Help

```
/* Monte Carlo Simulation for Parisian option :
   The program provides estimations for Price and Delta with
   a confidence interval.  */
/* Quasi Monte Carlo simulation is not yet allowed for this routine */

#include "bs1d_lim.h"
#include "enums.h"

static int check_parisianout(double *gt, double lnspot, double lastlnspot,
                             double barrier, double lastbarrier,
                             double *gt_increment,
                             double lnspot_increment, double lastlnspot_incremen
                             double rap, int upordown, double h, double time,
                             int *correction_active, int generator)
{
  double proba, uniform = 0.;

  if (((upordown == 0) && (lnspot < barrier)) || ((upordown == 1) && (lnspot > b
    {
      if (((lastlnspot > barrier) && (upordown == 1)) || ((lastlnspot < barrier)
        {
          proba = exp(-2.*rap * ((lastlnspot - lastbarrier) * (lnspot - lastbarr
          *correction_active = 1;
          uniform = pnl_rand_uni(generator);
          if (uniform < proba)
            *gt = time;
        }
      else *gt = (time - h) + (barrier - lastlnspot) / (lnspot - lastlnspot) * h
    }
  else *gt = time;

  if (((upordown == 0) && (lnspot_increment < barrier)) || ((upordown == 1) && (
    {
      if (((lastlnspot_increment > barrier) && (upordown == 1)) || ((lastlnspot_
        {
          proba = exp(-2.*rap * ((lastlnspot_increment - lastbarrier) * (lnspot_
          if (!*correction_active)
            uniform = pnl_rand_uni(generator);
          if (uniform < proba)
```

```
            *gt_increment = time;
        }
      else *gt_increment = (time - h) + (barrier - lastlnspot_increment) / (lnsp
    }
  else *gt_increment = time;

  return OK;

}


static int MC_ParisianOut(int upordown, double s, NumFunc_1 *PayOff, double l, d
                          double delay, double r, double divid, double sigma, in
                          long M, int N, double increment, double confidence, do
                          double *ptdelta, double *pterror_price, double *pterro
                          double *inf_price, double *sup_price, double *inf_delt
{
  double g, h;
  double time, lnspot, lastlnspot, price_sample = 0., delta_sample, lns;
  double lnspot_increment = 0., lastlnspot_increment, price_sample_increment = 0
  double rloc, sigmaloc, barrier, lastbarrier, rap;
  double gt, hd, gt_increment, hd_increment;
  double mean_price, var_price, mean_delta, var_delta;
  long i;
  int k, inside, inside_increment;
  int correction_active;
  int init_mc;
  int simulation_dim;
  double alpha, z_alpha;

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

  /*One forces N if necessary so that delay
    !!!!!!!!!! WARNING !!!!!!!!
    be greater than the time step increment h*/
  h = t / (double)N;
  if (delay <= h)
    {
      N = (int)ceil(t / delay) + 1;
```

```
    h = t / (double)N;

    Fprintf(TOSCREEN, "WARNING!!! N is forced to %d\ n", N);
  }

/*Initialisation*/
mean_price = 0.0;
mean_delta = 0.0;
var_price = 0.0;
var_delta = 0.0;
/* Maximum Size of the random vector we need in the simulation */
simulation_dim = N;

barrier = log(l);
lns = log(s);

rloc = (r - divid - SQR(sigma) / 2.) * h;
sigmaloc = sigma * sqrt(h);

/*Coefficient for the computation of the exit probability*/
rap = 1. / (sigmaloc * sigmaloc);

/*MC sampling*/
init_mc = pnl_rand_init(generator, simulation_dim, M);
/* Test after initialization for the generator */
if (init_mc == OK)
  {


    /* Begin M iterations */
    for (i = 1; i <= M; i++)
      {
        gt = 0.;
        hd = 0.;
        gt_increment = 0.;
        hd_increment = 0.;
        lnspot = lns;

        /*Inside=0 if the path stays beyond the barrier uninterruptedly
          for longer than delay*/
        inside = 1;
```

```
inside_increment = 1;

k = 0;
time = 0.;

/*Barrier at time*/
barrier = log(l);

/*Simulation of i-th path until Inside=0*/
while (((inside) && (k < N)) || ((inside_increment) && (k < N)))
  {
    correction_active = 0;

    lastlnspot = lnspot;
    lastbarrier = barrier;

    time += h;
    g = pnl_rand_normal(generator);
    lnspot += rloc + sigmaloc * g;

    lnspot_increment = lnspot + increment;
    lastlnspot_increment = lastlnspot + increment;

    barrier = log(l);

    /*Check if the i-th path has reached the barrier at time*/
    if (upordown == 0)
      check_parisianout(&gt, lnspot, lastlnspot, barrier, lastbarrier,
    else
      check_parisianout(&gt_increment, lnspot_increment, lastlnspot_in

    hd = time - gt;
    hd_increment = time - gt_increment;

    if (hd > delay)
      {
        inside = 0;
        price_sample = 0.;
      }

    if (hd_increment > delay)
```

```
          {
            inside_increment = 0;
            price_sample_increment = 0.;
          }

        k++;
      }

    if (inside)
      price_sample = exp(-r * t) * (PayOff->Compute)(PayOff->Par, exp(lnsp
    if (inside_increment)
      price_sample_increment = exp(-r * t) * (PayOff->Compute)(PayOff->Par

    /*Delta*/
    delta_sample = (price_sample_increment - price_sample) / (increment *

    /*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 */

/*Price*/
*ptprice = mean_price / (double)M;
*pterror_price = sqrt(var_price / (double)M - SQR(*ptprice)) / sqrt(M - 1)

/*Delta*/
*ptdelta = mean_delta / (double) M;
*pterror_delta = sqrt(var_delta / (double)M - SQR(*ptdelta)) / sqrt((doubl

/* Price Confidence Interval */
*inf_price = *ptprice - z_alpha * (*pterror_price);
*sup_price = *ptprice + z_alpha * (*pterror_price);

/* Delta Confidence Interval */
*inf_delta = *ptdelta - z_alpha * (*pterror_delta);
*sup_delta = *ptdelta + z_alpha * (*pterror_delta);
```

```
    }
  return init_mc;
}




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

  r = log(1. + ptMod->R.Val.V_DOUBLE / 100.);
  divid = log(1. + ptMod->Divid.Val.V_DOUBLE / 100.);
  limit = ((ptOpt->Limit.Val.V_NUMFUNC_1)->Compute)((ptOpt->Limit.Val.V_NUMFUNC_

  if ((ptOpt->DownOrUp).Val.V_BOOL == DOWN)
    upordown = 0;
  else upordown = 1;

  return MC_ParisianOut(upordown,
                        ptMod->S0.Val.V_PDOUBLE,
                        ptOpt->PayOff.Val.V_NUMFUNC_1,
                        limit,
                        ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
                        (ptOpt->Limit.Val.V_NUMFUNC_1)->Par[4].Val.V_PDOUBLE,
                        r,
                        divid,
                        ptMod->Sigma.Val.V_PDOUBLE,
                        Met->Par[1].Val.V_ENUM.value,
                        Met->Par[0].Val.V_LONG,
                        Met->Par[2].Val.V_INT,
                        Met->Par[3].Val.V_PDOUBLE,
                        Met->Par[4].Val.V_PDOUBLE,
                        &(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_ParisianOut)(void *Opt, void *Mod)
{
  Option *ptOpt = (Option *)Opt;
  TYPEOPT *opt = (TYPEOPT *)(ptOpt->TypeOpt);

  if ((opt->RebOrNo).Val.V_BOOL == NOREBATE)
    if ((opt->OutOrIn).Val.V_BOOL == OUT)
      if ((opt->EuOrAm).Val.V_BOOL == EURO)
        if ((opt->Parisian).Val.V_BOOL == TRUE)

          return OK;

  return  WRONG;
}


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

      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_INT2 = 250;
      Met->Par[3].Val.V_PDOUBLE = 0.01;
      Met->Par[4].Val.V_PDOUBLE = 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_ParisianOut) =
{
  "MC_Parisianout",
  { {"Iterations", LONG, {100}, ALLOW},
    {"RandomGenerator (Quasi Random not supported)", ENUM, {100}, ALLOW},
    {"TimeStepNumber", INT2, {100}, ALLOW},
    {"Delta Increment Rel", DOUBLE, {100}, ALLOW},
    {"Confidence Value", DOUBLE, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_ParisianOut),
  { {"Price", DOUBLE, {100}, FORBID},
    {"Delta", DOUBLE, {100}, FORBID} ,
    {"Error Price", DOUBLE, {100}, FORBID},
    {"Error Delta", 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_ParisianOut),
  CHK_mc,
  MET(Init)
} ;
```