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_finance.h"
#include "pnl/pnl_random.h"
#include "pnl/pnl_vector.h"
#include "pnl/pnl_mathtools.h"
#include "pnl/pnl_specfun.h"
#include "pnl/pnl_cdf.h"
#include "pnl/pnl_root.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2017+2) //The "#els
static int CHK_OPT(MC_Oosterlee_Sabr)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(MC_Oosterlee_Sabr)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else

/* AZEVEDO CHAVES HERVE
herve.azevedo-chaves@eleves.enpc.fr
hervetolentino@me.com
*/

static double beta_param;

typedef struct{
double a;
double b;
double u;
}newton_param;
```

```
static newton_param* new_newton_param (double a, double b,double u){
  newton_param *p;
  p = (newton_param*) malloc (sizeof (newton_param));
  p->a = a;
  p->b = b;
  p->u = u;
  return p;
}

static double maximum(double a){
return (a>0.0)?a:0.0;
}

static double pow_one_minus_beta(double x){return pow(x,1-beta_param);};

static void fdf(double c,double *f,double *df,void *p){// pnl_func for the enhan
const double a_l = ((newton_param *)p)->a;
double b = ((newton_param *)p)->b;
double u = ((newton_param *)p)->u;
double h = 1 - (2*(b+c)*(b+3*c))/(3*pnl_pow_i(b+2*c,2));
double P = (b+2*c)/pnl_pow_i(b+c,2);
double M = (h-1)*(1-3*h);
*f = (1-h*P*(1-h+0.5*(2-h)*M*P) - pow(a_l/(b+c),h))/(h*sqrt(2*P*(1+M*P))) - pnl_
double dh = -4.0*((2*b+3*c)*(b+2.0*c) - (b+c)*(b+3*c)*(b+3*c))/(3.*pnl_pow_i(b +
double dP = -4.0*c/pnl_pow_i(b+c,3);
double dM = -6.0*dh*h+4.*dh;
*df = ( (-dh*P-h*dP+2*h*P*dh+h*h*dP- ( (dh*M+h*dM)*P*P + 2*h*M*dP*P) + ( (0.5*dM
}

static void fdf2(double c,double *f,double *df,void *p){// function to use with
//  c  : root to determine
// *f  : H(a;b,c) = 1 - chi^2(a;b,c) - u
// *df : dH(a,b,c)/dc
double a_l = ((newton_param *)p)->a;//support
double b = ((newton_param *)p)->b;//degrees of freedom
double unif = ((newton_param *)p)->u;
double chi2,q,bound;
int status,which=1;
pnl_cdf_chn(&which, &chi2, &q, &a_l, &b, &c,&status, &bound);
*f = 1 - chi2 - unif;
```

```
*df = pow(0.5*(c/a_l),0.25*(b-2.0))*exp(-0.5*(a_l+c))*pnl_bessel_i(fabs(0.5*(b-2
}

int MCOosterleeSabr(double S0, NumFunc_1  *p, double T,double sigma0,double beta
{
  double K;
  //int flag_call;
  double delta;

  beta_param=beta;
  delta=T/(double)n;
  K = p->Par[0].Val.V_PDOUBLE;
  /* if ((p->Compute) == &Call) */
  /*   flag_call = 1; */
  /* else */
  /*   flag_call = 0; */

  pnl_rand_init(gen, 1, N);

   PnlVect *S_0 = pnl_vect_create_from_scalar(N,S0);// S_0 is a vector containin
PnlVect *S_delta = pnl_vect_create_from_zero(N);// S_delta is a vector which wil

PnlVect *sigma_0 = pnl_vect_create_from_scalar(N,sigma0);// sigma_0 is a vector
PnlVect *sigma_delta = pnl_vect_create_from_zero(N);// sigma_delta is a vector w

PnlRng *rng = pnl_rng_create(gen);// creating a generator
pnl_rng_sseed(rng, 0);/* rng must be initialized. When sseed=0, a default value

// Loop on each time step passing from S_0 to S_delta and sigma_0 to sigma_delta
for(int i = 0; i < n; i++){

//W2 is the N vector Brownian motion from the SDE : dsigma_t = alpha x sigma_t x
PnlVect *W2 = pnl_vect_new();// W2 = 0
pnl_vect_rng_normal(W2, N, rng);// W2 ~ N(0,1)
pnl_vect_mult_scalar(W2,sqrt(delta));// W2 = sqrt(delta) x W2 => W2 ~ N(0,delta)

PnlVect *A = pnl_vect_new();// A = NULL (use to store the integrated variance)


if( alpha > 0.0 ){
//Volatility computation : sigma_(i+1)*delta = sigma_(i) x exp(alphaxW2 - 0.5xde
```

```
pnl_vect_clone(sigma_delta,W2);// sigma_delta = W2
pnl_vect_mult_scalar(sigma_delta,alpha);// sigma_delta = alphaxW2
pnl_vect_minus_scalar(sigma_delta,0.5*delta*pnl_pow_i(alpha,2.0));// sigma_delta
pnl_vect_map_inplace (sigma_delta,exp);// sigma_delta = exp(alphaxW2 - 0.5xdelta
pnl_vect_mult_vect_term (sigma_delta,sigma_0);//sigma_delta = sigma_0 x exp(alph

// Integrated variance derivation by moment-matched log-normal distribution: A =

//Computation of the asymptotic mean: mean
PnlVect *mean = pnl_vect_copy(sigma_0);// mean = sigma_0
pnl_vect_mult_vect_term(mean,sigma_0);// mean = sigma_0^2
pnl_vect_mult_scalar(mean,delta);// mean = deltaxsigma_0^2

PnlVect *var = pnl_vect_copy(mean);// var = deltaxsigma_0^2

PnlVect *temp = pnl_vect_copy(W2);PnlVect *temp_2 = pnl_vect_copy(W2);PnlVect *t
// temp = temp_2 = temp_3 = temp_4 = W2 ~ N(0,delta)
pnl_vect_mult_vect_term(temp_2,temp);pnl_vect_mult_vect_term(temp_3,temp_2);pnl_
// temp_2 = W2^2 // temp_3 = W2^3 // temp_4 = W2^4
pnl_vect_axpby(2.0*pnl_pow_i(alpha,2.0)/3.0 - 0.1*3.0*delta*pnl_pow_i(alpha,4.0)
// temp = ( 2/3xalpha^2 - 3/10xdelta*alpha^4 )xW2^2 + ( alpha - 1/3xdelta*alpha^
pnl_vect_axpby(2.0*pnl_pow_i(alpha,4.0)/15.0,temp_4,pnl_pow_i(alpha,3.0)/3.0,tem
// temp_3 = ( 2/15xalpha^4 )xW2^4 + ( 1/3xalpha^3 )xW2^3
pnl_vect_plus_vect(temp,temp_3);
// temp = ( 2/3xalpha^2 - 3/10xdelta*alpha^4 )xW2^2 + ( alpha - 1/3xdelta*alpha^
pnl_vect_plus_scalar(temp,1.0 - delta*pnl_pow_i(alpha,2.0)/6.0 + 0.2*2.0*pnl_pow
// temp = ( 2/3xalpha^2 - 3/10xdelta*alpha^4 )xW2^2 + ( alpha - 1/3xdelta*alpha^
// 1 - 1/6xdeltaxalpha^2 + 2/5xdelta^2xalpha^4
pnl_vect_mult_vect_term(mean,temp);
// mean = deltaxsigma_0^2 x [ ( 2/3xalpha^2 - 3/10xdelta*alpha^4 )xW2^2 + ( alph
// 1 - 1/6xdeltaxalpha^2 + 2/5xdelta^2xalpha^4 ]
pnl_vect_free (&temp);pnl_vect_free (&temp_2);pnl_vect_free (&temp_3);pnl_vect_f

//Computation of the asymptotic variance: var
pnl_vect_mult_vect_term(var,var);// var = delta^2xsigma_0^4
pnl_vect_mult_scalar(var,delta*pnl_pow_i(alpha,2)/3.0);// var = 1/3xdelta^3xalph

//Computation of the parameters of the moment-matched log-normal distribution
PnlVect *sigma2 = pnl_vect_copy(mean);// sigma^2 = mean
pnl_vect_mult_vect_term(sigma2,sigma2);// sigma^2 = mean^2
pnl_vect_inv_term(sigma2);// sigma^2 = 1/mean^2
```

```
pnl_vect_mult_vect_term(sigma2,var);// sigma^2 = var/mean^2
pnl_vect_plus_scalar(sigma2,1.0);// sigma^2 = 1 + var/mean^2
pnl_vect_map_inplace (sigma2,log);// sigma^2 = log(1 + var/mean^2)

PnlVect *mu = pnl_vect_copy(mean);// mu = mean
pnl_vect_map_inplace(mu,log);// mu = log(mean)
pnl_vect_axpby(-0.5,sigma2,1.0,mu);// mu = log(mean) - 1/2xlog(1 + var/mean^2)

//Draw a vector of uniform random numers, U1, and determine their inverse by the
PnlVect *U1 = pnl_vect_new();// U1 = NULL
pnl_vect_rng_uni(U1, N, 0.0, 1.0, rng);// U1 ~ U([0,1])
pnl_vect_map_inplace(U1,pnl_inv_cdfnor);// U1 = N^-1 (U1)

//Log-Normal distribution
pnl_vect_clone(A,sigma2);// A = sigma^2
pnl_vect_map_inplace(A,sqrt);// A = sigma
pnl_vect_mult_vect_term(A,U1);// A = sigmaxN^-1(U1)
pnl_vect_plus_vect(A,mu);// A = sigma^-1(U1) + mu
pnl_vect_map_inplace(A,exp);// A = exp( sigmaxN^-1(U1) + mu )

pnl_vect_free (&mean);pnl_vect_free (&var);pnl_vect_free (&mu);pnl_vect_free (&s
}
else{//if alpha = 0 then the SDE only reads : dS_t = sigma_0 x S_t^beta x dS_t.
//then sigma_t is constant
pnl_vect_clone(sigma_delta,sigma_0);
//and A = \ int_(i)^(i+1) sigma_t^2 dt = deltaxsigma_0^2
A = pnl_vect_copy(sigma_0);// A = sigma_0
pnl_vect_mult_vect_term(A,sigma_0);// A = sigma_0^2
pnl_vect_mult_scalar(A,delta);// A = deltaxsigma_0^2
}




//low-biased scheme for SABR simulation

if (beta == 1.0){// Then we use the exact scheme of Broadie and Kaya: S_delta =
// U is the N vector Brownian motion from the Cholesky decomposition :
// dW1 = rhoxdW2 + sqrt(1-rho^2)xdU and dW2 = dW2.
PnlVect *U = pnl_vect_new();// U = NULL
pnl_vect_rng_normal(U, N, rng);// U ~ N(0,1)
```

```
pnl_vect_clone(S_delta,A);// S_delta = A = \ int_(i)^(i+1) sigma_t^2 dt :
pnl_vect_map_inplace(S_delta,sqrt);// S_delta = sqrt(A)
pnl_vect_mult_vect_term(S_delta,U);// S_delta = sqrt(A)xU
pnl_vect_axpby(-0.5,A,sqrt(1.0-pnl_pow_i(rho,2)),S_delta);// S_delta = sqrt(1-rh
if(alpha){
pnl_vect_axpby(rho/alpha,sigma_delta,1.,S_delta);// S_delta = sqrt(1-rho^2)xsqrt
pnl_vect_axpby(-rho/alpha,sigma_0,1.,S_delta);// S_delta = sqrt(1-rho^2)xsqrt(A)
}
pnl_vect_map_inplace(S_delta,exp);// S_delta = exp(sqrt(1-rho^2)xsqrt(A)xU - 1/2
pnl_vect_mult_vect_term(S_delta,S_0);// S_delta = S_0xexp(sqrt(1-rho^2)xsqrt(A)x
pnl_vect_free (&U);
}
else{
PnlVect *v = A;// *v = *A
pnl_vect_mult_scalar(v,1.0-pnl_pow_i(rho,2.0));// v = (1 - rho^2)xA
PnlVect *a = pnl_vect_copy(S_0);// a = S_0;
pnl_vect_map_inplace(a,pow_one_minus_beta);// a = S_0^(1-beta)

if( alpha > 0.0 ){
PnlVect *temp = pnl_vect_copy(sigma_delta);// temp = sigma_delta
pnl_vect_minus_vect(temp,sigma_0);// temp = sigma_delta - sigma_0
pnl_vect_axpby(rho/alpha,temp,1.0/(1.0-beta),a);// a = rho/alphax(sigma_delta -
pnl_vect_mult_vect_term(a,a);// a = ( rho/alphax(sigma_delta - sigma_0) + S_0^(1
pnl_vect_div_vect_term(a,v);// a = 1/v x ( rho/alphax(sigma_delta - sigma_0) + S
pnl_vect_free (&temp);
}
else{
pnl_vect_div_scalar(a,(1.0-beta));// a = S_0^(1-beta) / (1-beta)
pnl_vect_mult_vect_term(a,a);// a = S_0^(2-2xbeta) / (1-beta)^2
pnl_vect_div_vect_term(a,v);// a = 1/v x S_0^(2-2xbeta) / (1-beta)^2
}

const double k = (1.0-2.0*beta-pnl_pow_i(rho,2.0)*(1.0-beta))/((1.0-beta)*(1.0-p
double b = 2 - k;

//Draw a vector of uniform random numbers U
PnlVect *U = pnl_vect_new();// U = NULL;
pnl_vect_rng_uni(U, N, 0.0, 1.0, rng);// U ~ U([0,1])

for(int l = 0; l <N; l++){
```

```
//Compute the absorbtion probability p = P(S_delta = 0|S_0) = 1 - chi^2(a;b,0)
double p,q,bound,a_l=GET(a,l);//a_l = support and b = degrees of freedom of the
int status,which=1;
pnl_cdf_chi (&which, &p, &q, &a_l, &b, &status, &bound);
if (GET(S_0,l) == 0.0){
LET(S_delta,l) = 0.;
}
else if (GET(U,l) < q){
LET(S_delta,l) = 0.;
}
else{
//parameters for the moment-matched quadratic Gaussian approximation
const double m = k + a_l;// mean of the noncentral chi-square distribution
const double phi = 2.0*(k+2.0*a_l)/(m*m);

             const double phi_thres = 2.0;//threshold level to determine which a

            if ((phi > 0.0) && (phi <= phi_thres) && (m >= 0)){//S is not small
                 const double e2 = 2.0/phi - 1.0 + sqrt(2./phi)*sqrt(2./phi-1.);
                 const double d = m/(1.0+e2);
                 LET(S_delta,l) = pow((pnl_pow_i(1.0-beta,2.0)*GET(v,l)*d*pnl_po
}
else if ( (phi > phi_thres) || ((m < 0.0) && ((0.0 < phi) && (phi <= phi_thres))
// Newton-Raphson's algorithm to determine the root c* of the function H(a,b,c)
const int n_max = 20000;// maximum number of iterations
                 const double epsilon = 1.E-6;// tolerance
double c;

PnlFuncDFunc Func2;//pnl function that will contains the functions H(a,b,c) and
Func2.F = fdf2;
newton_param *param = new_newton_param(a_l,b,GET(U,l));
Func2.params = param;
pnl_root_newton (&Func2, a_l, epsilon, epsilon, n_max, &c);
// Find the root c of H(c) = (1 - chi^2(a;b,c) - U) starting from a_l using Newt
// Armijo's line search is used to make sure |H(c)| decreases along the iteratio
                 LET(S_delta,l) = pow(pnl_pow_i(1.0-beta,2.0)*c*GET(v,l),1.0/(2.
}
else{
LET(S_delta,l) = 0.0;
}
}
```

```
}
//Free memory
pnl_vect_free (&v);
pnl_vect_free (&a);
pnl_vect_free (&U);
}
//Free memory
pnl_vect_clone(S_0,S_delta);
pnl_vect_clone(sigma_0,sigma_delta);

pnl_vect_free (&W2);
}
//European call : payoff = (S_T - K)_+
pnl_vect_minus_scalar(S_delta,K);// S_delta = S_T - K
pnl_vect_map_inplace(S_delta,maximum);// S_delta = max(S_T - K,0)

*price = pnl_vect_sum(S_delta)/N;// price = 1/N x \ sum_1^N max(S_T - K,0)

//Confidence Interval 95%
pnl_vect_minus_scalar(S_delta,*price);// S_delta = max(S_T - K,0) - 1/N x \ sum_
pnl_vect_mult_vect_term(S_delta,S_delta);// S_delta = ( max(S_T - K,0) - 1/N x \
double std = pnl_vect_sum(S_delta)/(N-1);// std = 1/(N-1) x \ sum_1^N ( max(S_T
*up = *price + 1.96*sqrt(std)/sqrt(1.0*N);
*down = *price - 1.96*sqrt(std)/sqrt(1.0*N);

//Free memory
pnl_rng_free (&rng);pnl_vect_free (&S_0);pnl_vect_free (&S_delta);pnl_vect_free

  return OK;
}


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

      return MCOosterleeSabr(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), &(Met->Res[1].Val.V_DOUBLE), &(Met->Res[2].Val.V_D
                               );
}

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

    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 = 16;
      Met->Par[2].Val.V_ENUM.value = 0;
      Met->Par[2].Val.V_ENUM.members = &PremiaEnumMCRNGs;
    }

  return OK;
}

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

```
  CALC(MC_Oosterlee_Sabr),
  { {"Price", DOUBLE, {100}, FORBID},
{"Inf Price", DOUBLE, {100}, FORBID},
 {"Sup Price", DOUBLE, {100}, FORBID},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(MC_Oosterlee_Sabr),
  CHK_mc,
  MET(Init)
};
}
```