Help

```
#include "mc_lmm_glassermanzhao.h"

/** "Arbitrage-Free Discretization Of Lognormal Forward
    Libor Model" by Glasserman and Zhao (2000)
* We consider a tenor structure 0=T_0 < T_1 < ... < T_N <
    T_N+1 equaly spaced
* and Libor rates L(t, T_0), L(t,T_2),..., L(t, T_N) for a
    certain date t. L(., T_i) is set at Ti and payed at Ti+ten
    or.
* Convention: for t>T_i L(t,T_i)=L(T_i,T_i)
* Simulation can be done with the function "Sim_Libor_Glass
    erman" under two measure : Terminal measure and Spot measu
    re.
* @param start_index index of time from wich simulation
    starts. T(start_index) to T(end_index)
* @param end_index index of last date of simulation.
* @param ptLOld Libor structure, contains initial value of
    libor rates
* @param ptVol Volatility structure contains libor
    volatility deterministic function
* @param generator the index of the random generator to be
    used
* @param NbrMCsimulation the number of samples
* @param NbrStepPerTenor number of steps of discretization
    between T(i) and T(i+1)
* @param save_all_paths flag to decide wether we store the
    simulated value of libors at each date T(i) (if save_all_
    paths=1) or we store only the simulated value of libors at th
    e end, ie T(end_index).
* @param LiborPathsMatrix PnlMat contains libor simulated
    paths.
* @param save_brownian flag to decide to store brownian mot
    ion values.
* save_brownian==0. we dont store the value brownian motio
    n used in the simulation.
* save_brownian==1. we store the value brownian motion used
     in the simulation, at each T(i).
* save_brownian==2. we also store intermediate steps, betwe
    en T(i) and T(i+1). used in Schoenmakers et al. algorithm.
* @param BrownianMatrixPaths PnlMat contains brownian motio
```

```
    n simulated paths.
* @param flag_numeraire measure under wich simulation is
    done.
* flag_numeraire=0->Terminal measure, flag_numeraire=1->Spo
    t measure
*/

void Sim_Libor_Glasserman(int start_index, int end_index,
    Libor *ptLOld, Volatility *ptVol, int generator, int NbrMCsimu
    lation, int NbrStepPerTenor, int save_all_paths, PnlMat *
    LiborPathsMatrix, int save_brownian, PnlMat *BrownianMatrix
    Paths, int flag_numeraire)
{
    if (flag_numeraire==0)
    {
        Sim_Libor_Glasserman_TerminalMeasure(start_index,
    end_index, ptLOld, ptVol, generator, NbrMCsimulation, Nb
    rStepPerTenor, save_all_paths, LiborPathsMatrix, save_brownia
    n, BrownianMatrixPaths);
    }
    else
    {
        Sim_Libor_Glasserman_SpotMeasure(start_index, end_
    index, ptLOld, ptVol, generator, NbrMCsimulation, NbrStepPe
    rTenor, save_all_paths, LiborPathsMatrix, save_brownian, Br
    ownianMatrixPaths);
    }
}

/** "Swaption_Payoff_Discounted" computes the swaption payo
    ff discounted with respect to the relative numeraire.
* @param ptL Libor structure, contains value of libor ra
    tes.
* @param ptSwpt Swaption structure contains swaption inform
    ation.
* @param flag_numeraire flag decide the numeraire we dis
    counte with
* flag_numeraire=0->Terminal measure numeraire, flag_numera
    ire=1->Spot measure numeraire.
*/
double Swaption_Payoff_Discounted(Libor *ptL, Swaption *pt
```

```
    Swpt, NumFunc_1 *p, int flag_numeraire)
{
    if (flag_numeraire==0)
    {
        return Swaption_Payoff_TerminalMeasure(ptL, ptSwpt,
     p);
    }
    else
    {
        return Swaption_Payoff_SpotMeasure(ptL, ptSwpt, p);
    }
}


int Sim_Libor_Glasserman_TerminalMeasure(int start_index,
    int end_index, Libor *ptLOld, Volatility *ptVol, int     generator, int NbrM
    paths, PnlMat *LiborPathsMatrix, int save_brownian, PnlMat *Br
    ownianMatrixPaths)
{
    double tenor;
    double Di, tk, Ti, dt, sqrt_dt;
    int i, j, k, m, l, N, Nfac, Nsteps;

    PnlVect *Xvalue_0, *Xvalue_t;
    PnlVect *sigma;
    PnlVect *lambda1; // Volatility vector of the the proc
    ess L(.,Ti,Ti+1) valued at the runing time tk
    PnlVect *lambda2; // Volatility vector of the the proc
    ess L(.,Ti+1,Ti+2) valued at the runing time tk
    PnlVect *Xprev;
    PnlVect *ptW, *ptW_sum;

    Nfac = ptVol->numberOfFactors;
    N = ptLOld->numberOfMaturities;
    tenor = ptLOld->tenor;
    dt = tenor/NbrStepPerTenor;
    sqrt_dt = sqrt(dt);

    Xvalue_0 = pnl_vect_create(N-1);
    Xvalue_t = pnl_vect_create(N-1);
    Xprev    = pnl_vect_create(N-1);
```

```
ptW      = pnl_vect_create(Nfac);
ptW_sum  = pnl_vect_create(Nfac);
lambda1  = pnl_vect_create(Nfac);
lambda2  = pnl_vect_create(Nfac);
sigma    = pnl_vect_create(Nfac);

Nsteps = end_index - start_index;

// vector Xvalue is a transformation of libor rates.
cf Glasserman Zhao (2000)
// Initial value of the vector Xvalue
LET(Xvalue_0, N-2) = GET(ptLOld->libor,N-1); // X(0,T(
N-2)) = Libor(0, T(N-1), T(N))
for (i=N-3; i>=0; i--)
{
    LET(Xvalue_0, i) = GET(Xvalue_0, i+1) * GET(ptLOld-
>libor,i+1) * (1/GET(ptLOld->libor,i+2) + tenor) ;
}

if (save_all_paths==1)
{
    pnl_mat_resize(LiborPathsMatrix, 1 + Nsteps*NbrMCs
imulation, N);
    pnl_mat_set_row(LiborPathsMatrix, ptLOld->libor, 0)
;

    if (save_brownian==1)
    {
        pnl_mat_resize(BrownianMatrixPaths, Nsteps*NbrM
Csimulation, Nfac);
    }

    else if (save_brownian==2)
    {
        pnl_mat_resize(BrownianMatrixPaths, NbrStepPe
rTenor*Nsteps*NbrMCsimulation, Nfac);
    }
}
else
{
    pnl_mat_resize(LiborPathsMatrix, NbrMCsimulation,
```

```
N);
    if (save_brownian==1)
    {
        pnl_mat_resize(BrownianMatrixPaths, NbrMCsimu
lation, Nfac);
    }
}

for (m=0; m<NbrMCsimulation; m++)
{
    pnl_vect_set_zero(ptW_sum);
    pnl_vect_clone(Xprev, Xvalue_0);
    tk = start_index*tenor;

    for (l=1; l<=Nsteps; l++) // Time Loop from T(star
t_index) to T(end_index)
    {
        for (k=1 ; k<=NbrStepPerTenor; k++)
        {
            pnl_vect_rand_normal(ptW, Nfac, generator);
            pnl_vect_axpby(sqrt_dt, ptW, 1., ptW_sum);
/* y := a x + b y */

            // save_brownian==2. We also save interme
diate steps.
            if (save_all_paths==1 && save_brownian==2)
            {
                pnl_mat_set_row(BrownianMatrixPaths, pt
W_sum, k-1 + (l-1)*NbrStepPerTenor + m*NbrStepPerTenor*Ns
teps);
            }

            // Last component of the vector X
            Ti = (N-1) * tenor;
            for (j=0; j<Nfac; j++)
            {
                LET(lambda2,j) = evalVolatility(ptVol,
j, tk, Ti); // sigma(tk, T(N-1))
                LET(sigma,j) = GET(lambda2, j);
            }
            // Discretization equation for the last
```

```
component of the vector X:
            LET(Xvalue_t, N-2) = GET(Xprev, N-2) * exp(
-0.5 * pnl_vect_scalar_prod(sigma, sigma)*dt + sqrt(dt)*pn
l_vect_scalar_prod(sigma, ptW));

            // For the rest of the components of vector
 X.
            Di = 1;
            for (i=N-3; i>=0; i--)
            {
                Di += tenor * GET(Xprev, i+1);
                Ti -= tenor ;

                for (j=0; j<Nfac; j++)
                {
                    LET(lambda1,j) = evalVolatility(pt    Vol, j, tk, Ti);
                    LET(sigma,j) = GET(sigma,j) + GET(
lambda1,j) - GET(lambda2,j) + tenor*GET(Xprev, i+1)*GET(lam
bda2,j)/Di;
                    LET(lambda2,j) = GET(lambda1,j);
                }

                // Discretization equation :
                LET(Xvalue_t,i) = GET(Xprev,i) * exp(-0
.5 * pnl_vect_scalar_prod(sigma, sigma)*dt + sqrt(dt)*pnl_
vect_scalar_prod(sigma, ptW));
            }

            pnl_vect_clone(Xprev, Xvalue_t);
            tk += dt;
        }

        // If save_all_paths=1, we store the simulated
value of libors at each date T(i).
        // Transform the vector X into Libor values an
d store then in the matrix LiborPathsMatrix.

        if (save_all_paths==1)
        {
            MLET(LiborPathsMatrix, 1+m*Nsteps, N-1) =
GET(Xvalue_t, N-2);
```

```
            Di = 1;

            for (i=N-2; i>=l; i--)
            {
                Di += tenor * GET(Xvalue_t, i);
                MLET(LiborPathsMatrix, l+m*Nsteps, i) =
 GET(Xvalue_t, i-1) / Di;
            }
            for (i=l-1; i>=0; i--)
            {
                MLET(LiborPathsMatrix, l+m*Nsteps, i) =
 MGET(LiborPathsMatrix, l-1+m*Nsteps, i); // L(t, T(0), T(
1)). It stays constant
            }

            // Store the value brownian motion used in
the simulation.
            if (save_brownian==1)
            {
                pnl_mat_set_row(BrownianMatrixPaths, pt
W_sum, l-1 + m*Nsteps);
            }

        }

    }

    // If save_all_paths=0, we store only the simulated
 value of libors at the end, ie T(end_index).
    // Transform the vector X into Libor values and sto
re then in the matrix LiborPathsMatrix.
    if (save_all_paths==0)
    {
        MLET(LiborPathsMatrix, m, N-1) = GET(Xvalue_t,
N-2);
        Di = 1;
        for (i=N-2; i>0; i--)
        {
            Di += tenor * GET(Xvalue_t, i);
            MLET(LiborPathsMatrix, m, i) = GET(Xvalue_
t, i-1) / Di;
```

```
            }
            MLET(LiborPathsMatrix, m, 0) = GET(ptLOld->
    libor,0); // L(t, T(0), T(1)). It stays constant

            if (save_brownian==1)
            {
                pnl_mat_set_row(BrownianMatrixPaths, ptW_su
    m, m);
            }
        }
    }

    pnl_vect_free(&Xvalue_0);
    pnl_vect_free(&Xvalue_t);
    pnl_vect_free(&sigma);
    pnl_vect_free(&lambda1);
    pnl_vect_free(&lambda2);
    pnl_vect_free(&Xprev);
    pnl_vect_free(&ptW);
    pnl_vect_free(&ptW_sum);

    return(1);
}

double Swaption_Payoff_TerminalMeasure(Libor *ptL, Swaptio
    n *ptSwpt, NumFunc_1 *p)
{
    int j, alpha, beta, N;
    double SumDi, tenor, Di;
    double swaption_maturity, swap_maturity, swaption_stri
    ke, payoff;

    N = ptL->numberOfMaturities;
    tenor = ptL->tenor;

    swaption_maturity = ptSwpt->swaptionMaturity;
    swap_maturity = ptSwpt->swapMaturity;
    swaption_strike = ptSwpt->strike;

    alpha = intapprox(swaption_maturity/tenor); // T_alpha
    is the swaption maturity
```

```
    beta  = intapprox(swap_maturity/tenor); // T_beta is th
    e swap maturity

    p->Par[0].Val.V_DOUBLE = 0.0;

    // Di discounted bond.
    // D_beta
    Di = 1.;
    for (j=beta; j<N; j++)
    {
        Di *= (1+tenor*GET(ptL->libor, j));
    }
    payoff = Di;
    SumDi = Di;

    // sum D_j for j from alpha+1 to beta
    for (j=beta-1; j>alpha; j--)
    {
        Di *= (1+tenor*GET(ptL->libor, j));
        SumDi += Di;
    }

    payoff += swaption_strike * tenor * SumDi;

    // D_alpha
    Di *= (1+tenor*GET(ptL->libor, alpha));

    payoff -=  Di;

    payoff = ((p->Compute)(p->Par, payoff)); // Payoff

    return payoff;

}

int Sim_Libor_Glasserman_SpotMeasure(int start_index, int
    end_index, Libor *ptLOld, Volatility *ptVol, int generator,
    int NbrMCsimulation, int NbrStepPerTenor, int save_all_paths,
     PnlMat *LiborPathsMatrix, int save_brownian, PnlMat *Brow
    nianMatrixPaths)
{
```

```
double tenor, tk, Ti, dt, SumVi, Prod_i, sqrt_dt;
int i, j, k, m, l, N, Nfac, Nsteps, eta;

PnlVect *Vvalue_0, *Vvalue_t;
PnlVect *sigma;
PnlVect *lambda;
PnlVect *ci;
PnlVect *Vprev;
PnlVect *ptW, *ptW_sum;

Nfac = ptVol->numberOfFactors;
N = ptLOld->numberOfMaturities;
tenor = ptLOld->tenor;
dt = tenor/NbrStepPerTenor;
sqrt_dt = sqrt(dt);

Vvalue_0 = pnl_vect_create(N);
Vvalue_t = pnl_vect_create(N);
Vprev    = pnl_vect_create(N);
ptW      = pnl_vect_create(Nfac);
ptW_sum  = pnl_vect_create(Nfac);
lambda   = pnl_vect_create(Nfac);
ci       = pnl_vect_create(Nfac);
sigma    = pnl_vect_create(Nfac);

Nsteps = end_index - start_index;

// Initial value of the vector V
Prod_i = 1. / (1 + tenor*GET(ptLOld->libor, 1));
LET(Vvalue_0, 0) = tenor * GET(ptLOld->libor, 1) * Prod
_i;

for (i=1; i<N-1; i++)
{
    Prod_i /= (1 + tenor*GET(ptLOld->libor, i+1));
    LET(Vvalue_0, i) = tenor * GET(ptLOld->libor,i+1) *
 Prod_i;
}
LET(Vvalue_0, N-1) = Prod_i;

if (save_all_paths==1)
```

```
{
    pnl_mat_resize(LiborPathsMatrix, 1 + Nsteps*NbrMCs
imulation, N);
    pnl_mat_set_row(LiborPathsMatrix, ptLOld->libor, 0)
;

    if (save_brownian==1)
    {
        pnl_mat_resize(BrownianMatrixPaths, Nsteps*NbrM
Csimulation, Nfac);
    }

    else if (save_brownian==2)
    {
        pnl_mat_resize(BrownianMatrixPaths, NbrStepPe
rTenor*Nsteps*NbrMCsimulation, Nfac);
    }
}
else
{
    pnl_mat_resize(LiborPathsMatrix, NbrMCsimulation,
N);
    if (save_brownian==1)
    {
        pnl_mat_resize(BrownianMatrixPaths, NbrMCsimu
lation, Nfac);
    }
}

for (m=0; m<NbrMCsimulation; m++)
{
    pnl_vect_set_zero(ptW_sum);
    tk = start_index*tenor;
    pnl_vect_clone(Vprev, Vvalue_0);

    for (l=1; l<=Nsteps; l++) // Time Loop from T(1)
to T(N-1)
    {
        for (k=1 ; k<=NbrStepPerTenor; k++)
        {
            SumVi = pnl_vect_sum(Vprev);
```

```
            eta = (int) ceil(tk/tenor);
            pnl_vect_rand_normal(ptW, Nfac, generator);
            pnl_vect_axpby(sqrt_dt, ptW, 1., ptW_sum);
/* y := a x + b y */

            // save_brownian==2. We also save interme
diate steps.
            if (save_all_paths==1 && save_brownian==2)
            {
                pnl_mat_set_row(BrownianMatrixPaths, pt
W_sum, k-1 + (l-1)*NbrStepPerTenor + m*NbrStepPerTenor*Ns
teps);
            }

            /// Fisrt component of the vector V
            Ti = tenor;
            for (j=0; j<Nfac; j++)
            {
                LET(lambda, j) = evalVolatility(ptVol,
j, tk, Ti); // vol(tk, T_1)
                LET(sigma, j) = GET(lambda, j) * (SumV
i - GET(Vprev, 0)) / SumVi;
            }
            // Discretization equation for the first
component of the vector X:
            LET(Vvalue_t, 0) = GET(Vprev, 0) * exp(-0.5
 * pnl_vect_scalar_prod(sigma, sigma)*dt + sqrt(dt)*pnl_
vect_scalar_prod(sigma, ptW));

            pnl_vect_set_double(ci, 0.0);

            /// For the rest of the components of vec
tor V
            for (i=1; i<N; i++)
            {
                Ti += tenor;

                if (i>=eta)
                {
                    for (j=0; j<Nfac; j++)
                    {
```

```
                        LET(ci,j) +=  GET(Vprev, i-1) /
 SumVi * GET(lambda,j);
                    }
                }

                SumVi = SumVi - GET(Vprev, i-1);

                for (j=0; j<Nfac; j++)
                {
                    if (i==N-1)
                    {
                        LET(lambda,j) = 0;
                    }
                    else
                    {
                        LET(lambda,j) = evalVolatility(
ptVol, j, tk, Ti);
                    }

                    LET(sigma,j) = GET(lambda, j) * (Su
mVi - GET(Vprev, i)) / SumVi - GET(ci,j);
                }

                // Discretization equation :
                LET(Vvalue_t,i) = GET(Vprev,i) * exp(-0
.5 * pnl_vect_scalar_prod(sigma, sigma)*dt + sqrt(dt)*pnl_
vect_scalar_prod(sigma, ptW));

            }

            pnl_vect_clone(Vprev, Vvalue_t);
            tk += dt;
        }

        // If save_all_paths=1, we store the simulated
value of libors at each date T(i).
        // Transform the vector V into Libor values an
d store then in the matrix LiborPathsMatrix.
        if (save_all_paths==1)
        {
            SumVi = GET(Vvalue_t, N-1);
```

```
            for (i=N-1; i>0; i--)
            {
                MLET(LiborPathsMatrix, l+m*Nsteps, i) =
 GET(Vvalue_t, i-1) / (tenor*SumVi);
                SumVi += GET(Vvalue_t, i-1);
            }
            MLET(LiborPathsMatrix, l+m*Nsteps, 0) = GET
(ptLOld->libor,0); // L(t, T(0), T(1)). It stays constant

            // Store the value brownian motion used in
the simulation.
            if (save_brownian==1)
            {
                pnl_mat_set_row(BrownianMatrixPaths, pt
W_sum, l-1 + m*Nsteps);
            }
        }

    }

    // If save_all_paths=0, we store only the simulated
 value of libors at the end, ie T(end_index).
    // Transform the vector V into Libor values and sto
re then in the matrix LiborPathsMatrix.
    if (save_all_paths==0)
    {
        SumVi = GET(Vvalue_t, N-1);
        for (i=N-1; i>0; i--)
        {
            MLET(LiborPathsMatrix, m, i) = GET(Vvalue_
t, i-1) / (tenor*SumVi);
            SumVi += GET(Vvalue_t, i-1);
        }
        MLET(LiborPathsMatrix, m, 0) = GET(ptLOld->
libor,0); // L(t, T(0), T(1)). It stays constant

        // Store the value brownian motion used in the
simulation.
        if (save_brownian==1)
        {
            pnl_mat_set_row(BrownianMatrixPaths, ptW_su
```

```
    m, m);
            }
        }
    }

    pnl_vect_free(&Vvalue_0);
    pnl_vect_free(&Vvalue_t);
    pnl_vect_free(&sigma);
    pnl_vect_free(&lambda);
    pnl_vect_free(&ci);
    pnl_vect_free(&Vprev);
    pnl_vect_free(&ptW);
    pnl_vect_free(&ptW_sum);

    return(1);
}

double Swaption_Payoff_SpotMeasure(Libor *ptL, Swaption *pt
    Swpt, NumFunc_1 *p)
{
    int j, alpha, beta;
    double SumDi, tenor, Di;
    double swaption_maturity, swap_maturity, swaption_stri
    ke, payoff;

    tenor = ptL->tenor;

    swaption_maturity = ptSwpt->swaptionMaturity;
    swap_maturity = ptSwpt->swapMaturity;
    swaption_strike = ptSwpt->strike;

    alpha = intapprox(swaption_maturity/tenor); // T_alpha
    is the swaption maturity
    beta  = intapprox(swap_maturity/tenor); // T_beta is th
    e swap maturity

    p->Par[0].Val.V_DOUBLE = 0.0;

    // D_alpha
    Di = 1;
    for (j=0; j<alpha; j++)
```

```
    {
        Di /= (1+tenor*GET(ptL->libor, j));
    }
    payoff = Di;

    // sum D_j for j from alpha+1 to beta
    SumDi = 0;
    for (j=alpha; j<beta; j++)
    {
        Di /= (1+tenor*GET(ptL->libor, j));
        SumDi += Di;
    }

    payoff = payoff - Di - swaption_strike * tenor * SumDi;

    payoff = ((p->Compute)(p->Par, -payoff)); // Payoff

    return payoff;

}


double european_swaption_ap_rebonato(double valuation_date,
     NumFunc_1 *p, Libor *ptLib, Volatility *ptVol, Swaption *
    ptSwpt)
{
    int i, j, k, l, e, alpha, beta, n, Nfac, Nstep_integrat
    ion,  payer_or_receiver;
    double t, swaption_maturity, swap_maturity, swaption_
    strike, tenor, swap_rate, vol_swaption, integrale, somme_
    integrale, Ti, Tj, dt;
    double sum_discount_factor, black_volatility, d1, d2,
    price, zc;

    PnlVect * weight;

    swaption_maturity = ptSwpt->swaptionMaturity;
    swap_maturity = ptSwpt->swapMaturity;
    swaption_strike = ptSwpt->strike;
    tenor = ptSwpt->tenor;
    payer_or_receiver = ((p->Compute)==&Put);
```

```
e = intapprox(valuation_date/tenor);
alpha = intapprox(swaption_maturity/tenor); // index of
 swaption_maturity
beta  = intapprox(swap_maturity/tenor); // index of swa
p_maturity
n = beta - alpha; // Nbr of payements dates
Nfac = ptVol->numberOfFactors; // Nbr of factors in dif
fusion process

weight = pnl_vect_create(n);

LET(weight, 0) = 1.;

for (i=alpha; i<beta ; i++)
{
    LET(weight, i-alpha) /= (1+tenor*GET(ptLib->libor,
i));
}

sum_discount_factor = pnl_vect_sum(weight);

pnl_vect_div_double(weight, sum_discount_factor); //
Normilization of the weights

// swap_rate(0) = sum over i of weight(i)*LiborRate(0,
Ti,Ti+1) , see Brigo&Mercurio book
swap_rate = 0;
for (i=alpha; i<beta ; i++)
{
    swap_rate += GET(weight, i-alpha) * GET(ptLib->
libor, i);
}

Nstep_integration = 40; // number of step used to compu
te the integral of volatility(t,Ti)*volatility(t,Tj) for t
in [0,T_alpha]
dt = (swaption_maturity-valuation_date)/ Nstep_integrat
ion; // step for the integration
vol_swaption = 0; // Black's volatility of the swaption
```

```
for (i = 0; i<n ; i++)
{
    Ti = swaption_maturity + i * tenor;
    for (j = 0; j<n ; j++)
    {
        Tj = swaption_maturity + j * tenor;
        somme_integrale = 0;
        for (k=0; k<Nfac; k++) // computation of the
integral of volatility(t,Ti)*volatility(t,Tj) for t in [0,T_alpha]
        {
            // We use the simple trapezoidal rule
            integrale = evalVolatility(ptVol, k, valua
tion_date, Ti) * evalVolatility(ptVol, k, valuation_date, Tj
);
            integrale += evalVolatility(ptVol, k, swapt
ion_maturity, Ti) * evalVolatility(ptVol, k, swaption_matu
rity, Tj);
            integrale *= 0.5;

            for ( l=1 ; l<Nstep_integration; l++)
            {
                t = valuation_date+l*dt;
                integrale += evalVolatility(ptVol, k,
t, Ti) * evalVolatility(ptVol, k, t, Tj);
            }
            integrale *= dt;
            somme_integrale += integrale;
        }

        vol_swaption += GET(weight, i) * GET(weight, j)
 * GET(ptLib->libor,alpha+i) * GET(ptLib->libor,alpha+j) *
 somme_integrale;
    }
}

vol_swaption = vol_swaption / SQR(swap_rate) ;

black_volatility = sqrt(vol_swaption);

d1 = (log(swap_rate/swaption_strike))/ black_volatilit
y + 0.5 * black_volatility;
```

```
    d2 = d1 - black_volatility;

    zc = 1;
    for (i=e; i<alpha ; i++)
    {
        zc /= (1+tenor*GET(ptLib->libor,i));
    }

    sum_discount_factor = 0;
    for (i=alpha; i<beta ; i++)
    {
        zc /= (1+tenor*GET(ptLib->libor,i));
        sum_discount_factor += tenor*zc;
    }

    if (payer_or_receiver==1) // Case of Payer Swaption
    {
        price = sum_discount_factor * (swap_rate * cdf_nor(
    d1) - swaption_strike * cdf_nor(d2));
    }

    else  // if (payer_or_receiver==0) Case of Receiver Swa
    ption
    {
        price = sum_discount_factor * (swap_rate * (cdf_nor
    (d1)-1) - swaption_strike * (cdf_nor(d2)-1));
    }

    pnl_vect_free(&weight);

    return price;
}


double Numeraire(int i, Libor *ptLib_current, int flag_
    numeraire)
{
    int j, N;
    double B_j, tenor;

    N = ptLib_current->numberOfMaturities;
```

```
    tenor = ptLib_current->tenor;

    B_j = 1.;

    if (flag_numeraire==0)
    {
        for (j=i; j<N; j++)
        {
            B_j /= (1+tenor*GET(ptLib_current->libor, j));
        }

        return B_j;
    }

    else
    {
        for (j=0; j<i; j++)
        {
            B_j *= (1+tenor*GET(ptLib_current->libor, j));
        }

        return B_j;
    }
}
```

References