Help

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

#include "pnl/pnl_matrix.h"
#include "pnl/pnl_random.h"
#include "pnl/pnl_basis.h"
#include  "
href../../../mod/local_vol/local_vol_callable/local_vol_callable_h_src.pdflocal_

#define eps 0.0000001

//CODE POUR COUPON CONSTANT
typedef struct
{
  double Pbarre;
  double Nbarre;
  double Cbarre;
  double Sup;
  double Slow;
  double eta;
  double sigma;
  double r;
  double q;
  double gamma0;
  double alpha;
  double R;
  double cbarre;
} param;

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2011+2) //The "#els

#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((1.0 - P->eta) *
}

//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, para
{
  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 + v
        }
    }
  pnl_mat_free(&G);
}


//rend la matrice des H, chaque colonne correspond à une
//trajectoire. H(0,:) est initialisée à 0. Elle est mise à 1
//dès que le sous jacent dépasse Sup, et remise à 0 dès que
//le sous jacent passe sous Slow.

static void boolean_protection(PnlMatInt *H, PnlMat *asset, int M, int N, param
{
  int i, j;
  double Sij;
  pnl_mat_int_resize(H, N + 1, M);
  for (j = 0; j < M; j++)
    {
      if (pnl_mat_get(asset, 0, j) >= P->Sup) pnl_mat_int_set(H, 0, j, 1);
      else pnl_mat_int_set(H, 0, j, 0);
    }
  for (j = 0; j < M; j++)
    {
      for (i = 1; i < N + 1; i++)
        {
```

```
          Sij = pnl_mat_get(asset, i, j);
          if ((Sij > P->Slow + eps) && (Sij < P->Sup + eps)) pnl_mat_int_set(H,
          else if (Sij >= P->Sup) pnl_mat_int_set(H, i, j, 1);
          else pnl_mat_int_set(H, i, j, 0);
        }
    }
}


//defintion du premier instant où on passe au dessus de la
//barriere (elle marche)
static void theta(PnlVectInt *res, PnlMat *asset, int M, int N, param *P)
{
  int j, i;
  pnl_vect_int_resize(res, M);
  for (j = 0; j < M; j++)
    {
      i = 0;
      while ((pnl_mat_get(asset, i, j) < P->Sup - eps) && (i < N)) i++;
      pnl_vect_int_set(res, j, i);
    }
}

//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
{
  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);
}


//creation du vecteur tau, chaque composante est le premier
//instant avant theta sur une trajectoire où le prix vaut la barrière
//basse, si on dépasse theta, tau vaut N

static void tau(PnlVectInt *res, int M, int N, PnlMat *V, PnlMat *asset, PnlVect
{
  int i, j;

  pnl_vect_int_resize(res, M);
  for (j = 0; j < M; j++)
    {
      i = 0;
      /* printf("low=%f \ n",low(pnl_mat_get(asset,i,j)));
       * printf("V=%f \ n",pnl_mat_get(V,i,j)); */
      while (((pnl_mat_get(V, i, j) > low(pnl_mat_get(asset, i, j), P) + eps) ||
      if (i >= pnl_vect_int_get(res_theta, j)) pnl_vect_int_set(res, j, N);
      else 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 barrière
//basse soit la barriere haute (inf(tau,theta))
static void zeta(PnlVectInt *res, PnlVectInt *res_tau, PnlVectInt *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(

}


static void prix(PnlMat *res, PnlMatInt *H, int M, int N, PnlMat *asset, double
{
  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;
Si = pnl_vect_new();
alpha = pnl_vect_new();
c_iplus1 = pnl_vect_create(M);
V_iplus1 = pnl_vect_new();
pnl_mat_resize(res, N + 1, M);
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 +
    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);
        if (pnl_mat_int_get(H, i, j) == 0)
          {
            pnl_mat_set(res, i, j, MAX(low(spot * Sij, P), exp(-mu_ij * h)*pnl
          }
        else pnl_mat_set(res, i, j, MIN(up(spot * Sij, P), MAX(low(spot * 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),
pnl_vect_plus_vect(V_iplus1, c_iplus1);
v0 = pnl_vect_sum(V_iplus1) / M;
v0 = MAX(low(spot, P), exp(-mu(spot, spot, P) * h) * v0);
for (j = 0; j < M; j++)
  {
    if (pnl_mat_int_get(H, 0, j) == 0) pnl_mat_set(res, 0, j, v0);
    else pnl_mat_set(res, 0, j, MIN(up(spot, P), v0));
  }
pnl_vect_free(&Si);
```

```
  pnl_vect_free(&V_iplus1);
  pnl_vect_free(&c_iplus1);
  pnl_vect_free(&alpha);
}


static void prix_en_0_ls(double *res_prix, PnlMat *asset, int M, int N, double s
{
  PnlMat *V, *res_beta;
  PnlMatInt *H;
  PnlVectInt *res_zeta, *res_tau, *res_theta;
  PnlVect *tmp_prix;
  int j, i, zeta_j, tau_j, theta_j;
  double sprix, s;
  double h = T / N;
  //initialisation
  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);
  H = pnl_mat_int_new();


  //calcul de la matrice H
  boolean_protection(H, asset, M, N, P);
  //calcul du vecteur theta
  theta(res_theta, asset, M, N, P);

  //calcul du prix standard protection
  prix(V, H, M, N, asset, spot, T, P, basis);
  //calcul de tau, zeta et beta
  tau(res_tau, M, N, V, asset, res_theta, P);
  zeta(res_zeta, res_tau, res_theta);
  beta(res_beta, spot, asset, T, N, P);
  //calcul de la somme Monte Carlo
  for (j = 0; j < M; j++)
    {
      s = 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 (tau_j < theta_j)
        {
          pnl_vect_set(tmp_prix, j, pnl_mat_get(res_beta, zeta_j, j)*low(pnl_mat
        }
      else
        {
          pnl_vect_set(tmp_prix, j, pnl_mat_get(res_beta, zeta_j, j)*pnl_mat_get
        }

      for (i = 1; i <= zeta_j; i++) s = s + h * pnl_mat_get(res_beta, i, j) * c(
      pnl_vect_set(tmp_prix, j, pnl_vect_get(tmp_prix, j) + s);
    }
  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_mat_int_free(&H);
  pnl_vect_free(&tmp_prix);
  *res_prix = sprix / M;
}

static double prix_intermittent_protection(int M, int N, double spot, double T,
{
  PnlMat *asset;
  PnlBasis *basis;
  double sol;
  basis = pnl_basis_create(bindex, m, 1);
  asset = pnl_mat_new();

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

```
/**
 * @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_intermittent_protection(double *prix, TYPEMOD *Mod, TYPEOPT *Opt, i
{
  param *P;
  double T = (double) Opt->Maturity.Val.V_INT / 365.;
  int N = steps * Opt->Maturity.Val.V_INT; //nb dates discrétisation
  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;
  P->Sup = Opt->UpperBarrier.Val.V_PDOUBLE;
  P->Slow = Opt->LowerBarrier.Val.V_PDOUBLE;
  /* init */
  pnl_rand_init(gen, M, N);

  *prix = prix_intermittent_protection(M, N, spot, T, gen, bindex, m, P);

  free(P);
  return OK;;
}

#endif
```