Help

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


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

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2013+2) //The "#els
static int CHK_OPT(MC_JainOosterleeND)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(MC_JainOosterleeND)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else

/**
 * 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 */
  int dim;/*!< dimension de l'actif */
};
```

```
static double g(PnlVect *x)
{
  return pnl_vect_max(x);
}

/**
 * Computes the payoff of a call in dimension d
 *
 * @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(PnlVect *St, double t, const Product *P)
{
  return  MAX(g(St) - P->K, 0.);
}

/* calcule corr(X3,Y) où Y=max(X1,X2)
 * @param mu : moyenne de (X1,X2)
 * @param sigma : écart type de (X1,X2)
 * @param nu : moyenne et écart type de Y
 * @param corr1 : corr(X1,X3)
 * @param corr2 : corr(X2,X3)
 * @param corr : corr(X1,X2)
 *
 * output : corrélation entre X3 et Y
 */
static double corr_max(PnlVect *mu, PnlVect *sigma, PnlVect *nu, double corr1, d
{
  double mu0 = GET(mu, 0);
  double mu1 = GET(mu, 1);
  double sigma0 = GET(sigma, 0);
  double sigma1 = GET(sigma, 1);
  double sigma0_sq = sigma0 * sigma0;
  double sigma1_sq = sigma1 * 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);
  return (sigma0 * corr1 * PHI_alpha + sigma1 * corr2 * PHI_moins_alpha) / sqrt(
}

/**
 * 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 two non centered moments of max(X1,X2)
 */
static void deux_moments_max(PnlVect *nu, PnlVect *mu, PnlVect *sigma, double co
{
  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 sigma0_sq = sigma0 * sigma0;
  double sigma1_sq = sigma1 * 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);
  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
}

/**
 * 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 cumulants_max_2d(PnlVect *res, PnlVect *mu, PnlVect *sigma, double c
{
  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
  pnl_vect_free(&nu);
}

/* calcul des 4 premiers cumulants de Y=max(X1,...,Xd)
 * @param mu : vecteur des moyennes de X1,...,Xd
 * @param sigma : vecteur des écarts types de X1,...,Xd
 * @param P: product
 *
 * output : 4 premiers cumulants de max(X1,...,Xd)
 */
static void moments_max(PnlVect *moments, PnlVect *mu, PnlVect *sigma, const Pro
{
  int i, j;
  PnlVect *mu_2d, *mu_2d_tmp;
  PnlVect *sigma_2d, *sigma_2d_tmp;
  PnlVect *nu, *nu_tmp;
  PnlMat *mat_nu, *mat_corr;
  mat_nu = pnl_mat_create_from_double(P->dim - 1, 2, 0);
  //mat_corr matrice de taille dim*dim mat_corr(i,j)=corr(Y_(i+1),X_(j+1)); i<j
  mat_corr = pnl_mat_create_from_double(P->dim, P->dim, 0);
  nu = pnl_vect_create(2);
  mu_2d = pnl_vect_create(2);
  sigma_2d = pnl_vect_create(2);
  nu_tmp = pnl_vect_create(2);
  mu_2d_tmp = pnl_vect_create(2);
  sigma_2d_tmp = pnl_vect_create(2);
  //mat_nu matrice dim*2 dont chaque ligne i contient E[Y_(i+1)],sigma(Y_(i+1))
  MLET(mat_nu, 0, 0) = GET(mu, 0);
  MLET(mat_nu, 0, 1) = pow(GET(sigma, 0), 2) + pow(MGET(mat_nu, 0, 0), 2);
  for (i = 0; i < P->dim; i++) //corr(Y1,X_(j+1))=rho
    MLET(mat_corr, 0, i) = P->rho;
  for (i = 2; i < P->dim; i++)
    {
      LET(mu_2d, 0) = MGET(mat_nu, i - 2, 0);
      LET(mu_2d, 1) = GET(mu, i - 1);
      LET(sigma_2d, 0) = sqrt(MGET(mat_nu, i - 2, 1) - pow(MGET(mat_nu, i - 2, 0
      LET(sigma_2d, 1) = GET(sigma, i - 1);
      deux_moments_max(nu, mu_2d, sigma_2d, MGET(mat_corr, i - 2, i - 1));
      MLET(mat_nu, i - 1, 0) = GET(nu, 0);
```

```
      MLET(mat_nu, i - 1, 1) = GET(nu, 1);
      for (j = 1; j < i; j++)
        {
          LET(mu_2d_tmp, 0) = MGET(mat_nu, j - 1, 0);
          LET(mu_2d_tmp, 1) = GET(mu, j);
          LET(sigma_2d_tmp, 0) = sqrt(MGET(mat_nu, j - 1, 1) - pow(MGET(mat_nu,
          LET(sigma_2d_tmp, 1) = GET(sigma, j);
          LET(nu_tmp, 0) = MGET(mat_nu, j, 0);
          LET(nu_tmp, 1) = MGET(mat_nu, j, 1);
          MLET(mat_corr, j, i) = corr_max(mu_2d_tmp, sigma_2d_tmp, nu_tmp, MGET(
        }
    }

  LET(mu_2d, 0) = MGET(mat_nu, P->dim - 2, 0);
  LET(mu_2d, 1) = GET(mu, P->dim - 1);
  //printf("mu_2d=\ n");pnl_vect_print(mu_2d);
  LET(sigma_2d, 0) = sqrt(MGET(mat_nu, P->dim - 2, 1) - pow(MGET(mat_nu, P->dim
  LET(sigma_2d, 1) = GET(sigma, P->dim - 1);
  //printf("sigma_2d=\ n");pnl_vect_print(sigma_2d);
  //pnl_mat_print(mat_nu);
  cumulants_max_2d(moments, mu_2d, sigma_2d, MGET(mat_corr, P->dim - 2, P->dim -
  pnl_vect_free(&mu_2d);
  pnl_vect_free(&mu_2d_tmp);
  pnl_vect_free(&sigma_2d);
  pnl_vect_free(&sigma_2d_tmp);
  pnl_vect_free(&nu);
  pnl_vect_free(&nu_tmp);
  pnl_mat_free(&mat_nu);
  pnl_mat_free(&mat_corr);
}

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 matrix, each
 * entry of the array being an asset path. S[m] de taille (N+1)*dim
 * @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 *gSt;
  PnlVect *St;

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

/**
 * Construit la factorisée de Cholesky (triangulaire inférieure) de la
 * matrice de corrélation (avec des corr partour et une diagonale de 1
 *
 * @param correl (out) factorisée de Cholesky de la matrice de corrélation
 * @param d dimension du modèle
 * @param corr corrélation entre les composantes
 */

static void init_correl(PnlMat **correl, const Product *P)
{
```

```
  *correl = pnl_mat_create_from_double(P->dim, P->dim, P->rho);
  pnl_mat_set_diag(*correl, 1, 0.);
  pnl_mat_chol((*correl));
}

/**
 * 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)*d
 * 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, PnlMat *correl)
{
  double h, sqrt_h, s;
  PnlVect *drift;
  int k, j, i;
  drift = pnl_vect_create(P->dim);
  pnl_mat_set_row(S, P->spot, 0);

  h = P->T / P->N; /* step size between two exercise dates */
  sqrt_h = sqrt(h);
  for (j = 0; j < P->dim; j++)
    {
      LET(drift, j) = (P->r - GET(P->divid, j) - GET(P->sigma, j) * GET(P->sigma

    }

  for (k = 0 ; k < P->N ; k++)
    {
      for (i = 0; i < P->dim; i++)
        {
          s = 0.0;
          for (j = 0; j < i + 1; j++)
            {
              s = s + MGET(correl, i, j) * MGET(G, k, j);
            }
          MLET(S, k + 1, i) = MGET(S, k, i) * exp(GET(drift, i) + GET(P->sigma,
        }
    }
```

```
  pnl_vect_free(&drift);
}

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*/
  int i; /*indice dimension du sous jacent*/
  PnlBasis *Basis;
  double tn, h, s;
  PnlVect *Alpha, *prix, *cont_reg;
  // double sig_sq_2=pow(P->sigma,2)/2.0;

  PnlVect *vect_sigma;
  PnlVect *vect_mu;
  PnlVect *moments;
  PnlVect *ST, *ST_1, *St, *St_1, *r_div_sig_sq_2;
  moments = pnl_vect_create(4);
  vect_sigma = pnl_vect_create(P->dim);
  vect_mu = pnl_vect_create(P->dim);
  ST = pnl_vect_create(P->dim);
```

```
ST_1 = pnl_vect_create(P->dim);
St = pnl_vect_create(P->dim);
St_1 = pnl_vect_create(P->dim);
r_div_sig_sq_2 = pnl_vect_create(P->dim);
for (i = 0; i < P->dim; i++)
  {
    LET(r_div_sig_sq_2, i) = P->r - GET(P->divid, i) - GET(P->sigma, i) * GET(
  }
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 payoff;
    pnl_mat_get_row(ST, S[m], P->N);
    payoff = payoff_call(ST, P->T, P);
    LET(prix, m) = payoff;

  }

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

for (m = 0 ; m < P->MC ; m++)
  {
    pnl_mat_get_row(ST_1, S[m], P->N - 1);
    for (i = 0; i < P->dim; i++)
      {
        LET(vect_sigma, i) = GET(P->sigma, i) * sqrt(h);
        LET(vect_mu, i) = log(GET(ST_1, i)) + GET(r_div_sig_sq_2, i) * h;
      }
```

```
    moments_max(moments, vect_mu, vect_sigma, P);
    //printf("moments \ n");
    //pnl_vect_print(moments);
    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++)
      {
        double payoff;
        pnl_mat_get_row(St, S[m], n);
        payoff = payoff_call(St, 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++)
      {
        pnl_mat_get_row(St_1, S[m], n - 1);
        for (i = 0; i < P->dim; i++)
          {
            LET(vect_sigma, i) = GET(P->sigma, i) * sqrt(h);
```

```
            LET(vect_mu, i) = log(GET(St_1, i)) + GET(r_div_sig_sq_2, i) * h;
          }
        moments_max(moments, vect_mu, vect_sigma, P);

        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(P->spot, 0., P), GET(cont_reg, 0));

 pnl_basis_free(&Basis);
 pnl_vect_free(&cont_reg);
 pnl_vect_free(&ST);
 pnl_vect_free(&ST_1);
 pnl_vect_free(&St);
 pnl_vect_free(&St_1);
 pnl_vect_free(&Alpha);
 pnl_vect_free(&prix);
 pnl_vect_free(&vect_mu);
 pnl_vect_free(&vect_sigma);
 pnl_vect_free(&moments);
 pnl_vect_free(&r_div_sig_sq_2);
}


int MC_JainOosterleeND(PnlVect *spot, PnlVect *sig, double rho, double T, double
{
 int m;
 double prix, h;
 PnlRng *rng;
 PnlMat **G;
 PnlMat **S;
 PnlVectInt *tau;
 Product P;
```

```
double v;
PnlMat *correl;
PnlVect *S_tau;




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

P.N = N;//nb de dates de discrétisation

P.rho = rho;
P.degree = degree;//degré de la base de polynomes pour la régression
P.MC  = MC;//nb tirages Monte Carlo
P.K = K;
P.dim = spot->size;
h = P.T / P.N;


init_correl(&correl, &P);
S_tau = pnl_vect_create(P.dim);

P.spot = spot;//spot
P.sigma  = sig;//volatilité
P.divid  = divid;//dividende

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, P.dim);
```

```
      G[m] = pnl_mat_create(P.N, P.dim);
      pnl_mat_rng_normal(G[m], P.N, P.dim, rng);
      asset(S[m], &P, G[m], correl);
    }

  tau = pnl_vect_int_new();

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

  prix = 0.;
  for (m = 0 ; m < P.MC ; m++)
    {
      double payoff;
      const int tau_m = pnl_vect_int_get(tau, m);
      pnl_mat_get_row(S_tau, S[m], tau_m);
      payoff = payoff_call(S_tau, 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(&S_tau);
  pnl_mat_free(&correl);
  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_JainOosterleeND)(void *Opt, void *Mod, PricingMethod *Met)
{
  TYPEOPT *ptOpt = (TYPEOPT *)Opt;
  TYPEMOD *ptMod = (TYPEMOD *)Mod;
  double r;
```

```
  int i;
  PnlVect *divid = pnl_vect_create(ptMod->Size.Val.V_PINT);
  PnlVect *spot, *sig;

  spot = pnl_vect_compact_to_pnl_vect(ptMod->S0.Val.V_PNLVECTCOMPACT);
  sig = pnl_vect_compact_to_pnl_vect(ptMod->Sigma.Val.V_PNLVECTCOMPACT);
  for (i = 0; i < ptMod->Size.Val.V_PINT; i++)
    pnl_vect_set(divid, i, log(1. + pnl_vect_compact_get(ptMod->Divid.Val.V_PNLV
  r = log(1. + ptMod->R.Val.V_DOUBLE / 100.);

  MC_JainOosterleeND
  (spot,
   sig,
   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,
   divid,
   r,
   Met->Par[3].Val.V_INT,
   Met->Par[0].Val.V_LONG,
   Met->Par[1].Val.V_ENUM.value,
   Met->Par[2].Val.V_INT,
   &(Met->Res[0].Val.V_DOUBLE));

  pnl_vect_free(&divid);
  pnl_vect_free(&spot);
  pnl_vect_free(&sig);
  return OK;
}

static int CHK_OPT(MC_JainOosterleeND)(void *Opt, void *Mod)
{
  Option *ptOpt = (Option *)Opt;
  if (strcmp(ptOpt->Name, "CallMaximumAmer_nd") == 0) return OK;
  return  WRONG;
}
#endif //PremiaCurrentVersion

static int MET(Init)(PricingMethod *Met, Option *Opt)
{
  if (Met->init == 0)
```

```
    {
      Met->init = 1;
      Met->HelpFilenameHint = "MC_JainOosterlee_ND";
      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_INT = 3;
      Met->Par[3].Val.V_INT = 10;
    }
  return OK;
}

PricingMethod MET(MC_JainOosterleeND) =
{
  "MC_JainOosterleeND",
  { {"N iterations", LONG, {100}, ALLOW},
    {"RandomGenerator", ENUM, {0}, ALLOW},
    {"Approximation order", INT, {100}, ALLOW},
    {"Number of Exercise Dates", INT, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_JainOosterleeND),
  { {"Price", DOUBLE, {100}, FORBID},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(MC_JainOosterleeND),
  CHK_mc,
  MET(Init)
};
```