Help

```
#include  "lmm1d_cgmy_stdi.h"
#include "enums.h"
#include"pnl/pnl_vector.h"
#include"pnl/pnl_random.h"
#include"pnl/pnl_specfun.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2011+2) //The "#els
static int CHK_OPT(MC_LMM1d_CGMY_SWAPTION)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(MC_LMM1d_CGMY_SWAPTION)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else

//expected value of the jump size to power n
double cumulant(double C, double G, double M, double Y, int n)
{
  if (PNL_IS_EVEN(n) == 1)
    return C * pow(M, Y - n) * pnl_sf_gamma(n - Y) + C * pow(G, Y - n) * pnl_sf_
  else
    return C * pow(M, Y - n) * pnl_sf_gamma(n - Y) - C * pow(G, Y - n) * pnl_sf_
}

// dX_t = h_1(X_t) dt + h_2(X_{t-}) dZ_t
//Computing h1, h2 and their derivatives
void h2(double t, PnlVect *x, PnlVect *Sigma, PnlVect *h)
{
  int i;
  for (i = 0; i < x->size; i++)
    *pnl_vect_lget(h, i) = pnl_vect_get(x, i) * pnl_vect_get(Sigma, i);
}

void dh2(double t, PnlVect *x, PnlVect *Sigma, PnlVect *h, PnlMat *dh)
{
  int i, j;
  h2(t, x, Sigma, h);
```

```
  for (i = 0; i < dh->m; i++)
    {
      for (j = 0; j < dh->n; j++)
        {
          if (i != j)
            *pnl_mat_lget(dh, i, j) = 0;
          else
            *pnl_mat_lget(dh, i, i) = pnl_vect_get(Sigma, i);
        }
    }
}
void h1(double t, PnlVect *x, PnlVect *Sigma, double delta, double C, double G,
{
  int i, j, N;
  PnlVect *al, *coef, *cum;
  N = x->size;
  al = pnl_vect_create_from_double(N, 0.0);
  coef = pnl_vect_create_from_double(N, 0.0);
  cum = pnl_vect_create_from_double(N, 0.0);
  for (i = 1; i < N; i++)
    {
      *pnl_vect_lget(al, i) = delta * pnl_vect_get(x, i) * pnl_vect_get(Sigma, i
      *pnl_vect_lget(cum, i) = cumulant(C, G, M, Y, i + 1);
    }
  for (i = 0; i < N; i++)
    *pnl_vect_lget(h, i) = 0;
  *pnl_vect_lget(coef, N - 1) = 1;
  for (i = N - 2; i >= 0; i--)
    for (j = N - 1; j >= i + 1; j--)
      {
        *pnl_vect_lget(coef, j - 1) += pnl_vect_get(coef, j) * pnl_vect_get(al,
        *pnl_vect_lget(h, i) -= pnl_vect_get(Sigma, i) * pnl_vect_get(x, i) * pn
      }
  pnl_vect_free(&al);
  pnl_vect_free(&coef);
  pnl_vect_free(&cum);
}
void dh1(double t, PnlVect *x, PnlVect *Sigma, double delta, double C, double G,
{

  int i, j, k, N, shift;
```

```
PnlVect *al, *coef, *cum, *be;
N = x->size;
al = pnl_vect_create_from_double(N, 0.0);
coef = pnl_vect_create_from_double(N, 0.0);
cum = pnl_vect_create_from_double(N, 0.0);
be = pnl_vect_create_from_double(N, 0.0);
h1(t, x, Sigma, delta, C, G, M, Y, h);
for (i = 0; i < N; i++)
  {
    *pnl_vect_lget(cum, i) = cumulant(C, G, M, Y, i + 2);
    *pnl_vect_lget(al, i) = delta * pnl_vect_get(x, i) * pnl_vect_get(Sigma, i
    *pnl_vect_lget(be, i) = (delta * pnl_vect_get(Sigma, i) / (1 + delta * pnl
  }
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    *pnl_mat_lget(dh, i, j) = 0;
for (k = 1; k < N; k++)
  {
    for (i = 0; i < N - 1; i++)
      *pnl_vect_lget(coef, i) = 0;
    *pnl_vect_lget(coef, N - 1) = 1;
    shift = 0;
    for (i = N - 2; i >= 0; i--)
      {
        if (k > i)
          *pnl_mat_lget(dh, i, k) -= pnl_vect_get(cum, 0);
        if (k != i + 1)
          {
            for (j = N - 1; j >= i + 1 + shift; j--)
              {
                *pnl_vect_lget(coef, j - 1) += pnl_vect_get(coef, j) * pnl_vec
                if (k > i)
                  *pnl_mat_lget(dh, i, k) -= pnl_vect_get(coef, j) * pnl_vect_
              }
          }
        else
          shift = 1;
        *pnl_mat_lget(dh, i, k) *= (pnl_vect_get(x, i) * pnl_vect_get(Sigma, i
      }
  }
for (k = 0; k < N; k++)
```

```
    *pnl_mat_lget(dh, k, k) = pnl_vect_get(h, k) / pnl_vect_get(x, k);

  pnl_vect_free(&al);
  pnl_vect_free(&coef);
  pnl_vect_free(&cum);
  pnl_vect_free(&be);
}
void F(double t, PnlVect *X_Y0, PnlMat *X_Omega, PnlVect *Sigma, double delta, d
// This function will be called by the Runge-Kutta scheme algorithm
// We are simultaneously solving the system of equations
// dY_0(t)/dt = h_1(Y_0) + h_2(Y_0)gamma_eps
// dOmega/dt = Omega M + M^T Omega + N
// with M(t) = dh_1/dx(Y_0) + dh_2/dx(Y_0)gamma_eps
// N_t = h_2(Y_0)sigma2eps^2 h_2(Y_0)^T
{
  int i, j;
  PnlMat *Mat, *temp_Mat;
  PnlVect *V;
  Mat = pnl_mat_create(X_Y0->size, X_Y0->size);
  temp_Mat = pnl_mat_create(X_Y0->size, X_Y0->size);
  V = pnl_vect_create(X_Y0->size);
  dh1(t, X_Y0, Sigma, delta, C, G, M, Y, V, Mat);
  dh2(t, X_Y0, Sigma, Z_Y0, Z_Omega);
  pnl_mat_mult_double(Z_Omega, gammaeps);
  pnl_mat_plus_mat(Mat, Z_Omega);
  pnl_mat_clone(temp_Mat, Mat);
  pnl_mat_mult_mat_inplace(Mat, temp_Mat, X_Omega);
  pnl_mat_clone(Z_Omega, Mat);
  pnl_mat_sq_transpose(Mat);
  pnl_mat_plus_mat(Z_Omega, Mat);

  for (i = 0; i < X_Y0->size; i++)
    for (j = 0; j < X_Y0->size; j++)
      *pnl_mat_lget(Z_Omega, i, j) += pnl_vect_get(Z_Y0, i) * pnl_vect_get(Z_Y0,
  pnl_vect_mult_double(Z_Y0, gammaeps);
  pnl_vect_plus_vect(Z_Y0, V);

  pnl_vect_free(&V);
  pnl_mat_free(&Mat);
  pnl_mat_free(&temp_Mat);
}
```

```
//Runge-Kutta schema of order 4
static void RK4(double t0, double t, double maxstep, PnlVect *X_Y0, PnlMat *X_Om
// t0       : initial time
// t        : final time
// maxstep  : maximal time step
// X0       : initial value on entry, final on exit
{
  int i, nStep;
  double step;
  nStep = (int)((t - t0) / maxstep + 1);
  step = (t - t0) / nStep;
  pnl_vect_clone(temp2_Y0, X_Y0);
  pnl_mat_clone(temp2_Omega, X_Omega);
  for (i = 0; i < nStep; i++)
    {
      F(t0 + i * step, X_Y0, X_Omega, Sigma, delta, C, G, M, Y, gammaeps, sigma2
      pnl_vect_mult_double(temp1_Y0, step / 6);
      pnl_mat_mult_double(temp1_Omega, step / 6);
      pnl_vect_plus_vect(temp2_Y0, temp1_Y0);
      pnl_mat_plus_mat(temp2_Omega, temp1_Omega);
      pnl_vect_mult_double(temp1_Y0, 3.0);
      pnl_mat_mult_double(temp1_Omega, 3.0);
      pnl_vect_plus_vect(temp1_Y0, X_Y0);
      pnl_mat_plus_mat(temp1_Omega, X_Omega);
      F(t0 + i * step + step / 2, temp1_Y0, temp1_Omega, Sigma, delta, C, G, M,
      pnl_vect_mult_double(temp3_Y0, step / 3);
      pnl_mat_mult_double(temp3_Omega, step / 3);
      pnl_vect_plus_vect(temp2_Y0, temp3_Y0);
      pnl_mat_plus_mat(temp2_Omega, temp3_Omega);
      pnl_vect_mult_double(temp3_Y0, 1.5);
      pnl_mat_mult_double(temp3_Omega, 1.5);
      pnl_vect_plus_vect(temp3_Y0, X_Y0);
      pnl_mat_plus_mat(temp3_Omega, X_Omega);
      F(t0 + i * step + step / 2, temp3_Y0, temp3_Omega, Sigma, delta, C, G, M,
      pnl_vect_mult_double(temp1_Y0, step / 3);
      pnl_mat_mult_double(temp1_Omega, step / 3);
      pnl_vect_plus_vect(temp2_Y0, temp1_Y0);
      pnl_mat_plus_mat(temp2_Omega, temp1_Omega);
      pnl_vect_mult_double(temp1_Y0, 3.0);
      pnl_mat_mult_double(temp1_Omega, 3.0);
```

```
      pnl_vect_plus_vect(temp1_Y0, X_Y0);
      pnl_mat_plus_mat(temp1_Omega, X_Omega);
      F(t0 + (i + 1)*step, temp1_Y0, temp1_Omega, Sigma, delta, C, G, M, Y, gamm
      pnl_vect_mult_double(temp3_Y0, step / 6);
      pnl_mat_mult_double(temp3_Omega, step / 6);
      pnl_vect_plus_vect(temp2_Y0, temp3_Y0);
      pnl_mat_plus_mat(temp2_Omega, temp3_Omega);
      pnl_vect_clone(X_Y0, temp2_Y0);
      pnl_mat_clone(X_Omega, temp2_Omega);
    }
}


static void multi_rand_normal(PnlVect *W, PnlMat *Omega, int generator)
{
// Simulate a multivariate Gaussian random vector with mean zero and covariance
// Uses Singular Value decomposition (small negative eigenvalues are replaced wi
  PnlMat *P;
  PnlVect *V, *G;
  int i, N;
  N = W->size;
  V = pnl_vect_create(N);
  G = pnl_vect_create_from_double(N, 0.0);
  P = pnl_mat_create(N, N);

  pnl_mat_eigen(V, P, Omega, TRUE);

  for (i = 0; i < N; i++)
    {
      if (pnl_vect_get(V, i) > 0)
        *pnl_vect_lget(G, i) = pnl_rand_normal(generator) * sqrt(pnl_vect_get(V,
    }
  pnl_mat_mult_vect_inplace(W, P, G);

  pnl_vect_free(&G);
  pnl_vect_free(&V);
  pnl_mat_free(&P);
}
//Compute the positive or negative jump size between the smallest and the bigges
static double jump_generator_CGMY(double *cdf_jump_vect, double *cdf_jump_points
{
```

```
double z, v, y;
int test, temp, l, j, q;
test = 0;
v = pnl_rand_uni(generator);
y = cdf_jump_vect[cdf_jump_vect_size] * v;
l = cdf_jump_vect_size / 2;
j = cdf_jump_vect_size;
z = 0;
if (cdf_jump_vect[l] > y)
  {
    l = 0;
    j = cdf_jump_vect_size / 2;
  }
if (v == 1)
  {
    z = cdf_jump_points[cdf_jump_vect_size];
  }
if (v == 0)
  {
    z = cdf_jump_points[0];
  }
if (v != 1 && v != 0)
  {
    while (test == 0)
      {
        if (cdf_jump_vect[l + 1] > y)
          {
            q = l;
            test = 1;
          }
        else
          {
            temp = (j - l - 1) / 2 + l;
            if (cdf_jump_vect[temp] > y)
              {
                j = temp;
                l = l + 1;
              }
            else
              {
                l = temp * (temp > l) + (l + 1) * (temp <= l);
```

```
                }
            }
        }
      z = pow(1 / pow(cdf_jump_points[q], Y) - (y - cdf_jump_vect[q]) * Y * exp(
    }
  return z;
}
//payoff of receiver swaption
static double Payoff(int swaption_payer_receiver, double K, double delta, PnlVec
{
  double fact, sum, res;
  int i;
  fact = 1;
  sum = 0;
  for (i = 0; i < libor->size; i++)
    {
      fact /= (1 + delta * pnl_vect_get(libor, i));
      sum += fact;
    }
  res = K * delta * sum - (1 - fact);

  if (swaption_payer_receiver == 0) return MAX(res * notional, 0);
  else return MAX(-res * notional, 0);
}

static void Mc_ReceiverSwaption(int swaption_payer_receiver, double T, double fl
{
  double eps, sum_payoff, sum_payoffsquare, w1, w2, gammaeps, err, u, u0, z, sig
  double lambda_m, cdf_jump_bound, pas, var_payoff, payoff, factor, fact, min_M_
  double *cdf_jump_points, *cdf_jump_vect_p, *cdf_jump_vect_m, *jump_time_vect,
  int i, k, jump_number_p, jump_number_m, jump_number, m1, m2, cdf_jump_vect_siz
  PnlVect *Sigma, *W, *X_Y0, *Libor, *temp1_Y0, *temp2_Y0, *temp3_Y0;
  PnlMat *MatNull, *X_Omega, *temp1_Omega, *temp2_Omega, *temp3_Omega;
  MatNull = pnl_mat_create_from_double(n_libor, n_libor, 0.0);
  W = pnl_vect_create(n_libor);
  Sigma = pnl_vect_create_from_double(n_libor, sigma);
  X_Omega = pnl_mat_create(n_libor, n_libor);
  temp1_Omega = pnl_mat_create(n_libor, n_libor);
  X_Y0 = pnl_vect_create(n_libor);
  Libor = pnl_vect_create_from_double(n_libor, (exp(flat_yield * period) - 1.) /
  temp1_Y0 = pnl_vect_create(n_libor);
```

```
  temp2_Y0 = pnl_vect_create(n_libor);
  temp2_Omega = pnl_mat_create(n_libor, n_libor);
  temp3_Y0 = pnl_vect_create(n_libor);
  temp3_Omega = pnl_mat_create(n_libor, n_libor);
  maxstep = 1.;
  factor = exp(-flat_yield * T);
  for (i = 0; i < n_libor; i++)
    factor /= (1 + period * pnl_vect_get(Libor, i));
  control_expec = exp(-flat_yield * T);
  err = 1E-16;
  eps = 0.1;
  cdf_jump_vect_size = 100000;
  jump_number = 0;
  sum_payoff = 0;
  sum_payoffsquare = 0;
  sum_control = 0;
  sum_controlsquare = 0;
  sum_controlpayoff = 0;
  if (M <= 1 || G <= 0 || Y >= 2 || Y == 0)
    {
      printf("Function MC_ReceiverSwaption : invalid parameters\ n");
    }
  lambda_p = C * pow(M, Y) * pnl_sf_gamma_inc(-Y, eps * M); //positive jump inte
  while (lambda_p * T < 5)
    {
      eps = eps * 0.9;
      lambda_p = C * pow(M, Y) * pnl_sf_gamma_inc(-Y, eps * M);
    }
  lambda_m = C * pow(G, Y) * pnl_sf_gamma_inc(-Y, eps * G); //negative jump inte
  while (lambda_m * T < 5)
    {
      eps = eps * 0.9;
      lambda_m = C * pow(G, Y) * pnl_sf_gamma_inc(-Y, eps * G);
    }
  lambda_p = C * pow(M, Y) * pnl_sf_gamma_inc(-Y, eps * M);
////////////////////////////////////////////////////////
  cdf_jump_bound = 1;
  min_M_G = MIN(M, G);
  //Computation of the biggest jump that we tolerate
  while (C * exp(-min_M_G * cdf_jump_bound) / (min_M_G * pow(cdf_jump_bound, 1 +
    cdf_jump_bound++;
```

```
  pas = (cdf_jump_bound - eps) / cdf_jump_vect_size;
  cdf_jump_points = malloc((cdf_jump_vect_size + 1) * sizeof(double));
  cdf_jump_vect_p = malloc((cdf_jump_vect_size + 1) * sizeof(double));
  cdf_jump_vect_m = malloc((cdf_jump_vect_size + 1) * sizeof(double));
  cdf_jump_points[0] = eps;
  cdf_jump_vect_p[0] = 0;
  cdf_jump_vect_m[0] = 0;
  //computation of the cdf of the positive and negative jumps at some points
  for (i = 1; i <= cdf_jump_vect_size; i++)
    {
      cdf_jump_points[i] = i * pas + eps;
      cdf_jump_vect_p[i] = cdf_jump_vect_p[i - 1] + exp(-M * cdf_jump_points[i -
      cdf_jump_vect_m[i] = cdf_jump_vect_m[i - 1] + exp(-G * cdf_jump_points[i -
    }
////////////////////////////////////////////////////
  drift = -C * (pow(M, Y - 1) * pnl_sf_gamma_inc(1 - Y, M) - pow(G, Y - 1) * pnl
  sigma2eps = C * (pow(M, Y - 2) * (pnl_sf_gamma(2 - Y) - pnl_sf_gamma_inc(2 - Y
  gammaeps = drift - C * (pow(M, Y - 1) * (pnl_sf_gamma_inc(1 - Y, eps * M) - pn
////////////////////////////////////////////////////
  m1 = (int)(1000 * lambda_p * T);
  m2 = (int)(1000 * lambda_m * T);
  jump_time_vect_p = malloc((m1) * sizeof(double));
  jump_time_vect_m = malloc((m2) * sizeof(double));
  jump_time_vect_p[0] = 0;
  jump_time_vect_m[0] = 0;
  jump_time_vect = malloc((m1 + m2) * sizeof(double));
  jump_time_vect[0] = 0;
////////////////////////////////////////////////////
  pnl_rand_init(generator, 1, n_paths);
  for (i = 0; i < n_paths; i++)
    {
      pnl_vect_clone(X_Y0, Libor);
      //simulation of the positive jump times and number
      tau = -(1 / lambda_p) * log(pnl_rand_uni(generator));
      jump_number_p = 0;
      while (tau < T)
        {
          jump_number_p++;
          jump_time_vect_p[jump_number_p] = tau;
          tau += -1 / (lambda_p) * log(pnl_rand_uni(generator));
        }
```

```
//simulation of the negative jump times and number
tau = -(1 / lambda_m) * log(pnl_rand_uni(generator));
jump_number_m = 0;
while (tau < T)
  {
    jump_number_m++;
    jump_time_vect_m[jump_number_m] = tau;
    tau += -1 / (lambda_m) * log(pnl_rand_uni(generator));
  }
jump_time_vect_p[jump_number_p + 1] = T;
jump_time_vect_m[jump_number_m + 1] = T;
jump_number = jump_number_p + jump_number_m;
////////////////////////////////////////////////////////////
k1 = 1;
k2 = 1;
u0 = 0;
u = 0;
for (k = 1; k <= jump_number; k++)
  {
    w1 = jump_time_vect_p[k1];
    w2 = jump_time_vect_m[k2];
    if (w1 < w2)
      {
        u = w1;
        k1++;
        z = jump_generator_CGMY(cdf_jump_vect_p, cdf_jump_points, cdf_jump
      }
    else
      {
        u = w2;
        k2++;
        z = -jump_generator_CGMY(cdf_jump_vect_m, cdf_jump_points, cdf_jum
      }
    pnl_mat_clone(X_Omega, MatNull);
    jump_time_vect[k] = u;
    // Run the Runge-Kutta scheme
    RK4(u0, u, maxstep, X_Y0, X_Omega, temp1_Y0, temp1_Omega, temp2_Y0, te
    multi_rand_normal(W, X_Omega, generator); // Simulate a centered gauss
    pnl_vect_plus_vect(X_Y0, W); // add the 1st-order correction Y1 to the
    h2(u, X_Y0, Sigma, temp1_Y0); // compute the function h2 just before j
    pnl_vect_mult_double(temp1_Y0, z);
```

```
        pnl_vect_plus_vect(X_Y0, temp1_Y0); // Update X0
        u0 = u;
      }
    pnl_mat_clone(X_Omega, MatNull);
    jump_time_vect[jump_number + 1] = T;
    // Run the Runge-Kutta scheme
    RK4(u0, T, maxstep, X_Y0, X_Omega, temp1_Y0, temp1_Omega, temp2_Y0, temp2_
    multi_rand_normal(W, X_Omega, generator); // Simulate a centered gaussian
    pnl_vect_plus_vect(X_Y0, W); // add the 1st-order correction Y1 to the 0-o
    ////////////////////////////////////////////////////////
    //computation of the payoff
    fact = 1;
    for (k = 0; k < n_libor; k++)
      fact *= (1 + period * pnl_vect_get(X_Y0, k));
    payoff = factor * Payoff(swaption_payer_receiver, K, period, X_Y0, notiona
    control = factor * fact;
    sum_payoff += payoff;
    sum_payoffsquare += payoff * payoff;
    sum_control += control;
    sum_controlsquare += control * control;
    sum_controlpayoff += control * payoff;
  }
var_payoff = (sum_payoffsquare - sum_payoff * sum_payoff / ((double)n_paths))
cov_payoff_control = (sum_controlpayoff - sum_control * sum_payoff / ((double)
var_control = (sum_controlsquare - sum_control * sum_control / ((double)n_path
cor_payoff_control = cov_payoff_control / (sqrt(var_payoff) * sqrt(var_control
control_coef = cov_payoff_control / var_control;
*ptprice = sum_payoff / n_paths - control_coef * (sum_control / n_paths - cont
*priceerror = 1.96 * sqrt(var_payoff * (1 - cor_payoff_control * cor_payoff_co

pnl_vect_free(&Sigma);
pnl_vect_free(&W);
pnl_mat_free(&MatNull);
pnl_vect_free(&X_Y0);
pnl_vect_free(&Libor);
pnl_mat_free(&X_Omega);
pnl_vect_free(&temp1_Y0);
pnl_mat_free(&temp1_Omega);
pnl_mat_free(&temp2_Omega);
pnl_vect_free(&temp2_Y0);
pnl_mat_free(&temp3_Omega);
```

```
  pnl_vect_free(&temp3_Y0);
  free(cdf_jump_points);
  free(cdf_jump_vect_p);
  free(cdf_jump_vect_m);
  free(jump_time_vect_p);
  free(jump_time_vect_m);
  free(jump_time_vect);
}


static int mc_lmm1d_cgmy_swaption(NumFunc_1 *p, double l0, double sigma, double
{
  int swaption_payer_receiver = ((p->Compute) == &Call);
  int nbr_payments = (swap_maturity - swaption_maturity) / period;
  double flat_yield = log(period * l0 + 1) / period;

  Mc_ReceiverSwaption(swaption_payer_receiver, swaption_maturity, flat_yield, pe

  return OK;
}

int CALC(MC_LMM1d_CGMY_SWAPTION)(void *Opt, void *Mod, PricingMethod *Met)
{
  TYPEOPT *ptOpt = (TYPEOPT *)Opt;
  TYPEMOD *ptMod = (TYPEMOD *)Mod;

  return mc_lmm1d_cgmy_swaption(
           ptOpt->PayOff.Val.V_NUMFUNC_1,
           ptMod->l0.Val.V_PDOUBLE,
           ptMod->Sigma.Val.V_PDOUBLE,
           ptMod->C.Val.V_PDOUBLE,
           ptMod->G.Val.V_PDOUBLE,
           ptMod->M.Val.V_PDOUBLE,
           ptMod->Y.Val.V_PDOUBLE,
           ptOpt->BMaturity.Val.V_DATE - ptMod->T.Val.V_DATE,
           ptOpt->OMaturity.Val.V_DATE - ptMod->T.Val.V_DATE,
           ptOpt->Nominal.Val.V_PDOUBLE,
           ptOpt->FixedRate.Val.V_PDOUBLE,
           ptOpt->ResetPeriod.Val.V_DATE,
           Met->Par[0].Val.V_ENUM.value,
           Met->Par[1].Val.V_LONG,
```

```
            &(Met->Res[0].Val.V_DOUBLE),
            &(Met->Res[1].Val.V_DOUBLE));
}

static int CHK_OPT(MC_LMM1d_CGMY_SWAPTION)(void *Opt, void *Mod)
{
  if ((strcmp(((Option *)Opt)->Name, "PayerSwaption") == 0) || (strcmp(((Option
    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_ENUM.value = 0;
      Met->Par[0].Val.V_ENUM.members = &PremiaEnumRNGs;
      Met->Par[1].Val.V_LONG = 1000;
    }
  return OK;
}

PricingMethod MET(MC_LMM1d_CGMY_SWAPTION) =
{
  "MC_Lmm1d_CGMY_Swaption",
  {
    {"RandomGenerator", ENUM, {100}, ALLOW},
    {"N Simulation", LONG, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_LMM1d_CGMY_SWAPTION),
  {
    {"Price", DOUBLE, {100}, FORBID},
    {"Price Error", DOUBLE, {100}, FORBID},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
```

```
  CHK_OPT(MC_LMM1d_CGMY_SWAPTION),
  CHK_ok,
  MET(Init)
} ;
```