Help

```
#include "mc_lmm_glassermanzhao.h"

/** "Arbitrage-Free Discretization Of Lognormal Forward Libor Model" by Glasserm
* 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(.,
* Convention: for t>T_i L(t,T_i)=L(T_i,T_i)
* Simulation can be done with the function "Sim_Libor_Glasserman" under two meas
* @param start_index index of time from wich simulation starts. T(start_index) t
* @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 func
* @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+
* @param save_all_paths flag to decide wether we store the simulated value of li
* @param LiborPathsMatrix PnlMat contains libor simulated paths.
* @param save_brownian flag to decide to store brownian motion values.
* save_brownian==0. we dont store the value brownian motion used in the simulati
* save_brownian==1. we store the value brownian motion used in the simulation, a
* save_brownian==2. we also store intermediate steps, between T(i) and T(i+1). u
* @param BrownianMatrixPaths PnlMat contains brownian motion simulated paths.
* @param flag_numeraire measure under wich simulation is done.
* flag_numeraire=0->Terminal measure, flag_numeraire=1->Spot measure
*/

void Sim_Libor_Glasserman(int start_index, int end_index, Libor *ptLOld, Volatil
{
  if (flag_numeraire == 0)
    {
      Sim_Libor_Glasserman_TerminalMeasure(start_index, end_index, ptLOld, ptVol
    }
  else
    {
      Sim_Libor_Glasserman_SpotMeasure(start_index, end_index, ptLOld, ptVol, ge
    }
}

/** "Swaption_Payoff_Discounted" computes the swaption payoff discounted with re
* @param ptL Libor structure, contains value of libor rates.
```

```
* @param ptSwpt Swaption structure contains swaption information.
* @param flag_numeraire flag decide the numeraire we discounte with
* flag_numeraire=0->Terminal measure numeraire, flag_numeraire=1->Spot measure n
*/
double Swaption_Payoff_Discounted(Libor *ptL, Swaption *ptSwpt, NumFunc_1 *p, in
{
  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 *
{
  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 process L(.,Ti,Ti+1) valued
  PnlVect *lambda2; // Volatility vector of the the process L(.,Ti+1,Ti+2) value
  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(
for (i = N - 3; i >= 0; i--)
  {
    LET(Xvalue_0, i) = GET(Xvalue_0, i + 1) * GET(ptLOld->libor, i + 1) * (1 /
  }

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

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

    else if (save_brownian == 2)
      {
        pnl_mat_resize(BrownianMatrixPaths, NbrStepPerTenor * Nsteps * NbrMCsi
      }
  }
else
  {
    pnl_mat_resize(LiborPathsMatrix, NbrMCsimulation, N);
    if (save_brownian == 1)
      {
        pnl_mat_resize(BrownianMatrixPaths, NbrMCsimulation, 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(start_index) to T(end_in
  {
    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 intermediate steps.
        if (save_all_paths == 1 && save_brownian == 2)
          {
            pnl_mat_set_row(BrownianMatrixPaths, ptW_sum, k - 1 + (l - 1)*
          }

        // 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(t
            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_sca

        // 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(ptVol, j, tk, Ti);
                LET(sigma, j) = GET(sigma, j) + GET(lambda1, j) - GET(lamb
                LET(lambda2, j) = GET(lambda1, j);
              }

            // Discretization equation :
```

```
          LET(Xvalue_t, i) = GET(Xprev, i) * exp(-0.5 * pnl_vect_scalar_
        }

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

  // If save_all_paths=1, we store the simulated value of libors at each
  // Transform the vector X into Libor values and store then in the matr

  if (save_all_paths == 1)
    {
      MLET(LiborPathsMatrix, l + m * Nsteps, N - 1) = GET(Xvalue_t, N -
      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 -
        }
      for (i = l - 1; i >= 0; i--)
        {
          MLET(LiborPathsMatrix, l + m * Nsteps, i) = MGET(LiborPathsMat
        }

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

    }

}

// If save_all_paths=0, we store only the simulated value of libors at the
// Transform the vector X into Libor values and store then in the matrix L
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(

        if (save_brownian == 1)
          {
            pnl_mat_set_row(BrownianMatrixPaths, ptW_sum, 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, Swaption *ptSwpt, NumFunc_1 *
{
  int j, alpha, beta, N;
  double SumDi, tenor, Di;
  double swaption_maturity, swap_maturity, swaption_strike, payoff;

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

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

  alpha = pnl_iround(swaption_maturity / tenor); // T_alpha is the swaption matu
  beta  = pnl_iround(swap_maturity / tenor); // T_beta is the 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 *ptLO
{
  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 * NbrMCsimulation, N);
    pnl_mat_set_row(LiborPathsMatrix, ptLOld->libor, 0);

    if (save_brownian == 1)
      {
        pnl_mat_resize(BrownianMatrixPaths, Nsteps * NbrMCsimulation, Nfac);
      }
```

```
    else if (save_brownian == 2)
      {
        pnl_mat_resize(BrownianMatrixPaths, NbrStepPerTenor * Nsteps * NbrMCsi
      }
  }
else
  {
    pnl_mat_resize(LiborPathsMatrix, NbrMCsimulation, N);
    if (save_brownian == 1)
      {
        pnl_mat_resize(BrownianMatrixPaths, NbrMCsimulation, 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 intermediate steps.
            if (save_all_paths == 1 && save_brownian == 2)
              {
                pnl_mat_set_row(BrownianMatrixPaths, ptW_sum, k - 1 + (l - 1)*
              }

            /// Fisrt component of the vector V
            Ti = tenor;
            for (j = 0; j < Nfac; j++)
              {
                LET(lambda, j) = evalVolatility(ptVol, j, tk, Ti); // vol(tk,
```

```
    LET(sigma, j) = GET(lambda, j) * (SumVi - GET(Vprev, 0)) / Sum
  }
// Discretization equation for the first component of the vector X
LET(Vvalue_t, 0) = GET(Vprev, 0) * exp(-0.5 * pnl_vect_scalar_prod

pnl_vect_set_double(ci, 0.0);

/// For the rest of the components of vector 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,
          }
      }

    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) * (SumVi - GET(Vprev, i)) /
      }

    // Discretization equation :
    LET(Vvalue_t, i) = GET(Vprev, i) * exp(-0.5 * pnl_vect_scalar_

  }
```

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

    // If save_all_paths=1, we store the simulated value of libors at each
    // Transform the vector V into Libor values and store then in the matr
    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 -
            SumVi += GET(Vvalue_t, i - 1);
          }
        MLET(LiborPathsMatrix, l + m * Nsteps, 0) = GET(ptLOld->libor, 0);

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

  }

// If save_all_paths=0, we store only the simulated value of libors at the
// Transform the vector V into Libor values and store then in the matrix L
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 * Sum
        SumVi += GET(Vvalue_t, i - 1);
      }
    MLET(LiborPathsMatrix, m, 0) = GET(ptLOld->libor, 0); // L(t, T(0), T(

    // Store the value brownian motion used in the simulation.
    if (save_brownian == 1)
      {
        pnl_mat_set_row(BrownianMatrixPaths, ptW_sum, 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 *ptSwpt, NumFunc_1 *p)
{
  int j, alpha, beta;
  double SumDi, tenor, Di;
  double swaption_maturity, swap_maturity, swaption_strike, payoff;

  tenor = ptL->tenor;

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

  alpha = pnl_iround(swaption_maturity / tenor); // T_alpha is the swaption matu
  beta  = pnl_iround(swap_maturity / tenor); // T_beta is the 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
{
  int i, j, k, l, e, alpha, beta, n, Nfac, Nstep_integration,  payer_or_receiver
  double t, swaption_maturity, swap_maturity, swaption_strike, tenor, swap_rate,
  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 = pnl_iround(valuation_date / tenor);
  alpha = pnl_iround(swaption_maturity / tenor); // index of swaption_maturity
  beta  = pnl_iround(swap_maturity / tenor); // index of swap_maturity
  n = beta - alpha; // Nbr of payements dates
  Nfac = ptVol->numberOfFactors; // Nbr of factors in diffusion 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 weig

// swap_rate(0) = sum over i of weight(i)*LiborRate(0,Ti,Ti+1) , see Brigo&Mer
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 compute the integral of vola
dt = (swaption_maturity - valuation_date) / Nstep_integration; // step for the
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 volatilit
          {
            // We use the simple trapezoidal rule
            integrale = evalVolatility(ptVol, k, valuation_date, Ti) * evalVol
            integrale += evalVolatility(ptVol, k, swaption_maturity, Ti) * eva
            integrale *= 0.5;

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

```
            }

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

  vol_swaption = vol_swaption / SQR(swap_rate) ;

  black_volatility = sqrt(vol_swaption);

  d1 = (log(swap_rate / swaption_strike)) / black_volatility + 0.5 * black_volat
  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 *
    }

  else  // if (payer_or_receiver==0) Case of Receiver Swaption
    {
      price = sum_discount_factor * (swap_rate * (cdf_nor(d1) - 1) - swaption_st
    }

  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;
    }
}
```