTree() must b
    }

    eps = Meth->Tf / Meth->Ngrid;
    alpha = -log(bond(eps)) / eps;

    Meth->pLQij = malloc((Meth->Ngrid + 1) * sizeof(double *));
    Meth->pLQij[0] = malloc(3 * sizeof(double));
    Meth->pLQij[0][0] = 0;
    Meth->pLQij[0][1] = 1.;
    Meth->pLQij[0][2] = 0;

    /* Recalculate the 'x' the translated short rate variable in the tree : x=Var
    for (i = 0; i < Meth->Ngrid + 1; i++)
    {
        for (j = 0; j < Meth->TSize[i]; j++)
        {
            Meth->pLRij[i][j] = VarTree(Meth->pLRij[i][j]);
        }
    }
}

```

```

/* Iteration for alpha translation to obtain the real short rate variable r in
for (i = 0; i < Meth->Ngrid; i++)
{

    Meth->P_T = 0.0;
    Meth->pLQij[i + 1] = malloc(Meth->TSize[i + 1] * sizeof(double));

    for (j = 0; j < Meth->TSize[i]; j++)
    {
        Meth->pLRij[i][j] += alpha;
    }

    for (j = 0; j < Meth->TSize[i + 1]; j++)
    {

        sum = 0.0;
        comp = 0;
        for (k = 0; k < Meth->TSize[i]; k++)
        {
            if (Meth->pLRef[i][k] == j - 1)
            {
                sum += (Meth->pLPUp[i][k] * Meth->pLQij[i][k] * exp(-Meth->pL
            }
            if (Meth->pLRef[i][k] == j)
            {
                sum += (Meth->pLPMi[i][k] * Meth->pLQij[i][k] * exp(-Meth->pL
            }
            if (Meth->pLRef[i][k] == j + 1)
            {
                sum += (Meth->pLPDo[i][k] * Meth->pLQij[i][k] * exp(-Meth->pL
            }
        }

        Meth->pLQij[i + 1][j] = sum;
        Meth->P_T = Meth->P_T + sum;
    }

    sum = 0;

```



```

if (Meth->t == NULL)
{
    printf("FATAL ERROR IN Computepayoff(), SetTimegrid() and SetTree() must b
}
if (Meth->pLRij == NULL)
{
    printf("FATAL ERROR IN Computepayoff(), SetTimegrid() and SetTree() must b
}

if (Meth->Payofffunc == NULL)
{
    initPayoff1_tr(Meth, Meth->Tf);
    printf("DEFAULT PAYOFF 1\ n"); /*Payoff 1 par default.*/
}

/* pLQij[i_end][j] register the payoff at expiry time */
for (j = 0; j < Meth->TSize[i_end]; j++)
{
    Meth->pLQij[i_end][j] = Meth->Payofffunc[i_end][j];
}

/* Computation in pLQij[i][j] of the value of payoff at time step i, backward
for (i = i_end - 1; i >= 0; i--)
{
    for (j = 0; j < Meth->TSize[i]; j++)
    {

        ht = 0;
        ht = exp(- Meth->pLRij[i][j] * (Meth->t[i + 1] - Meth->t[i]));
        ht = ht * (Meth->pLPDo[i][j] * (Meth->pLQij[i + 1][ Meth->pLRef[i][j]
            + Meth->pLPMi[i][j] * (Meth->pLQij[i + 1][ Meth->pLRef[i][j]
            +Meth->pLPUp[i][j] * (Meth->pLQij[i + 1][ Meth->pLRef[i][j]

        /* Compare, in case of american, the computed value with the under nex
        if (ht < Meth->Payofffunc[i][j])
        {
            ht = Meth->Payofffunc[i][j];
        }

        Meth->pLQij[i][j] = ht;
    }
}

```

```

    }
}

/* printf("FIN de l'actualisation payoff de l'arbre des taux\ n"); */
}

/*static double ZB(struct Tree* Meth)
{

return Meth->P_T;

}
*/

/*
static double ZBr(struct Tree* Meth, double r, double s)
{
double theta, sum, P_T1, P_T2;
int i, j, jmin, jmax, k, Ns, Nr;

Ns=indiceTime(Meth, s);

P_T1=0;
P_T2=0;
j=0;

while(Meth->pLRij[Ns][j]<r && j<Meth->TSize[Ns]-1)
{
j=j+1;
}

if(j==0){theta=0;}
else{theta=(r-Meth->pLRij[Ns][j-1])/(Meth->pLRij[Ns][j]-Meth->pLRij[Ns][j-1]);}
if(theta>1){theta=1;j=j+1;}

```

```

/
Nr=j-1;
if(Nr<0){Nr=0;}

jmin=Nr;
jmax=Nr;

Meth->pLQij[Ns][Nr]=1;

for(i=Ns; i<Meth->Ngrid; i++)
{
P_T1=0;
for(j=Meth->pLRef[i][jmin]-1; j<=Meth->pLRef[i][jmax]+1; j++)
{
sum=0.0;

for(k=jmin;k<=jmax; k++)
{
if( Meth->pLRef[i][k]== j-1 ){sum+=( Meth->pLPUp[i][k] * Meth->pLQij[i][k] *
if( Meth->pLRef[i][k]== j ){ sum+=( Meth->pLPMi[i][k] * Meth->pLQij[i][k] *
if( Meth->pLRef[i][k]== j+1 ){sum+=( Meth->pLPDo[i][k] * Meth->pLQij[i][k] *
}

Meth->pLQij[i+1][j]=sum;
P_T1+=sum;

}
jmin=Meth->pLRef[i][jmin]-1;
jmax=Meth->pLRef[i][jmax]+1;
}

Nr=Nr+1;
jmin=Nr;
jmax=Nr;

Meth->pLQij[Ns][Nr]=1;
for(i=Ns; i<Meth->Ngrid; i++)
{
P_T2=0;

```

```

for(j=Meth->pLRef[i][jmin]-1; j<=Meth->pLRef[i][jmax]+1; j++)
{
sum=0.0;

for(k=jmin;k<=jmax; k++)
{
if( Meth->pLRef[i][k]+1 == j ){sum+=( Meth->pLPUp[i][k] * Meth->pLQij[i][k] *
if( Meth->pLRef[i][ k ] == j ){sum+=( Meth->pLPMi[i][k] * Meth->pLQij[i][k] *
if( Meth->pLRef[i][k]-1 == j ){sum+=( Meth->pLPDo[i][k] * Meth->pLQij[i][k] *
}

Meth->pLQij[i+1][j]=sum;
P_T2+=sum;

}
jmin=Meth->pLRef[i][jmin]-1;
jmax=Meth->pLRef[i][jmax]+1;
}

Meth->P_T=theta*P_T2 + (1-theta)*P_T1;

return Meth->P_T;
}
*/

static double OPTIONr_tr(struct Tree *Meth, double r, double s)
{
double theta, R_T;
int j, Ns, Nr;

Ns = indiceTime(Meth, s);
j = 0;

while (Meth->pLRij[Ns][j] < r && j < Meth->TSize[Ns] - 1)
{
j++;
}
if (j == 0)

```

```

        {
            theta = 0;
        }
    else
    {
        theta = (r - Meth->pLRij[Ns][j - 1]) / (Meth->pLRij[Ns][j] - Meth->pLRij[Ns][j - 1]);
    }
    if (theta > 1)
    {
        theta = 1;
        j = j + 1;
    }

    Nr = j - 1;

    if (Nr < 0)
    {
        Nr = 0;
    }
    if (j > Meth->TSize[Ns] - 2)
    {
        printf("WARNING : Instantaneous futur spot rate is out of tree\ n");
    }
    if (Nr == 0)
    {
        printf("WARNING : Instantaneous futur spot rate is out of tree\ n");
    }

    R_T = theta * Meth->pLQij[Ns][Nr + 1] + (1 - theta) * Meth->pLQij[Ns][Nr];

    return R_T;
}

static double OPTION(struct Tree *Meth)
{
    return Meth->pLQij[0][1];
}

```

```

static int DeleteTree(struct Tree *Meth)
{
    int i;

    for (i = 0; i < Meth->Ngrid + 1; i++)
    {
        free(Meth->pLRij[i]);
    }
    for (i = 0; i < Meth->Ngrid + 1; i++)
    {
        free(Meth->pLQij[i]);
    }
    for (i = 0; i < Meth->Ngrid; i++)
    {
        free(Meth->pLPDo[i]);
    }
    for (i = 0; i < Meth->Ngrid; i++)
    {
        free(Meth->pLPMi[i]);
    }
    for (i = 0; i < Meth->Ngrid; i++)
    {
        free(Meth->pLPUi[i]);
    }
    for (i = 0; i < Meth->Ngrid; i++)
    {
        free(Meth->pLRef[i]);
    }

    free(Meth->pLRij);
    free(Meth->pLQij);
    free(Meth->pLPDo);
    free(Meth->pLPMi);
    free(Meth->pLPUi);
    free(Meth->pLRef);
    free(Meth->TSize);

    DeleteTimegrid(Meth);
    free(Meth->Payofffunc);
    return 1;
}

```

```
}
```

```
/*static void PrintTree(struct Tree *Meth)
{
    int i, j, etat;
    FILE* fich;
    char* nomfich="arbre.dat";

    fich=fopen(nomfich, "w");
    fprintf(fich, "\ n");
    for(i=0;i<30; i++)
    {
        for(j=0; j<(Meth->TSize)[i]; j++)
        {
            fprintf(fich, "%f ", (Meth->pLRij)[i][j]);
        }
        fprintf(fich, "\ n");
    }

    etat=fclose(fich);
}*/
```

```
static int AddTime(struct Tree *Meth, double T)
{
    int i, j;
    double *tmp;

    if (T < 0)
    {
        printf("Error: I can't add negative times to a tree !\ n");
        return -1;
    }

    if ((Meth->t == NULL) && (T == 0))
    {
        Meth->t = malloc(sizeof(double));
    }
```

```

        Meth->t[0] = 0;
        Meth->Ngrid = 0;
        Meth->Tf = 0;
        return 0;
    }

    if ((Meth->t == NULL) && (T > 0))
    {
        Meth->t = malloc(2 * sizeof(double));
        Meth->t[0] = 0;
        Meth->t[1] = T;
        Meth->Ngrid = 1;
        Meth->Tf = T;
        return 1;
    }

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

    /* we know that t[i]!=T and (i=Ngrid or t[i]>=T) */
    if (Meth->t[i] < T) i = Meth->Ngrid + 1;
    tmp = malloc((Meth->Ngrid + 1) * sizeof(double));
    for (j = 0; j <= Meth->Ngrid; j++) tmp[j] = Meth->t[j];

    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; j++) Meth->t[j] = tmp[j - 1];

    free(tmp);
    return i;
}

```



```

static void DeletePayoff1(struct Tree *Meth, double T0)
{
    int i, n;
    n = indiceTime(Meth, T0);
    for (i = 0; i < n + 1; i++)
        free(Meth->Payofffunc[i]);
}

```

```

static void initPayoffBERMSWAPTION(struct Tree *Meth, double T0, double per, int
{

```

```

    int *ind, *Size, i, j, k, payer_sign;
    double *T, **cumul;

```

```

    /* definition of the exercise dates*/
    T = malloc((m + 1) * sizeof(double));
    for (i = 0; i <= m; i++) T[i] = i * per + T0;
    /*T[0]=T0           : first resetting date of the swap
       T[1],...,T[m]     : payment dates of the swap
       T[0],...,T[n-1]   : exercise dates of the swaption    ---> We suppose m>=n !

```

```

    ind = malloc(n * sizeof(int));
    for (i = 0; i < n; i++)
        ind[i] = indiceTime(Meth, T[i]);
    /* we have: ind[i] = max{ l=0,...,Meth->Ngrid ; Meth->t[l] <= T[i] }*/

```

```

    Size = malloc(n * sizeof(int));
    for (i = 0; i < n; i++)
        Size[i] = Meth->TSize[ind[i]];
    /* at T[i], the tree has Size[i] nodes */

```

```

initPayoff1_tr(Meth, T[m]);
Computepayoff(Meth, T[m]);
freePayoff1_tr(Meth, T[m]);
/* now Meth->pLQij represents P(.,T[m]) */

cumul = malloc(n * sizeof(double *));
for (i = 0; i < n; i++)
{
    cumul[i] = malloc(Size[i] * sizeof(double));

    for (j = 0; j < Size[i]; j++)
        cumul[i][j] = (1 + K * per) * Meth->pLQij[ind[i]][j];
}
/* for the moment, cumul[i] represents (1+K*per)*P(T[i],T[m]) */

for (k = 1; k < m; k++)
{
    initPayoff1_tr(Meth, T[k]);
    Computepayoff(Meth, T[k]);
    freePayoff1_tr(Meth, T[k]);
    /* now Meth->pLQij represents P(.,T[k])*/

    for (i = 0; i < MIN(k, n); i++)
        for (j = 0; j < Size[i]; j++)
            cumul[i][j] += K * per * Meth->pLQij[ind[i]][j];
}
/* now cumul[i] represents P(T[i],T[m]) + K*per*( P(T[i],T[i+1]) +...+ P(T[i],

if (payer) payer_sign = 1;
else payer_sign = -1;

initPayoff1_tr(Meth, T[n - 1]);
for (i = 0; i < n; i++)
    for (j = 0; j < Size[i]; j++)
        Meth->Payoffunc[ind[i]][j] = MAX(payer_sign * (1 - cumul[i][j]) , 0);

```

```

    for (i = 0; i < n; i++) free(cumul[i]);
    free(cumul);
    free(T);
    free(ind);
    free(Size);
}

/*Swaption=Option on Coupon-Bearing Bond*/
/*All details comments for the functions used here are mainly in "hwtree1dinclud
static int bermudanswaption_cirpp1d(int flat_flag, double a0, double b0, double
{
    int m, payer;
    double *T;
    int i;

    m = (int)((T_final - T0) / per);
    if ((p->Compute) == &Put)
        payer = 1;
    else
        /*if ((p->Compute)==&Call)*/
        payer = 0;

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

    /* Flag to decide to read or not the ZC bond prices P(0,T) in "initialyields.d
    /* If P(0,T) not read then P(0,T)=exp(-FM*T) */
    /* If P(0,T) read then rcc becomes the futur knowing rate name here r0 */
    if (flat_flag == 0)
    {
        FM = rc;
    }
    else
    {
        FM = -1;
        Nvalue = lecture(curve);
    }
}

```

```

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

/* T_final is the final time of the tree, N_step is the number of time steps */
SetTimegrid(&Tr, N_step, T_final);

/* add (if necessary) the payment dates T[i] of the swap to the time grid of t
T = malloc((m + 1) * sizeof(double));
for (i = 0; i <= m; i++)
{
    T[i] = i * per + T0;
    AddTime(&Tr, T[i]);
}

/* Allocate and initialize the tree*/
SetTree(&Tr);

/* translate the tree by "alpha" */
TranslateTree(&Tr);

/* Initialize the payoff for Bermudan swaptions */
initPayoffBERMSWAPTION(&Tr, T0, per, n, m, payer, K);

/* Compute the option from the last exercise date T[n-1] to 0 in pLQij */
Computepayoff(&Tr, T[n - 1]);

/* return the result plQij[0][1] or plQij[indiceTime(t0)][] in case of present
if (t0 == 0)
{
    *price = Nominal * OPTION(&Tr);
}
else
{
    *price = Nominal * OPTIONr_tr(&Tr, rc, t0);
}

```

```

    /**delta=(Tr.pLQij[0][2]-Tr.pLQij[0][0])/(Tr.pLRij[0][2]-Tr.pLRij[0][0]);*/

    DeletePayoff1(&Tr, T[n - 1]);
    DeleteTree(&Tr);
    free(T);

    return OK;
}

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

    return bermudanswaption_cirpp1d(ptMod->flat_flag.Val.V_INT, ptMod->a.Val.V_DOUB
        ptMod->Sigma.Val.V_PDOUBLE, MOD(GetYield)(ptMod),
        MOD(GetCurve)(ptMod), ptOpt->BMaturity.Val.V_DATE,
        ptOpt->OMaturity.Val.V_DATE, ptOpt->PayOff.Val.V_PDOUBLE,
        ptOpt->Nominal.Val.V_PDOUBLE, ptOpt->FixedRate.Val.V_PDOUBLE,
        Met->Par[0].Val.V_LONG, &(Met->Res[0].Val.V_DOUBLE));
}

static int CHK_OPT(TR_BERMUDANSWAPTION)(void *Opt, void *Mod)
{
    if ((strcmp(((Option *)Opt)->Name, "PayerBermudanSwaption") == 0) || (strcmp(O
        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_LONG = 140;
    }
}

```

```

    }
    return OK;
}

PricingMethod MET(TR_BERMUDANSWAPTION) =
{
    "TR_Cirpp1d_BERMUDANSWAPTION",
    { {"TimeStepNumber", LONG, {100}, ALLOW},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CALC(TR_BERMUDANSWAPTION),
    {"Price", DOUBLE, {100}, FORBID}/*,{"Delta",DOUBLE,{100},FORBID} */ , {" ", PR
    CHK_OPT(TR_BERMUDANSWAPTION),
    CHK_ok,
    MET(Init)
} ;

```