Help

```
//Calcul du prix d'une option américaine sur maximum par l'algo sgm en dimension
//polynomes locaux. Ce code fonctionne


#include  "bs2d_std2d.h"
#include "pnl/pnl_random.h"
#include "pnl/pnl_basis.h"
#include "pnl/pnl_vector.h"
#include "pnl/pnl_cdf.h"

/**
 * Characteristics of a product
 */
typedef struct _Product  Product;
struct _Product
{
  double r; /*!< interest rate */
  PnlVect *spot; /*!< spot */
  double T; /*!< maturity */
  PnlVect *sigma; /*!< volatility */
  PnlVect *divid; /*!< dividende */
  double K; /*!< strike */
  double rho; /*!< correlation */
  int N; /*!< number of exercise dates */
  int degree; /*!< degree of the polynomial regression */
  int MC; /*!< number of iterations of the MC procedure */
};

static double g(double x, double y)
{
  return MAX(x, y);
}

/**
 * Computes the payoff of a call in dimension 1
 *
 * @param St value of the underlying asset at time t
 * @param Xt value of the moving window at time t
 * @param t current time
```

```
 * @param P a Product instance
 *
 * @return the payoff
 */
static double payoff_call(double S1t, double S2t, double t, const Product *P)
{
  return  MAX(g(S1t, S2t) - P->K, 0.);
}

/**
 * Computes the first four centered moments of max(X1,X2), where X1 with law N(m
 * sigma0^2), X1 with law N(mu1, *sigma1^2), and corr(X1,X2)=rho
 * @param mu : vecteur des moyennes (mu0,mu1)
 * @param sigma : vecteur des écarts type (sigma0,sigma1)
 * @param corr : correlation between X1 and X2
 *
 * @output : res : the first four centered moments of max(X1,X2)
 */
static void moments_max(PnlVect *res, PnlVect *mu, PnlVect *sigma, double corr)
{
  PnlVect *nu;
  double nu0;
  double nu1;
  double nu2;
  double nu3;
  double mu0 = GET(mu, 0);
  double mu1 = GET(mu, 1);
  double sigma0 = GET(sigma, 0);
  double sigma1 = GET(sigma, 1);
  double mu0_sq = mu0 * mu0;
  double mu1_sq = mu1 * mu1;
  double mu0_3 = mu0_sq * mu0;
  double mu1_3 = mu1_sq * mu1;
  double mu0_4 = mu0_3 * mu0;
  double mu1_4 = mu1_3 * mu1;
  double sigma0_sq = sigma0 * sigma0;
  double sigma1_sq = sigma1 * sigma1;
  double sigma0_3 = sigma0_sq * sigma0;
  double sigma1_3 = sigma1_sq * sigma1;
  double sigma0_4 = sigma0_3 * sigma0;
  double sigma1_4 = sigma1_3 * sigma1;
```

```
  double a = sqrt(sigma0_sq + sigma1_sq - 2.0 * sigma0 * sigma1 * corr);
  double alpha = (mu0 - mu1) / a;
  double PHI_alpha = pnl_cdfnor(alpha);
  double PHI_moins_alpha = pnl_cdfnor(-alpha);
  double phi_alpha = pnl_normal_density(alpha);
  pnl_vect_resize(res, 4);
  nu = pnl_vect_create(4);
  LET(nu, 0) = mu0 * PHI_alpha + mu1 * PHI_moins_alpha + a * phi_alpha;
  LET(nu, 1) = (mu0_sq + sigma0_sq) * PHI_alpha + (mu1_sq + sigma1_sq) * PHI_moi
  LET(nu, 2) = (mu0_3 + 3.0 * mu0 * sigma0_sq) * PHI_alpha + (mu1_3 + 3.0 * mu1
  LET(nu, 3) = (mu0_4 + 6.0 * mu0_sq * sigma0_sq + 3.0 * sigma0_4) * PHI_alpha +
  nu0 = GET(nu, 0);
  nu1 = GET(nu, 1);
  nu2 = GET(nu, 2);
  nu3 = GET(nu, 3);
  LET(res, 0) = nu0;
  LET(res, 1) = nu1 - pow(nu0, 2);
  LET(res, 2) = nu2 - 3.0 * nu0 * nu1 + 2.0 * pow(nu0, 3);
  LET(res, 3) = nu3 - 4.0 * nu0 * nu2 - 3.0 * pow(nu1, 2) + 12.0 * nu1 * pow(nu0
  //LET(res,3)=nu3-4.0*nu0*nu2+6.0*pow(nu0,2)*nu1-3*pow(nu0,4);
  pnl_vect_free(&nu);
}

static double I_infini(int j, PnlVect *moments)
{
  return exp(j * GET(moments, 0) + GET(moments, 1) * j * j / 2.0) * (1.0 + 1.0 /
}

/**
 *
 *
 * @param Alpha (output) coefficients of the decomposition
 * @param Basis basis
 * @param S (input) asset trajectories. An array of vector, each
 * entry of the array being an asset path.
 * @param prix vector of the option prices
 * @param M number of trajectories
 * @param P product instance
 * @param n index of the current time (starts from 0)
 */
static void regression(PnlVect *Alpha, PnlBasis *Basis, PnlMat **S, PnlVect *pri
```

```
{
  int m;
  PnlMat *St;


  //St matrice de taille M*1. SXt=(S1(tn),S2(tn))
  St = pnl_mat_create(P->MC, 1);
  /*
   * Extract the value of S on each path at time n
   */
  for (m = 0 ; m < P->MC ; m++)
    {
      MLET(St, m, 0) = g(MGET(S[m], n, 0), MGET(S[m], n, 1));
    }
  pnl_basis_fit_ls(Basis, Alpha, St, prix);
  pnl_mat_free(&St);
}

/**
 * Draw one path of the model
 *
 * @param S (output) contains one path of the model
 * with size P->N. S matice de taille (N+1)*1
 * built using the gaussian r.v. G.
 * @param P instance of the product
 * @param G standard normal r.v.
 */
static void asset(PnlMat *S, const Product *P, const PnlMat *G)
{
  double h, sqrt_h, drift1, drift2;
  int k;
  MLET(S, 0, 0) = GET(P->spot, 0);
  MLET(S, 0, 1) = GET(P->spot, 1);
  h = P->T / P->N; /* step size between two exercise dates */
  sqrt_h = sqrt(h);
  drift1 = (P->r - GET(P->divid, 0) - GET(P->sigma, 0) * GET(P->sigma, 0) / 2.)
  drift2 = (P->r - GET(P->divid, 1) - GET(P->sigma, 1) * GET(P->sigma, 1) / 2.)
  for (k = 0 ; k < P->N ; k++)
    {
      MLET(S, k + 1, 0) = MGET(S, k, 0) * exp(drift1 + GET(P->sigma, 0) * sqrt_h
      MLET(S, k + 1, 1) = MGET(S, k, 1) * exp(drift2 + GET(P->sigma, 1) * sqrt_h
```

```
    }
}

static void scale_domain(PnlBasis *B, const Product *P)
{
  PnlVect *center, *scale;
  center = pnl_vect_create_from_double(1, 0.0);
  scale = pnl_vect_create_from_double(1, GET(P->spot, 0));
  pnl_basis_set_reduced(B, center, scale);
  pnl_vect_free(&center);
  pnl_vect_free(&scale);
}

/**
 * Compute an optimal strategy using Longstaff Schwarz approach
 *
 * @param S (input) asset trajectories. An array of vector, each
 * entry of the array being an asset path.
 * @param tau (output) an integer vector, optimal stragey
 * @param M number of trajectories
 * @param P product instance
 */
static void strategie_optimale(double *v, PnlVectInt *tau, PnlMat **S, const Pro
{
  int m; /* indice tirage MC */
  int n; /* indice pas de temps */
  int j;/*indice degré du polynome*/
  PnlBasis *Basis;
  double tn, h, s;
  PnlVect *Alpha, *prix, *cont_reg;
  // double sig_sq_2=pow(P->sigma,2)/2.0;
  double r_div0_sig0_sq_2 = P->r - GET(P->divid, 0) - GET(P->sigma, 0) * GET(P->
  double r_div1_sig1_sq_2 = P->r - GET(P->divid, 1) - GET(P->sigma, 1) * GET(P->
  PnlVect *vect_sigma;
  PnlVect *vect_mu;
  PnlVect *moments;
  moments = pnl_vect_create(4);
  vect_sigma = pnl_vect_create(2);
  vect_mu = pnl_vect_create(2);
  pnl_vect_int_resize(tau, P->MC);
  prix = pnl_vect_create(P->MC);
```

```
cont_reg = pnl_vect_create(P->MC);
Basis = pnl_basis_create_from_degree(PNL_BASIS_CANONICAL, P->degree, 1);
scale_domain(Basis, P);
Alpha = pnl_vect_create(Basis->nb_func);

h = P->T / P->N;

/*
 * init at time T
 */
pnl_vect_int_resize(tau, P->MC);
pnl_vect_int_set_int(tau, P->N);
for (m = 0 ; m < P->MC ; m++)
  {
    double S1T = MGET(S[m], P->N, 0);
    double S2T = MGET(S[m], P->N, 1);
    double payoff = payoff_call(S1T, S2T, P->T, P);
    LET(prix, m) = payoff;

  }

regression(Alpha, Basis, S, prix, P->N, P);

for (m = 0 ; m < P->MC ; m++)
  {
    double S1T_1 = MGET(S[m], P->N - 1, 0);
    double S2T_1 = MGET(S[m], P->N - 1, 1);
    LET(vect_sigma, 0) = GET(P->sigma, 0) * sqrt(h);
    LET(vect_sigma, 1) = GET(P->sigma, 1) * sqrt(h);
    LET(vect_mu, 0) = log(S1T_1) + r_div0_sig0_sq_2 * h;
    LET(vect_mu, 1) = log(S2T_1) + r_div1_sig1_sq_2 * h;
    moments_max(moments, vect_mu, vect_sigma, P->rho);

    s = 0;
    for (j = 0; j < Basis->nb_func; j++)
      {
        s += GET(Alpha, j) * pow(1.0 / GET(P->spot, 0), j) * I_infini(j, momen
      }
    //cont_reg contient la valeur de continuation en N-1
    LET(cont_reg, m) = exp(-P->r * h) * s;
  }
```

```
/*
 * Start iterations
 */
for (n = P->N - 1, tn = P->T - h ; n >= 1 ; n--, tn -= h)
  {

    for (m = 0 ; m < P->MC ; m++)
      {
        const double S1t = MGET(S[m], n, 0);
        const double S2t = MGET(S[m], n, 1);
        const double payoff = payoff_call(S1t, S2t, tn, P);
        LET(prix, m) = MAX(payoff, GET(cont_reg, m));

        if (payoff > 0. && GET(cont_reg, m) < payoff)
          {
            pnl_vect_int_set(tau, m, n);
          }
      }

    regression(Alpha, Basis, S, prix, n, P);

    for (m = 0 ; m < P->MC ; m++)
      {
        const double S1t_1 = MGET(S[m], n - 1, 0);
        const double S2t_1 = MGET(S[m], n - 1, 1);
        LET(vect_sigma, 0) = GET(P->sigma, 0) * sqrt(h);
        LET(vect_sigma, 1) = GET(P->sigma, 1) * sqrt(h);
        LET(vect_mu, 0) = log(S1t_1) + r_div0_sig0_sq_2 * h;
        LET(vect_mu, 1) = log(S2t_1) + r_div1_sig1_sq_2 * h;
        moments_max(moments, vect_mu, vect_sigma, P->rho);

        s = 0.0;
        for (j = 0; j < Basis->nb_func; j++)
          {
            s += GET(Alpha, j) * pow(1.0 / GET(P->spot, 0), j) * I_infini(j, m
          }
        //cont_reg contient la valeur de continuation en n-1
        LET(cont_reg, m) = exp(-P->r * h) * s;
      }
    //pnl_vect_print(cont_reg);
```

```
    }

  *v = MAX(payoff_call(GET(P->spot, 0), GET(P->spot, 1), 0., P), GET(cont_reg, 0

  pnl_basis_free(&Basis);
  pnl_vect_free(&cont_reg);
  pnl_vect_free(&Alpha);
  pnl_vect_free(&prix);
  pnl_vect_free(&vect_mu);
  pnl_vect_free(&vect_sigma);
  pnl_vect_free(&moments);
}

int MC_JainOosterlee2D(double s01, double s02, double sig1, double sig2, double
{
  int m;
  double prix, h;
  PnlRng *rng;
  PnlMat **G;
  PnlMat **S;
  PnlVectInt *tau;
  Product P;
  double v;
  PnlVect *spot, *sigma, *divid;

  spot = pnl_vect_create(2);
  LET(spot, 0) = s01;
  LET(spot, 1) = s02;

  sigma = pnl_vect_create(2);
  LET(sigma, 0) = sig1;
  LET(sigma, 1) = sig2;

  divid = pnl_vect_create(2);
  LET(divid, 0) = divid1;
  LET(divid, 1) = divid2;



  P.r = r;//taux d'intérêt
  P.T = T; //maturité
```

```
P.spot = spot;//spot
P.N = N; // nombre de dates
P.sigma  = sigma;//volatilité
P.divid  = divid;//dividende
P.rho = rho; // corrélation
P.degree = degree;//degré de la base de polynomes pour la régression
P.MC  = MC; //nb tirages Monte Carlo
P.K = K;
h = P.T / P.N;

rng = PnlRngArray[rngtype];
pnl_rng_sseed(rng, 0);

/*
 * Init
 */
S = malloc(P.MC * sizeof(PnlMat *));
G = malloc(P.MC * sizeof(PnlMat *));


for (m = 0 ; m < P.MC ; m++)
  {
    S[m] = pnl_mat_create(P.N + 1, 2);
    G[m] = pnl_mat_create(P.N, 2);
    pnl_mat_rng_normal(G[m], P.N, 2, rng);
    asset(S[m], &P, G[m]);
  }

tau = pnl_vect_int_new();

strategie_optimale(&v, tau, S, &P);

prix = 0.;
for (m = 0 ; m < P.MC ; m++)
  {
    const int tau_m = pnl_vect_int_get(tau, m);
    const double Stau1 = MGET(S[m], tau_m, 0);
    const double Stau2 = MGET(S[m], tau_m, 1);
    const double payoff = payoff_call(Stau1, Stau2, tau_m * h, &P);
    prix += exp(-P.r * tau_m * h) * payoff;
  }
```

```
  prix /= P.MC;
  *ptprix = prix;

  /* Free memory */
  pnl_vect_int_free(&tau);
  pnl_vect_free(&spot);
  pnl_vect_free(&sigma);
  pnl_vect_free(&divid);
  for (m = 0 ; m < P.MC ; m++)
    {
      pnl_mat_free(&(S[m]));
      pnl_mat_free(&(G[m]));
    }
  free(S);
  free(G);

  return OK;
}

int CALC(MC_JainOosterlee2D)(void *Opt, void *Mod, PricingMethod *Met)
{
  TYPEOPT *ptOpt = (TYPEOPT *)Opt;
  TYPEMOD *ptMod = (TYPEMOD *)Mod;
  double r, divid1, divid2, prix, prix1, prix2;

  r = log(1. + ptMod->R.Val.V_DOUBLE / 100.);
  divid1 = log(1. + ptMod->Divid1.Val.V_DOUBLE / 100.);
  divid2 = log(1. + ptMod->Divid2.Val.V_DOUBLE / 100.);

  MC_JainOosterlee2D
  (ptMod->S01.Val.V_PDOUBLE,
   ptMod->S02.Val.V_PDOUBLE,
   ptMod->Sigma1.Val.V_PDOUBLE,
   ptMod->Sigma2.Val.V_PDOUBLE,
   ptMod->Rho.Val.V_RGDOUBLE,
   ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
   ptOpt->PayOff.Val.V_NUMFUNC_2->Par[0].Val.V_PDOUBLE,
   divid1,
   divid2,
   r,
   Met->Par[4].Val.V_INT,
```

```
 Met->Par[0].Val.V_LONG,
 Met->Par[1].Val.V_ENUM.value,
 Met->Par[3].Val.V_DOUBLE,
 &prix);

MC_JainOosterlee2D
(ptMod->S01.Val.V_PDOUBLE * (1 + Met->Par[2].Val.V_PDOUBLE),
 ptMod->S02.Val.V_PDOUBLE,
 ptMod->Sigma1.Val.V_PDOUBLE,
 ptMod->Sigma2.Val.V_PDOUBLE,
 ptMod->Rho.Val.V_RGDOUBLE,
 ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
 ptOpt->PayOff.Val.V_NUMFUNC_2->Par[0].Val.V_PDOUBLE,
 divid1,
 divid2,
 r,
 Met->Par[4].Val.V_INT,
 Met->Par[0].Val.V_LONG,
 Met->Par[1].Val.V_ENUM.value,
 Met->Par[3].Val.V_DOUBLE,
 &prix1);

MC_JainOosterlee2D
(ptMod->S01.Val.V_PDOUBLE,
 ptMod->S02.Val.V_PDOUBLE * (1 + Met->Par[2].Val.V_PDOUBLE),
 ptMod->Sigma1.Val.V_PDOUBLE,
 ptMod->Sigma2.Val.V_PDOUBLE,
 ptMod->Rho.Val.V_RGDOUBLE,
 ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
 ptOpt->PayOff.Val.V_NUMFUNC_2->Par[0].Val.V_PDOUBLE,
 divid1,
 divid2,
 r,
 Met->Par[4].Val.V_INT,
 Met->Par[0].Val.V_LONG,
 Met->Par[1].Val.V_ENUM.value,
 Met->Par[3].Val.V_DOUBLE,
 &prix2);

Met->Res[0].Val.V_DOUBLE = prix;
Met->Res[1].Val.V_DOUBLE = (prix1 - prix) / (ptMod->S01.Val.V_PDOUBLE * Met->P
```

```
  Met->Res[2].Val.V_DOUBLE = (prix2 - prix) / (ptMod->S02.Val.V_PDOUBLE * Met->P

  return OK;
}

static int CHK_OPT(MC_JainOosterlee2D)(void *Opt, void *Mod)
{
  Option *ptOpt = (Option *)Opt;

  if (strcmp(ptOpt->Name, "CallMaximumAmer") == 0)
    return OK;

  return  WRONG;

}

static int MET(Init)(PricingMethod *Met, Option *Opt)
{
  if (Met->init == 0)
    {
      Met->init = 1;
      Met->HelpFilenameHint = "MC_JainOosterlee";
      Met->Par[0].Val.V_LONG = 50000;
      Met->Par[1].Val.V_ENUM.value = 0;
      Met->Par[1].Val.V_ENUM.members = &PremiaEnumMCRNGs;
      Met->Par[2].Val.V_PDOUBLE = 0.1;
      Met->Par[3].Val.V_INT = 6;
      Met->Par[4].Val.V_INT = 20;

    }
  return OK;
}


PricingMethod MET(MC_JainOosterlee2D) =
{
  "MC_JainOosterlee2D",
  { {"N iterations", LONG, {100}, ALLOW},
    {"RandomGenerator", ENUM, {100}, ALLOW},
    {"Delta Increment Rel", PDOUBLE, {100}, ALLOW},
    {"Dimension Approximation", INT, {100}, ALLOW},
```

```
    {"Number of Exercise Dates", INT, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_JainOosterlee2D),
  { {"Price", DOUBLE, {100}, FORBID},
    {"Delta1", DOUBLE, {100}, FORBID} ,
    {"Delta2", DOUBLE, {100}, FORBID},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(MC_JainOosterlee2D),
  CHK_mc,
  MET(Init)
};
```