Help

```
extern "C" {
#include  "
href../../../mod/sabr1d/sabr1d_std/sabr1d_std_h_src.pdfsabr1d_std.h"

#include "
href../../../common/enums_h_src.pdfenums.h"
#include "
href../../../common/error_msg_h_src.pdferror_msg.h"
#include "pnl/pnl_mathtools.h"
#include "pnl/pnl_random.h"
#include "pnl/pnl_complex.h"
#include "pnl/pnl_matrix.h"
#include "pnl/pnl_cdf.h"
#include "pnl/pnl_specfun.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2019+2) //The "#els
static int CHK_OPT(MC_LeitaoGrzelakOosterlee)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(MC_LeitaoGrzelakOosterlee)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else

/*****************************************************************************
********************************************** mSABR CONSTANTS ************
*****************************************************************************
static int N = 50; // COS elements
static int M = 3000; // Monitoring dates
static int L = 150; // IM subintervals
static int S = 1000; // CDF inversion
static int NCP1 = 3; // SCMC points
static int NCP2 = NCP1; // SCMC points

/*****************************************************************************
************************************* AUXILIARY FUNCTIONS (mSABR) ************
*****************************************************************************
```

```
/*********************************************************
 * Auxiliary function for the function 'iterp1'. Finds the
 * nearest index of the evaluated interpolation point.
 *********************************************************/
static int findNearestNeighbourIndex(double value, double *x, int len){

double dist, newDist;
int idx;

idx = -1;
dist = DBL_MAX;
for (int i = 0; i < len; i++) {
newDist = value - x[i];
if (newDist > 0 && newDist < dist){
dist = newDist;
idx = i;
}
}

return idx;
}

/*********************************************************
 * Construcs the linear interpolator.
 * Evaluates the interpolator in the given values.
 *********************************************************/
static void interp1(double *x, int x_tam, double *y, double *xx, int xx_tam, dou

double dx, dy;
int i, indiceEnVector;

double *slope = (double *)calloc(x_tam,sizeof(double));
double *intercept=(double *)calloc(x_tam,sizeof(double));

for(i = 0; i < x_tam; i++){
if(i < x_tam-1){
dx = x[i + 1] - x[i];
dy = y[i + 1] - y[i];
slope[i] = dy / dx;
intercept[i] = y[i] - x[i] * slope[i];
```

```
}else{
slope[i] = slope[i - 1];
intercept[i] = intercept[i - 1];
}
}

for (i = 0; i < xx_tam; i++) {
indiceEnVector = findNearestNeighbourIndex(xx[i], x, x_tam);
if (indiceEnVector != -1)
yy[i] = slope[indiceEnVector] * xx[i] + intercept[indiceEnVector];
else
yy[i] = DBL_MAX;
}
free(slope);
free(intercept);
}

/*********************************************************
 * Generates two vectors of uniforms following a 2D
 * Gaussian Copula with correlation coefficien 'Rho'.
 *********************************************************/
static int copularand(double Rho, PnlRng *crng, int n, double *U1, double *U2){

double Z1, Z2, X1, X2;
for(int i = 0; i < n; i++){
Z1 = pnl_rng_normal(crng);
Z2 = pnl_rng_normal(crng);
X1 = Z1;
X2 = Rho*Z1 + sqrt(1.0 - Rho)*Z2;

U1[i] = pnl_cdfnor(X1);
U2[i] = pnl_cdfnor(X2);
}

return 0;
}

/*********************************************************
 * Evaluates the normal inverse CDF with mean 'm' and
 * standard deviation 'sd'. Based on the PNL analogous.
 *********************************************************/
```

```
static double my_norminv(double p, double m, double sd){

int which = 2;
double q = 1 - p;
double x;
int status;
double bound;
pnl_cdf_nor(&which, &p, &q, &x, &m, &sd, &status, &bound);

return x;
}

/*********************************************************
 * Computes the double factorial.
 *********************************************************/
static int doublefactorial(int n){

if (n <= 1)
return 1;

//  return n*doublefactorial(n - 2);

int dfact = n;
for(int i = 1; i < n/2; i++)
dfact*= (n - 2*i);
return dfact;
}

/*********************************************************
 * Auxiliary function for 'lagrange_interp_2d'.
 * Downloaded from:
 * https://people.sc.fsu.edu/~jburkardt/c_src/lagrange_interp_2d/lagrange_interp
 *********************************************************/
static double lagrange_basis_function_1d(int mx, double xd[], int i, double xi){
int j;
double yi = 1.0;
if (xi != xd[i])
for (j = 0; j < mx + 1; j++)
if (j != i)
yi = yi*(xi - xd[j])/(xd[i] - xd[j]);
```

```
return yi;
}

/*****************************************************
 * Construcs the 2D Lagrange interpolator.
 * Evaluates the interpolator in the given values.
 * Downloaded from:
 * https://people.sc.fsu.edu/~jburkardt/c_src/lagrange_interp_2d/lagrange_interp
 *****************************************************/
void lagrange_interp_2d(int mx, int my, double *xd_1d, double *yd_1d, double *zd
int i, j, k, l;
double lx, ly;
//  double *zi = (double *) malloc (ni*sizeof(double));
for (k = 0; k < ni; k++){
l = 0;
zi[k] = 0.0;
for (j = 0; j < my + 1; j++){
for (i = 0; i < mx + 1; i++){
lx = lagrange_basis_function_1d ( mx, xd_1d, i, xi[k] );
ly = lagrange_basis_function_1d ( my, yd_1d, j, yi[k] );
zi[k] = zi[k] + zd[l] * lx * ly;
l = l + 1;
}
}
}
//  return zi;
}

/*****************************************************
 * Transposes a matrix stored row-wise in an 1D array.
 *****************************************************/
static void column_wise_2DArray(double *M_array, int rows, int cols){

int i, j;
double *M_aux = (double *)malloc(rows*cols*sizeof(double));
for(i = 0; i < rows; i++)
for(j = 0; j < cols; j++)
M_aux[i*cols + j] = M_array[i*cols + j];
for(i = 0; i < rows; i++)
for(j = 0; j < cols; j++)
M_array[i*rows + j] = M_aux[j*cols + i];
```

```
free(M_aux);
}


/*****************************************************************************
************************************* AUXILIARY FUNCTIONS (LOW-BIAS) ********
*****************************************************************************/

/*********************************************************
 * Evaluation of the Non-central Chi-square CDF by
 * Sankaran's approximation.
 *********************************************************/
static double SanProx(double u, double x0, double k, double lambda){

double x, h, p, m;

x = pnl_inv_cdfnor(u);
h = 1.0 - 2.0*(k+lambda)*(k + 3.0*lambda)/pow((k + 2.0*lambda), 2)/3.0;
p = (k + 2.0*lambda)/pow((k + lambda), 2);
m = (h - 1.0)*(1.0 - 3.0*h);

return ( 1.0 - h*p*(1 - h + 0.5*(2.0 - h)*m*p) - pow((x0/(k + lambda)), h) )/(h*
}

/*********************************************************
 * Partial derivative of the Sankaran's approximation of
 * the non-central Chi-square CDF.
 *********************************************************/
static double partialDSan(double x, double k, double v){

double A, B;
A = (-1/pow((pow(k,2)+4*k*v+6*pow(v,2)),2)*12*sqrt(6.0)*k*v*pow((k+v),4)*(k+2*v)

B = (2*pow((k+v),7)*pow((k+2*v),2)*sqrt(pow(((3*pow(k,4)+21*pow(k,3)*v+54*pow(k,

return A/B;
}

/*********************************************************
 * Iterative computation of the non-central Chi-square CDF.
 *********************************************************/
```

```
double ncx2cdfIterative(double x, double delta, double lambda){

double z, v, kappa, Rnew, gA, gB, Sg, h;
int n;
double crit = pow(2.0,-40);
double R = 0.0;

z = x/2.0;
v = delta/2.0;
kappa = lambda/2.0;
if(z > 0){

gA = exp(-z)*pow(z,v)/tgamma(1.0 + v);
gB = exp(-kappa);
Sg = gB;
R = gA*Sg;
n = 1;

while (1){
n = n + 1;
gA = gA*z/(n + v - 1.0);
gB = gB*kappa/(n - 1.0);
Sg = Sg + gB;
Rnew = R + gA*Sg;
h  = R - Rnew;
if (fabs(h)<=crit)
 break;
R = Rnew;
}
}

return R;
}

/*********************************************************
 * Direct SABR CDF inversion by means of Newton's method,
 * using Sankaran's approximation.
 *********************************************************/
static double ncx2parav7_sankaran(double p, double X, double delta){

double crit, dim, pS, LL, Lnew, Prox, newProx, h;
```

```
int count, count_limit;

crit = sqrt(pow(2.0,-40));
dim = 2.0 - delta;

count_limit = 2;
count = 0;

LL = X;
Prox = SanProx(p, X, dim, LL);
while(count < count_limit){

count = count + 1;

pS = partialDSan(X, dim, LL);
h = Prox/pS;

Lnew = MAX(LL/5.0, MIN(5.0*LL, LL - h));

newProx = SanProx(p, X, dim, Lnew);

while(1){

if(fabs(newProx) <= fabs(Prox)*(1.0 + crit) || (fabs(LL-Lnew) <= crit*LL))
break;
Lnew = 0.5*(Lnew + LL);
newProx = SanProx(p, X, dim, Lnew);
     }
h = LL - Lnew;
LL = Lnew;
if ((fabs(h) <= crit*fabs(LL)) || (fabs(h) <= crit))
break;

Prox = newProx;
}

return LL;
}

/*********************************************************
 * Direct SABR CDF inversion by means of Newton's method,
```

```
 * using the actual non-central Chi-square CDF formula.
 ****************************************************/
static double ncx2parav7_exact(double p, double X, double delta, double Lsan){

double crit, dim, pF, LL, Lnew, F, newF, h;
int count, count_limit;

crit = sqrt(pow(2.0,-40));
dim = 2.0 - delta;

LL = Lsan;
if(LL < 40)
F = 1.0 - ncx2cdfIterative(X, dim, LL) - p;
else
F = 1.0 - pnl_cdfchi2n(X, dim, LL) - p;

count = 0;
count_limit = 100;

while(count < count_limit){

count = count + 1;
pF = 0.5*powf((LL/X),(0.25*(delta - 2)))*exp(-(X + LL)/2.0)*pnl_bessel_i(fabs(0.
h = F/pF;

Lnew = MAX(LL/5.0, MIN(5.0*LL, LL - h));

if(Lnew<40)
newF = 1.0 - ncx2cdfIterative(X, dim, Lnew) - p;
else
newF = 1.0 - pnl_cdfchi2n(X, dim, Lnew) - p;

while(1){

if(fabs(newF) <= fabs(F)*(1.0 + crit) || (fabs(LL - Lnew) <= crit*LL))
break;
Lnew = 0.5*(Lnew + LL);
newF = 1.0 - pnl_cdfchi2n(X, dim, Lnew) - p;
}

h = LL - Lnew;
```

```
LL = Lnew;
if ((fabs(h) <= crit*fabs(LL)) || (fabs(h) <= crit))
break;

F = newF;
}

return LL;
}

/*******************************************************
 * Computes the inversion of the SABR CDF,
 * combining direct inversion (by Newton's method)
 * and moment-matching approaches.
 *******************************************************/
static double ncx2parav7(double p, double X, double delta){

double pAb, m1, s2, psi, psi_inv, Lsan, b2, a, Z;
int grens;

pAb = 1.0 - pnl_cdfchi2n(X, 2.0 - delta, 0.0);
m1 = delta + X;
s2 = 2.0*(delta + 2.0*X);
psi = s2/pow(m1,2);
psi_inv = 1.0/psi;
grens = 2;

Lsan = 0.0;

if ((psi > grens && p > pAb) || (m1 < 0 && psi > 0 && psi <= grens && p > pAb)){
Lsan = ncx2parav7_sankaran(p, X, delta);
return ncx2parav7_exact(p, X, delta, Lsan);
}else{
b2 = 2.0*psi_inv - 1.0 + sqrt(2.0*psi_inv)*sqrt(2*psi_inv - 1.0);
a = m1/(1.0 + b2);
Z = pnl_inv_cdfnor(p);
return a*pow((sqrt(b2) + Z),2);
}

}
```

```
/*****************************************************************************
************************************************ mSABR FUNCTIONS ************
*****************************************************************************/

/*******************************************************
 * Computes the support of the distribution of process Y_M.
 *******************************************************/
static void support_YM(double alpha, double dt, double *a, double *b){

int H = 10;
double alpha2 = SQR(alpha);
double c1 = -0.5*alpha2*dt;
double c2 = alpha2*dt;
*a = c1 - H*sqrt(c2*M);
*b = log(M * 1.0) + M*c1 + H*sqrt(M*c2);
}

/*******************************************************
 * Evaluates the sub-integral I_L.
 *******************************************************/
static dcomplex integral_IL(double uk, double ul, double c2, double a, double aj

return CRdiv( Cadd( Cmul( Cexp(CRmul(CI, aj*c2*uk)), RCsub(ul*sin((a - aj)*ul),
}

/*******************************************************
 * Auxiliary function for 'integral_IM'.
 *******************************************************/
static double g(double x){

return log(exp(x) + 1);
}

/*******************************************************
 * Evaluates the integral I_M.
 *******************************************************/
static dcomplex integral_IM(double uk, double ul, double a, double b){

double aj, bj, c1, c2, pg_j, pg_j1, g_prime;

double s = 3.0;
```

```
double Pg_a = 1.0/(1.0 + exp(-a/s));
double Pg_b = 1.0/(1.0 + exp(-b/s));
double h_Pg = (Pg_b - Pg_a)/(L - 1);

dcomplex sum_IM = CZERO;
for(int j = 0; j < L-1; j++){

pg_j = Pg_a + j*h_Pg;
pg_j1 = Pg_a + (j + 1)*h_Pg;
aj = s*log(pg_j/(1.0 - pg_j));
bj = s*log(pg_j1/(1.0 - pg_j1));
g_prime = (g(bj) - g(aj))/(bj - aj);
c1 = g(aj) - aj*g_prime;
c2 = g_prime;
sum_IM = Cadd(sum_IM, Cmul( Cexp(CRmul(CI, uk*c1)), integral_IL(uk, ul, c2, a, a
}

return sum_IM;
}

/*********************************************************
 * Computes the matrix M.
 *********************************************************/
static int compute_M(double *u, double a, double b, PnlMatComplex *MM){

for(int k = 0; k < N-1; k++)
for(int l = 0; l < N-1; l++)
pnl_mat_complex_set(MM, k, l, integral_IM(u[k], u[l], a, b));

pnl_mat_complex_set(MM, 0, 0, Complex(b - a, 0.0));

return 0;
}

/*********************************************************
 * Evaluates the characteristic function of the log-return
 * proccess 'R'.
 *********************************************************/
static dcomplex phi_R(double u, double alpha, double dt){

return Cexp( CRsub(CRmul(CI, -u*SQR(alpha)*dt), 2.0*SQR(u*alpha)*dt) );
```

```
}

/*****************************************************
 * Constructs and evaluates the characteristic function of
 * the process 'Y_M'.
 *****************************************************/
static int Phi_YM(double *u, double alpha, double dt, double a, double b, dcompl

int l;

PnlMatComplex *MM = pnl_mat_complex_create_from_zero(N, N);
compute_M(u, a, b, MM);

PnlVectComplex *phiR = pnl_vect_complex_create(N);
for(l = 0; l < N; l++)
pnl_vect_complex_set(phiR, l, phi_R(u[l], alpha, dt));

PnlVectComplex *phiYM = pnl_vect_complex_copy(phiR);

PnlVectComplex *exp_ul = pnl_vect_complex_create(N);
for(l = 0; l < N; l++)
pnl_vect_complex_set(exp_ul, l, Cexp(CRmul(CI, -a*u[l])));

PnlVectComplex *Aj = pnl_vect_complex_create(N);
for(int j = 1; j < M; j++){

pnl_vect_complex_mult_vect_term(phiYM, exp_ul);
for(l = 0; l < N; l++){
pnl_vect_complex_set_real(Aj, l, GET_REAL(phiYM, l));
pnl_vect_complex_set_imag(Aj, l, 0.0);
}

pnl_vect_complex_mult_double(Aj, 2.0/(b - a));
pnl_vect_complex_set(Aj, 0, RCmul(0.5, GET_COMPLEX(Aj, 0)));

phiYM = pnl_mat_complex_mult_vect(MM, Aj);

pnl_vect_complex_mult_vect_term(phiYM, phiR);

}
pnl_vect_complex_free(&Aj);
```

```
pnl_vect_complex_free(&exp_ul);

for(int k = 0; k < N; k++)
phi_YM[k] = GET_COMPLEX(phiYM, k);

pnl_vect_complex_free(&phiYM);
pnl_vect_complex_free(&phiR);
pnl_mat_complex_free(&MM);

return 0;
}

/*********************************************************
 * Evaluates the characteristic function of
 * the log-transformation of process 'Y'.
 *********************************************************/
static int Phi_log_Y(double *u, double sigma0, double dt, dcomplex *phi_YM, dcom

for(int k = 0; k < N; k++)
phi_logY[k] = Cmul( Cexp(CRmul(CI, u[k]*log(dt*SQR(sigma0)))), phi_YM[k] );

return 0;
}

/*********************************************************
 * Constructs and evaluates, employing COS method,
 * the CDF of the log-transformation of process 'Y'.
 *********************************************************/
static int F_log_Y(double sigma0, double dt, dcomplex *phiYM, double *u, double

int i;

dcomplex *phi_logY = (dcomplex *)malloc(N*sizeof(dcomplex));
Phi_log_Y(u, sigma0, dt, phiYM, phi_logY);

double ahat = log(SQR(sigma0)*dt) + a;
double bhat = log(SQR(sigma0)*dt) + b;
double h_y = (bhat - ahat)/(S - 1);
for(i = 0; i < S; i++)
y[i] = ahat + i*h_y;
```

```
double * C = (double *)malloc(N*sizeof(double));
for(int k = 0; k < N; k++)
C[k] = Creal(Cmul(phi_logY[k], Cexp(CRmul(CI, -ahat*u[k]))));

double yi, Ck, sumFlogY;
for(i = 0; i < S; i++){
yi = y[i];
Ck = (1.0/(bhat - ahat))*C[0];
sumFlogY = Ck*(yi - ahat);
for(int k = 1; k < N; k++){
Ck = (2.0/(bhat - ahat))*C[k];
sumFlogY += ((bhat - ahat)/M_PI)*Ck*sin((yi - ahat)*u[k])/k;
}
FlogY[i] = sumFlogY;
}

free(C);
free(phi_logY);
return 0;
}

/*********************************************************
 * Determines the position of the position
 * of the collocation points.
 *********************************************************/
static int SC_collocation_points(int NCP, PnlMat *MCP, PnlVect *x){

pnl_mat_chol(MCP);
pnl_mat_sq_transpose(MCP);

PnlVect *alpha = pnl_vect_create(NCP);
PnlVect *beta = pnl_vect_create(NCP-1);
PnlMat *Jhat = pnl_mat_create_from_zero(NCP, NCP);

pnl_vect_set(alpha, 0, MGET(MCP, 0, 1)/MGET(MCP, 0, 0));
pnl_vect_set(beta, 0, SQR(MGET(MCP, 1, 1)/MGET(MCP, 0, 0)));
pnl_mat_set(Jhat, 0, 0, GET(alpha, 0));
pnl_mat_set(Jhat, 0, 1, sqrt(GET(beta, 0)));

for(int i = 1; i < NCP-1; i++){
pnl_vect_set(alpha, i, MGET(MCP, i, i+1)/MGET(MCP, i, i) - MGET(MCP, i-1, i)/MGE
```

```
pnl_vect_set(beta, i, SQR(MGET(MCP, i+1, i+1)/MGET(MCP, i, i)));

pnl_mat_set(Jhat, i, i-1, sqrt(GET(beta, i-1)));
pnl_mat_set(Jhat, i, i, GET(alpha, i));
pnl_mat_set(Jhat, i, i+1, sqrt(GET(beta, i)));
}

pnl_vect_set(alpha, NCP-1, MGET(MCP, NCP-1, NCP)/MGET(MCP, NCP-1, NCP-1) - MGET(
pnl_mat_set(Jhat, NCP-1, NCP-1, GET(alpha, NCP-1));
pnl_mat_set(Jhat, NCP-1, NCP-2, sqrt(GET(beta, NCP-2)));

pnl_mat_eigen(x, NULL, Jhat, FALSE);

pnl_vect_free(&alpha);
pnl_vect_free(&beta);
pnl_mat_free(&Jhat);

return 0;
}

/*********************************************************
 * Determines the optimal collocation point, using the
 * normal distribution as a cheap variable.
 *********************************************************/
static int SC_optimal_points_normal(int NCP, double mu, double sigma, double *op

int i;

PnlMat *MCP = pnl_mat_create_from_zero(NCP+1, NCP+1);
for(i = 0; i <= NCP; i++)
for(int j = 0; j <= NCP; j++)
if((i+j)%2 == 0)
pnl_mat_set(MCP, i, j, pow(sigma, i + j)*doublefactorial(i + j - 1));

PnlVect *vop = pnl_vect_create(NCP);
SC_collocation_points(NCP, MCP, vop);
pnl_vect_qsort (vop, 'i');
for(i = 0; i < NCP; i++)
op[i] = mu + GET(vop, i);
pnl_vect_free(&vop);
pnl_mat_free(&MCP);
```

```
return 0;
}

/*******************************************************
 * Generates samples of the conditional integrated variance
 * and the conditional volatility processes.
 *******************************************************/
static int mSABR_Integrated_Variance(double sigma0, double alpha, int t, double

int i;

double T1, T2, logRho;
T1 = t*dT;
T2 = (t+1)*dT;
logRho = 0.5*(SQR(T2) - SQR(T1))/sqrt(SQR(SQR(T2))/3.0 + 2.0*T2*CUB(T1)/3.0 - SQ

double *U1 = (double *)malloc(N_mc*sizeof(double));
double *U2 = (double *)malloc(N_mc*sizeof(double));
copularand(logRho, rng, N_mc, U1, U2);

double mulogsigma, slogsigma;
double *FlogY = (double *)malloc(NCP1*S*sizeof(double));
double *y = (double *)malloc(NCP1*S*sizeof(double));
double *logY = (double *)malloc(N_mc*sizeof(double));
if(t == 0){

F_log_Y(sigma0, dt, phiYM, u, a, b, FlogY, y);
interp1(FlogY, S, y, U2, N_mc, logY);

mulogsigma = log(sigma0) - 0.5*SQR(alpha)*dT;
slogsigma = alpha*sqrt(dT);

for(i = 0; i < N_mc; i++){
sigma[i] = exp(my_norminv(U1[i], mulogsigma, slogsigma));
Y[i] = exp(logY[i]);
}

}else{

double *sx = (double *)malloc(NCP1*sizeof(double));
mulogsigma = log(sigma0) - 0.5*SQR(alpha)*T1;
```

```
slogsigma = alpha*sqrt(T1);
SC_optimal_points_normal(NCP1, mulogsigma, slogsigma, sx);

double *yx = (double *)malloc(NCP2*sizeof(double));
SC_optimal_points_normal(NCP2, 0.0, 1.0, yx);
double *Fyx = (double *)malloc(NCP2*sizeof(double));
for(i = 0; i < NCP2; i++)
Fyx[i] = pnl_cdfnor(yx[i]);

double *logY_SC = (double *)malloc(NCP1*NCP2*sizeof(double));
for(i = 0; i < NCP1; i++){
F_log_Y(exp(sx[i]), dt, phiYM, u, a, b, &FlogY[i*S], &y[i*S]);
interp1(&FlogY[i*S], S, &y[i*S], Fyx, NCP2, &logY_SC[i*NCP2]);
}

for(i = 0; i < N_mc; i++){
sigma[i] = log(sigma[i]);
U2[i] = pnl_inv_cdfnor(U2[i]);
}

column_wise_2DArray(logY_SC, NCP1, NCP2);
lagrange_interp_2d(NCP1-1, NCP2-1, sx, yx, logY_SC, N_mc, sigma, U2, logY);

for(i = 0; i < N_mc; i++){
mulogsigma = sigma[i] - 0.5*SQR(alpha)*(T2 - T1);
slogsigma = alpha*sqrt(T2 - T1);
sigma[i] = exp(my_norminv(U1[i], mulogsigma, slogsigma));
Y[i] = exp(logY[i]);
}

free(logY_SC);
free(yx);
free(Fyx);
free(sx);
}
free(logY);
free(y);
free(FlogY);

free(U2);
free(U1);
```

```
return 0;
}

/*********************************************************
 * Generates samples of the underlying price, emplying
 * the Log-Euler+ scheme, conditional on  the volatility
 * and the integrated variance simulation.
 *********************************************************/
// static int LogEulerplus(double beta, double alpha, double rho, double *sigma,

//  double dW, dZ, nu;
//  for(int i = 0; i < N_mc; i++){
//  dZ = pnl_rng_normal(rng);
//  dW = sqrt(1.0 - SQR(rho))*sqrt(Y[i])*dZ;
//  nu = pow(F[i], beta - 1.0);
//  F[i] = MAX(F[i]*exp(-0.5*Y[i]*SQR(nu) + nu*(rho/alpha)*(sigma[i] - sigma_prv
//  *sumF += F[i];
//  }

//  return 0;
// }

/*********************************************************
 * Generates samples of the underlying price, emplying
 * the Low-bias scheme, conditional on  the volatility
 * and the integrated variance simulation.
 *********************************************************/
static int LowBias(double beta, double alpha, double rho, double *sigma, double

int i;

double x0, u, x;
double delta = (1.0 - 2.0*beta - rho*rho*(1.0 - beta))/((1.0 - beta)*(1.0 - rho*
for(i = 0; i < N_mc; i++){
if(F[i] > 0.0){
x0 = pow((pow(F[i], (1.0 - beta))/(1.0 - beta) + (rho/alpha)*(sigma[i] - sigma_p
u = pnl_rng_uni(rng);
x = ncx2parav7(u, x0, delta);

F[i] = pow(((1.0 - rho*rho)*Y[i]*x*pow((1.0 - beta), 2.0)), (1.0/(2.0 - 2.0*beta
```

```
if (pnl_isnan(F[i])) F[i] = 0.0;
*sumF += F[i];
}
}

return 0;
}

int mSABR_underlying(double S0, double beta, double alpha, double rho, double *s

int i;
double sumF = 0.0;
if (beta == 1.0){
double dZ;
for(i = 0; i < N_mc; i++){
dZ = pnl_rng_normal(rng);
F[i] = F[i]*exp(-0.5*Y[i] + (rho/alpha)*(sigma[i] - sigma_prv[i]) + sqrt((1.0 -
sumF += F[i];
}
}else if (beta == 0.0){
double a, pAb, U, dZ;
for(i = 0; i < N_mc; i++){
a = SQR(F[i] + (rho/alpha)*(sigma - sigma_prv))/((1.0 - SQR(rho))*Y[i]);
pAb = 1.0 - pnl_cdfchi2n(a, 1.0, 0.0);
U = pnl_rng_uni(rng);
if (F[i] <= 0.0 || pAb > U){
F[i] = 0.0;
}else{
dZ = pnl_rng_normal(rng);
F[i] = MAX(F[i] + (rho/alpha)*(sigma[i] - sigma_prv[i]) + sqrt((1.0 - SQR(rho))*
sumF += F[i];
}
}
}else
//  LogEulerplus(beta, alpha, rho, sigma, sigma_prv, Y, rng, N_mc, F, &sumF);
LowBias(beta, alpha, rho, sigma, sigma_prv, Y, rng, N_mc, F, &sumF);

// Martingale correction
//for(i = 0; i < N_mc; i++)
//if (F[i] > 0.0) F[i] += S0 - sumF/N_mc;
```

```
return 0;
}

int MCLeitaoGrzelakOosterlee(double S0, NumFunc_1  *p, double T,double sigma0,do
{
  double K;
  int flag_call;
  int i;
  double dT = T/N_time;
  double dt = dT/M;

  K = p->Par[0].Val.V_PDOUBLE;
  if ((p->Compute) == &Call)
    flag_call = 1;
  else
    flag_call = 0;

  // Characteristic function of the process Y_M
  double a, b;
  support_YM(alpha, dt, &a, &b);

  double *u = (double *)malloc(N*sizeof(double));
  for(int k = 0; k < N; k++)
u[k] = k*M_PI/(b - a);

  dcomplex *phiYM = (dcomplex *)malloc(N*sizeof(dcomplex));
  Phi_YM(u, alpha, dt, a, b, phiYM);

  // Random number generator
  PnlRng *rng = pnl_rng_create(gen);
  pnl_rng_sseed(rng, 0);

  // Initialization
  double *FT = (double *)malloc(N_mc*sizeof(double));
  double *sigma = (double *)malloc(N_mc*sizeof(double));
for(i = 0; i < N_mc; i++){
FT[i] = S0;
sigma[i] = sigma0;
}
double *Y = (double *)malloc(N_mc*sizeof(double));
double *sigma_prv = (double *)malloc(N_mc*sizeof(double));
```

```
for(int t = 0; t < N_time; t++){

for(i = 0; i < N_mc; i++)
sigma_prv[i] = sigma[i];

mSABR_Integrated_Variance(sigma0, alpha, t, dT, dt, phiYM, u, a, b, rng, N_mc, s

mSABR_underlying(S0, beta, alpha, rho, sigma, sigma_prv, Y, rng, N_mc, FT);
}
free(sigma_prv);
free(Y);
free(sigma);

pnl_rng_free(&rng);
free(phiYM);
free(u);

double sum_price = 0.0;
if(flag_call == 1)
for(i = 0; i < N_mc; i++)
sum_price += MAX(FT[i] - K, 0.0);
else
for(i = 0; i < N_mc; i++)
sum_price += MAX(K - FT[i], 0.0);

*ptprice = sum_price/N_mc;
free(FT);

  return OK;
}


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

      return MCLeitaoGrzelakOosterlee(ptMod->S0.Val.V_PDOUBLE,
                                ptOpt->PayOff.Val.V_NUMFUNC_1,
                                ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
                                ,ptMod->Beta.Val.V_PDOUBLE,
```

```
                                ptMod->Sigma.Val.V_PDOUBLE,
                                ptMod->Rho.Val.V_PDOUBLE,
                                Met->Par[0].Val.V_LONG,
   Met->Par[1].Val.V_PINT,
                                Met->Par[2].Val.V_ENUM.value,
 &(Met->Res[0].Val.V_DOUBLE));
}

static int CHK_OPT(MC_LeitaoGrzelakOosterlee)(void *Opt, void *Mod)
{
  if ((strcmp(((Option *)Opt)->Name, "CallEuro") == 0)  || (strcmp(((Option *)Op

    return OK;
  return WRONG;
}

#endif //PremiaCurrentVersion
static int MET(Init)(PricingMethod *Met, Option *Opt)
{
  if (Met->init == 0)
    {
      Met->init = 1;

      Met->Par[0].Val.V_LONG = 100000;
  Met->Par[1].Val.V_PINT = 8;
      Met->Par[2].Val.V_ENUM.value = 0;
      Met->Par[2].Val.V_ENUM.members = &PremiaEnumMCRNGs;
    }

  return OK;
}

PricingMethod MET(MC_LeitaoGrzelakOosterlee) =
{
  "MC_LeitaoGrzelakOosterlee",
  { {"N iterations", LONG, {100}, ALLOW},
{"Time Steps", PINT, {100}, ALLOW},
    {"RandomGenerator", ENUM, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_LeitaoGrzelakOosterlee),
```

```
  { {"Price", DOUBLE, {100}, FORBID},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(MC_LeitaoGrzelakOosterlee),
  CHK_mc,
  MET(Init)
};
}
```