Help

```
/* Monte Carlo Simulation for Lookback option on minimum:
   Put Fixed Euro and Call Floating Euro.
   The program provides estimations for Price and Delta with
   a confidence interval (for MC only)  */

#include  "bs1d_pad.h"
#include "enums.h"


static double inverse_min(double s1, double s2, double h, double sigma, double u
{
  return ((s1 + s2) - sqrt(SQR(s1 - s2) - 2 * SQR(sigma) * h * log(1. - un))) /
}



static int  LookBackInf_AndersenMontecarlo(double s, double pad, double strike,

{
  long i;
  double gs, un, min_log_norm, log_pad, log_s;
  int init_mc;
  int simulation_dim;
  double forward, forward_stock, exp_sigmaxwt, S_T, S_min, sigma_sqrt;
  double price_sample = 0., delta_sample = 0., mean_price, mean_delta, var_price
  PnlVect *U = pnl_vect_create(0);
  double alpha, z_alpha;

  /* Value to construct the confidence interval */
  alpha = (1. - confidence) / 2.;
  z_alpha = pnl_inv_cdfnor(1. - alpha);

  /*Initialisation*/
  mean_price = 0.0;
  mean_delta = 0.0;
  var_price = 0.0;
  var_delta = 0.0;
```

```
/* Size of the random vector we need in the simulation */
simulation_dim = 2;

/*Median forward stock and delta values*/
sigma_sqrt = sigma * sqrt(t);
forward = exp(((r - divid) - SQR(sigma) / 2.0) * t);
forward_stock = s * forward;
log_s = log(s);
log_pad = log(pad);

/*MonteCarlo sampling*/
init_mc = pnl_rand_init(generator, simulation_dim, N);

/* Test after initialization for the generator */
if (init_mc != OK)
  return init_mc;

/* Initialization of the model just allows to use Monte Carlo method */

/* We test if simulation is MC or PNL_QMC.
   This involves two parts in the program because simulation for random vector
   must be called from different functions */

for (i = 1; i <= N; i++)
  /* Begin N iterations */
  {
    /* For MC simulation, generation of two independent variables,
       a gaussian one and a uniform one, can be realized with the
       same preudo random number generator without problem of independence*/
    pnl_vect_rand_uni_d(U, 2, 0, 1, generator);
    gs = pnl_inv_cdfnor(pnl_vect_get(U, 0));
    un = pnl_vect_get(U, 1);

    exp_sigmaxwt = exp(sigma_sqrt * gs);
    S_T = forward_stock * exp_sigmaxwt;

    min_log_norm = inverse_min(log_s, log(S_T), t, sigma, un);
    S_min = exp(MIN(log_pad, min_log_norm));

    /* Price and Delta */
    /* PutFixedEuro */
```

```
      if (p->Compute == &Put_OverSpot2)
        {
          price_sample = (p->Compute)(p->Par, strike, S_min);
          delta_sample = 0.;
          if (price_sample > 0.)
            {
              if (pad == s)
                delta_sample = -S_min / s;
              else
                {
                  if (log_pad < min_log_norm)
                    delta_sample = 0.;
                  else  delta_sample = -S_min / s;
                }
            }
        }
      else
        /* CallFloatingEuro */
        if (p->Compute == &Call_StrikeSpot2)
          {
            price_sample = (p->Compute)(p->Par, S_T, S_min);
            if (pad == s)
              delta_sample = price_sample / s;
            else
              {
                if (log_pad < min_log_norm)
                  delta_sample = S_T / s;
                else  delta_sample = price_sample / s;
              }
          }

      /*Sum*/
      mean_price += price_sample;
      mean_delta += delta_sample;

      /*Sum of squares*/
      var_price += SQR(price_sample);
      var_delta += SQR(delta_sample);
    }
  /* End N iterations */
```

```
  /* errors are meaningless if PNL_QMC, but computed anyway to factorize code.
     DO NOT take them into account */

  /*Price*/
  *ptprice = exp(-r * t) * (mean_price / (double) N);
  *pterror_price = sqrt(exp(-2.0 * r * t) * var_price / (double)N - SQR(*ptprice
  /* Price Confidence Interval */
  *inf_price = *ptprice - z_alpha * (*pterror_price);
  *sup_price = *ptprice + z_alpha * (*pterror_price);

  /*Delta*/
  *ptdelta = exp(-r * t) * mean_delta / (double) N;
  *pterror_delta = sqrt(exp(-2.0 * r * t) * (var_delta / (double)N - SQR(*ptdelt
  /* Delta Confidence Interval */
  *inf_delta = *ptdelta - z_alpha * (*pterror_delta);
  *sup_delta = *ptdelta + z_alpha * (*pterror_delta);


  return OK;
}



int CALC(MC_LookBackMin_Andersen)(void *Opt, void *Mod, PricingMethod *Met)
{
  TYPEOPT *ptOpt = (TYPEOPT *)Opt;
  TYPEMOD *ptMod = (TYPEMOD *)Mod;
  double r, divid;

  r = log(1. + ptMod->R.Val.V_DOUBLE / 100.);
  divid = log(1. + ptMod->Divid.Val.V_DOUBLE / 100.);

  return LookBackInf_AndersenMontecarlo(ptMod->S0.Val.V_PDOUBLE,
                                        (ptOpt->PathDep.Val.V_NUMFUNC_2)->Par[4]
                                        (ptOpt->PayOff.Val.V_NUMFUNC_2)->Par[0].
                                        ptOpt->PayOff.Val.V_NUMFUNC_2,
                                        ptOpt->Maturity.Val.V_DATE - ptMod->T.Va
                                        r,
                                        divid,
                                        ptMod->Sigma.Val.V_PDOUBLE,
                                        Met->Par[0].Val.V_LONG,
```

```
                                        Met->Par[1].Val.V_ENUM.value,
                                        Met->Par[2].Val.V_DOUBLE,
                                        &(Met->Res[0].Val.V_DOUBLE),
                                        &(Met->Res[1].Val.V_DOUBLE),
                                        &(Met->Res[2].Val.V_DOUBLE),
                                        &(Met->Res[3].Val.V_DOUBLE),
                                        &(Met->Res[4].Val.V_DOUBLE),
                                        &(Met->Res[5].Val.V_DOUBLE),
                                        &(Met->Res[6].Val.V_DOUBLE),
                                        &(Met->Res[7].Val.V_DOUBLE));
}



static int CHK_OPT(MC_LookBackMin_Andersen)(void *Opt, void *Mod)
{
  if ((strcmp(((Option *)Opt)->Name, "LookBackPutFixedEuro") == 0) || (strcmp(((
    return OK;
  return WRONG;
}



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

      Met->Par[0].Val.V_LONG = 10000;
      Met->Par[1].Val.V_ENUM.value = 0;
      Met->Par[1].Val.V_ENUM.members = &PremiaEnumRNGs;
      Met->Par[2].Val.V_DOUBLE = 0.95;
    }

  type_generator = Met->Par[1].Val.V_ENUM.value;

  if (pnl_rand_or_quasi(type_generator) == PNL_QMC)
    {
```

```
      Met->Res[2].Viter = IRRELEVANT;
      Met->Res[3].Viter = IRRELEVANT;
      Met->Res[4].Viter = IRRELEVANT;
      Met->Res[5].Viter = IRRELEVANT;
      Met->Res[6].Viter = IRRELEVANT;
      Met->Res[7].Viter = IRRELEVANT;

    }
  else
    {
      Met->Res[2].Viter = ALLOW;
      Met->Res[3].Viter = ALLOW;
      Met->Res[4].Viter = ALLOW;
      Met->Res[5].Viter = ALLOW;
      Met->Res[6].Viter = ALLOW;
      Met->Res[7].Viter = ALLOW;
    }

  return OK;
}



PricingMethod MET(MC_LookBackMin_Andersen) =
{
  "MC_LookBackMin_Andersen",
  { {"N iterations", LONG, {100}, ALLOW},
    {"RandomGenerator", ENUM, {100}, ALLOW},
    {"Confidence Value", DOUBLE, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_LookBackMin_Andersen),
  { {"Price", DOUBLE, {100}, FORBID},
    {"Delta", DOUBLE, {100}, FORBID} ,
    {"ErrorPrice", DOUBLE, {100}, FORBID},
    {"ErrorDelta", DOUBLE, {100}, FORBID} ,
    {"Inf Price", DOUBLE, {100}, FORBID},
    {"Sup Price", DOUBLE, {100}, FORBID} ,
    {"Inf Delta", DOUBLE, {100}, FORBID},
    {"Sup Delta", DOUBLE, {100}, FORBID} ,
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
```

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