

Help

```
#include <stdlib.h>
#include <stdio.h>

#include "pnl/pnl_matrix.h"
#include "pnl/pnl_random.h"
#include "pnl/pnl_basis.h"
#include "local_vol_callable.h"

#define eps 0.0000001

typedef struct
{
    double Pbarre;
    double Nbarre;
    double Cbarre;
    double eta;
    double sigma;
    double r;
    double q;
    double gamma0;
    double alpha;
    double R;
    double cbarre;
} param;

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion <
    (2011+2) //The "#else" part of the code will be freely available after the (year of creation of this file + 2)

#else

//definition du drift local
static double b(double t, double x, double spot, param *P)
{
    return P->r-P->q+P->gamma0*pow(spot/x,P->alpha);
}

//definition de la vol locale
static double vol(double t, double x, param *P)
```

```
{
    return P->sigma;
}

static double c(double x, double spot, param *P)
{
    return P->cbarre+P->gamma0*pow(spot/x,P->alpha)*MAX((1-P->eta)*x,P->R);
}

//definition de la barriere basse
static double low(double x, param *P)
{
    return MAX(x,P->Pbarre);
}

//definition de la barriere haute
static double up(double x, param *P)
{
    return MAX(x,P->Cbarre);
}

//définition du payoff
static double g(double x, param *P)
{
    return MAX(x,P->Nbarre);
}

//definition de mu=r+gamma(S) (elle marche)
static double mu(double spot, double x, param *P)
{
    return P->r+P->gamma0*pow(spot/x,P->alpha);
}

//simulation par schéma d'euler de la matrice des
//trajectoires. On obtient une matrice de taille (N+1)*M
//(la fonction marche)
static void simul_asset(PnlMat *asset, int M, int N,
```

```

    double spot, double T, param *P, int type_generator)
{
    double h=T/N;
    int i,j;
    PnlMat *G;
    double Si_1;
    G=pnl_mat_create(0,0);
    pnl_mat_resize(asset,N+1,M);
    pnl_mat_rand_normal(G,N,M,type_generator);
    for(j=0;j<M;j++) {pnl_mat_set(asset,0,j,spot);}
    for(i=1;i<N+1;i++)
    {
        for(j=0;j<M;j++)
        {
            Si_1=pnl_mat_get(asset,i-1,j);
            pnl_mat_set(asset,i,j,Si_1*(1+b((i-1)*h,Si_1,spot,P)*h+vol((i-1)*h,Si_1,P)*sqrt(h)*pnl_mat_get(G,i-1,j)));
        }
    }
    pnl_mat_free(&G);
}

//definition de beta (voir page 10) matrice de taille
//(N+1)*M (elle marche)
static void beta(PnlMat *res, double spot, PnlMat *asset,
    double T, int N, param *P)
{
    int i,j;
    double h=T/N;
    int M=asset->n;
    pnl_mat_resize(res,N+1,M);
    for(j=0;j<M;j++) pnl_mat_set(res,0,j,0);
    for(i=1;i<N+1;i++)
    {
        for(j=0;j<M;j++)
        {
            pnl_mat_set(res,i,j,mu(spot,pnl_mat_get(asset,i-1,j),P));
        }
    }
    pnl_mat_cumsum(res,'r');
}

```

```

    pnl_mat_mult_double(res,-h);
    pnl_mat_map_inplace(res,exp);
}

//création de la matrice v qui représente le prix. Elle est
//de taille (N+1)*M
static void prix(PnlMat *res, int M, int N, PnlMat *asset,
    double spot, double T, param *P, PnlBasis *basis)
{
    int i,j;
    double Sij,mu_ij,v0;
    PnlVect *Si,*V_iplus1,*alpha, *c_iplus1;//(ligne i de la
        matrice)
    PnlMat MSi;
    double h=T/N;
    pnl_mat_resize(res,N+1,M);
    Si=pnl_vect_new();
    c_iplus1=pnl_vect_create(M);
    alpha=pnl_vect_new();
    V_iplus1=pnl_vect_new();
    for(j=0;j<M;j++) pnl_mat_set(res,N,j,g(pnl_mat_get(asset,
        N,j),P));
    for(i=N-1;i>=1;i--)
    {
        for(j=0;j<M;j++) pnl_vect_set(c_iplus1,j,c(pnl_mat_
            get(asset,i+1,j),spot,P)*h);
        pnl_mat_get_row(Si,asset,i);
        pnl_vect_mult_double(Si,1.0/spot);
        pnl_mat_get_row(V_iplus1,res,i+1);
        pnl_vect_plus_vect(V_iplus1,c_iplus1);
        MSi = pnl_mat_wrap_vect(Si);
        pnl_basis_fit_ls(basis,alpha,&MSi,V_iplus1);
        for(j=0;j<M;j++)
        {
            Sij=pnl_mat_get(asset,i,j)/spot;
            mu_ij=mu(spot,spot*Sij,P);
            pnl_mat_set(res,i,j,MIN(up(spot*Sij,P),MAX(low(
                spot*Sij,P),exp(-mu_ij*h)*pnl_basis_eval(basis,alpha,&Sij)))
        );
        }
    }
}

```

```

    }
    pnl_mat_get_row(V_iplus1,res,1);
    for(j=0;j<M;j++) pnl_vect_set(c_iplus1,j,c(pnl_mat_get(
        asset,1,j),spot,P)*h);
    pnl_vect_plus_vect(V_iplus1,c_iplus1);
    v0=pnl_vect_sum(V_iplus1)/M;
    v0=MIN(up(spot,P),MAX(low(spot,P),exp(-mu(spot,spot,P)*h)
        *v0));
    for(j=0;j<M;j++) pnl_mat_set(res,0,j,v0);
    pnl_vect_free(&Si);
    pnl_vect_free(&c_iplus1);
    pnl_vect_free(&alpha);
    pnl_vect_free(&V_iplus1);
}

//creation du vecteur theta, chaque composante est le prem
    ier
//instant sur une trajectoire où le prix vaut la barrière
//haute
static void theta(PnlVectInt *res, int M, int N, PnlMat *V,
    PnlMat *asset, param *P)
{
    int i,j;

    pnl_vect_int_resize(res,M);
    for(j=0;j<M;j++)
    {
        i=0;
        while(((pnl_mat_get(V,i,j)>up(pnl_mat_get(asset,i,j),
            P)+eps)|| (pnl_mat_get(V,i,j)<up(pnl_mat_get(asset,i,j),P)-
            eps))&&(i<N)) i++;
        pnl_vect_int_set(res,j,i);
    }
}

//creation du vecteur tau, chaque composante est le premier
//instant sur une trajectoire où le prix vaut la barrière
//basse
static void tau(PnlVectInt *res, int M, int N, PnlMat *V,

```

```

    PnlMat *asset, param *P)
{
    int i,j;

    pnl_vect_int_resize(res,M);
    for(j=0;j<M;j++)
    {
        i=0;
        while(((pnl_mat_get(V,i,j)>low(pnl_mat_get(asset,i,j)
,P)+eps)|| (pnl_mat_get(V,i,j)<low(pnl_mat_get(asset,i,j),
P)-eps))&&(i<N)) i++;
        pnl_vect_int_set(res,j,i);
    }
}

//creation du vecteur zeta, chaque composante est le premier
//instant sur une trajectoire où le prix vaut soit la bar
rière
//basse soit la barriere haute (inf(tau,theta))
static void zeta(PnlVectInt *res, PnlVectInt *res_tau, PnlV
ectInt *res_theta)
{
    int M=res_tau->size;
    int j;
    pnl_vect_int_resize(res,M);
    for(j=0;j<M;j++)
        pnl_vect_int_set(res,j,MIN(pnl_vect_int_get(res_tau,j),
pnl_vect_int_get(res_theta,j)));
}

static void prix_en_0_ls(double *res_prix, PnlMat *asset,
    int M, int N, double spot, double T, param *P, PnlBasis *basi
s)
{
    PnlMat *V, *res_beta;
    PnlVectInt *res_zeta, *res_tau,*res_theta;
    PnlVect *tmp_prix;
    int i, j, zeta_j, tau_j, theta_j;
    double sprix,s_tmp_prix;
    double h=T/N;

```

```

V=pnl_mat_new();
res_beta=pnl_mat_new();
res_zeta=pnl_vect_int_new();
res_theta=pnl_vect_int_new();
res_tau=pnl_vect_int_new();
tmp_prix=pnl_vect_create(M);
prix(V,M,N,asset,spot,T,P,basis);
tau(res_tau,M,N,V,asset,P);
theta(res_theta,M,N,V,asset,P);
zeta(res_zeta,res_tau,res_theta);
beta(res_beta,spot,asset,T,N,P);
for(j=0;j<M;j++)
{
    s_tmp_prix=0;
    tau_j=pnl_vect_int_get(res_tau,j);
    theta_j=pnl_vect_int_get(res_theta,j);
    zeta_j=pnl_vect_int_get(res_zeta,j);

    if(zeta_j==N)
    {
        pnl_vect_set(tmp_prix,j,pnl_mat_get(res_beta,zeta
_j,j)*g(pnl_mat_get(asset,N,j),P));
    }
    else if ((zeta_j==tau_j)&&tau_j<N)
    {
        pnl_vect_set(tmp_prix,j,pnl_mat_get(res_beta,zeta
_j,j)*low(pnl_mat_get(asset,tau_j,j),P));
    }
    else
    {
        pnl_vect_set(tmp_prix,j,pnl_mat_get(res_beta,zeta
_j,j)*up(pnl_mat_get(asset,theta_j,j),P));
    }
    for(i=1;i<=zeta_j;i++) {s_tmp_prix=s_tmp_prix+h*pnl_
mat_get(res_beta,i,j)*c(pnl_mat_get(asset,i,j),spot,P);}
    pnl_vect_set(tmp_prix,j,pnl_vect_get(tmp_prix,j)+s_tm
p_prix);
}
sprix=pnl_vect_sum(tmp_prix);

pnl_mat_free(&V);

```

```

    pnl_mat_free(&res_beta);
    pnl_vect_int_free(&res_zeta);
    pnl_vect_int_free(&res_tau);
    pnl_vect_int_free(&res_theta);
    pnl_vect_free(&tmp_prix);
    *res_prix=sprix/M;
}

static void no_call_protection(double *res_prix, int M,
    int N, double spot, double T, int gen, int bindex, int m,
    param *P)
{
    PnlMat *asset;
    PnlBasis *basis;
    basis=pnl_basis_create(bindex, m, 1);
    asset=pnl_mat_new();

    simul_asset(asset,M,N,spot,T,P,gen);
    prix_en_0_ls(res_prix,asset,M,N,spot,T,P,basis);
    pnl_basis_free(&basis);
    pnl_mat_free(&asset);
}

/**
 * @param prix (output) contains the price on exit
 * @param Mod (input) a pointer to the model type
 * @param Opt (input) a pointer to the option type
 * @param gen (input) the random number generator index
 * @param bindex (input) the basis index
 * @param m (input) the number of basis functions
 * @param M (input) the number of Monte Carlo samples
 * @param steps (input) the number of discretisation steps
    per day, It must be
 * an integer
 */
int callable_no_call_protection(double *prix, TYPEMOD *Mod,
    TYPEOPT *Opt, int gen, int bindex, int m, int M, int step
    s)
{
    param *P;

```



```

double T=(double) Opt->Maturity.Val.V_INT/365.;
int N=steps*Opt->Maturity.Val.V_INT;//nb dates discrétis
    ation
double spot=Mod->S0.Val.V_PDOUBLE;
P=malloc(sizeof(param));
P->r=log(1.+Mod->Interest.Val.V_DOUBLE/100.);
P->q=log(1.+Mod->Divid.Val.V_DOUBLE/100.);
P->cbarre=Opt->Coupon.Val.V_PDOUBLE;
P->Pbarre=Opt->PutStrike.Val.V_PDOUBLE;//intervient dans
    low
P->Nbarre=Opt->Strike.Val.V_PDOUBLE;//intervient dans le
    payoff g
P->Cbarre=Opt->CallStrike.Val.V_PDOUBLE;//intervient dans
    up
P->sigma=Mod->Sigma.Val.V_PDOUBLE;
P->alpha=1.2;
P->gamma0=0.02;
P->eta=Mod->Eta.Val.V_PDOUBLE;
P->R=Opt->Recovery.Val.V_PDOUBLE;

pnl_rand_init(gen, M, N);

no_call_protection(prix,M,N,spot,T,gen,bindex,m,P);
free(P);
return OK;;
}

#endif

```

References