Help

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

#define WITH_boundary 1
#include "
href../../../mod/bs1d/bs1d_lim/bs1d_lim_h_src.pdfbs1d_lim.h"
#include "
href../../../common/enums_h_src.pdfenums.h"

/* Check if the spot has crossed the barrier during the time interval */
static int check_barrierin(int *inside, double lnspot, double lastlnspot,
                           double barrier, double lastbarrier,
                           int *inside_increment,
                           double lnspot_increment, double lastlnspot_increment,
                           double rap, double time,
                           int *correction_active,
                           int generator,
                           double *exit_time, double *exit_time_increment)
{
  double proba = 0., uniform = 0.;

  if (*inside)
    {
      proba = exp(-2.*rap * ((lastlnspot - lastbarrier) * (lnspot - lastbarrier)
      /* Simulation of an uniform variable */
      uniform = pnl_rand_uni(generator);

      *correction_active = 1;
      if (uniform < proba)
        {
          *inside = 0;
          *exit_time = time;
        }
    }
  if (*inside_increment)
    {
```

```
      proba = exp(-2.*rap * ((lastlnspot_increment - lastbarrier) * (lnspot_incr
      if (!*correction_active)
        /* Simulation of an uniform variable */
        uniform = pnl_rand_uni(generator);

      if (uniform < proba)
        {
          *inside_increment = 0;
          *exit_time_increment = time;
        }
    }
  return OK;
}

static int MC_InBaldi_97(int upordown, double s, NumFunc_1 *PayOff, double l, do
{
  double h = t / (double)M;
  double time, lnspot, lastlnspot, price_sample, delta_sample, exit_time;
  double lnspot_increment = 0., lastlnspot_increment, price_sample_increment, ex
  double rloc, sigmaloc,  barrier, lastbarrier, rap, g;
  double mean_price, var_price, mean_delta, var_delta;
  long i;
  int k, inside, inside_increment, 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);

  /*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 = M;

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

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

/*MonteCarlo sampling*/
init_mc = pnl_rand_init(generator, simulation_dim, Nb);
if (init_mc == OK)
  {


    /* Begin N iterations */
    for (i = 1; i <= Nb; i++)
      {
        time = 0.;
        lnspot = log(s);

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

        /*Inside=0 if the path reaches the barrier*/
        inside = 1;
        inside_increment = 1;
        k = 0;

        /*Simulation of i-th path until its exit if it does*/
        while (((inside) && (k < M)) || ((inside_increment) && (k < M)))
          {
            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 (inside)
      if (((upordown == 0) && (lnspot < barrier)) || ((upordown == 1)
        {
          inside = 0;
          exit_time = time;
        }

    if (inside_increment)
      if (((upordown == 0) && (lnspot_increment < barrier)) || ((upord
        {
          inside_increment = 0;
          exit_time_increment = time;
        }

    /*Check if the i-th path has reached the barrier during (time-1,ti
    if (upordown == 0)
      check_barrierin(&inside, lnspot, lastlnspot, barrier, lastbarrie
                      &inside_increment, lnspot_increment, lastlnspot_
                      &correction_active, generator, &exit_time, &exit
    else
      check_barrierin(&inside_increment, lnspot_increment, lastlnspot_
                      lastbarrier, &inside, lnspot, lastlnspot, rap, t

    k++;
  }
/*Inside=0 means that the payoff does not nullify
  Inside=1 means that the payoff is equal to the rebate*/

if (inside == 0)
  {
    if (t - exit_time > 0)
      price_sample = exp(-r * exit_time) * Boundary(l, PayOff, t - exi
    else
      price_sample = exp(-r * t) * (PayOff->Compute)(PayOff->Par, l);
  }
else
  price_sample = exp(-r * t) * rebate;

if (inside_increment == 0)
  {
```

```
              if (t - exit_time_increment > 0)
                price_sample_increment = exp(-r * exit_time_increment) * Boundar
              else
                price_sample_increment = exp(-r * t) * (PayOff->Compute)(PayOff-
            }
          else
            price_sample_increment = exp(-r * t) * rebate;

          /*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)Nb;
      *pterror_price = sqrt(var_price / (double)Nb - SQR(*ptprice)) / sqrt(Nb -

      /*Delta*/
      *ptdelta = mean_delta / (double) Nb;
      *pterror_delta = sqrt(var_delta / (double)Nb - SQR(*ptdelta)) / sqrt((doub

      /* 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_InBaldi)(void *Opt, void *Mod, PricingMethod *Met)
{
  TYPEOPT *ptOpt = (TYPEOPT *)Opt;
  TYPEMOD *ptMod = (TYPEMOD *)Mod;
  double r, divid, limit, rebate;
  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_
  rebate = ((ptOpt->Rebate.Val.V_NUMFUNC_1)->Compute)((ptOpt->Rebate.Val.V_NUMFU

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

  return  MC_InBaldi_97(upordown,
                      ptMod->S0.Val.V_PDOUBLE,
                      ptOpt->PayOff.Val.V_NUMFUNC_1,
                      limit,
                      rebate,
                      ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
                      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_InBaldi)(void *Opt, void *Mod)
{
  Option *ptOpt = (Option *)Opt;
  TYPEOPT *opt = (TYPEOPT *)(ptOpt->TypeOpt);


  if ((opt->OutOrIn).Val.V_BOOL == IN)
    if ((opt->EuOrAm).Val.V_BOOL == EURO)
      if ((opt->Parisian).Val.V_BOOL == FALSE)
        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 = &PremiaEnumMCRNGs;
      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_InBaldi) =
{
  "MC_Baldi_In",
  { {"N iterations", LONG, {100}, ALLOW},
    {"RandomGenerator", ENUM, {100}, ALLOW},
    {"TimeStepNumber M", INT2, {100}, ALLOW},
    {"Delta Increment Rel", PDOUBLE, {100}, ALLOW},
    {"Confidence Value", DOUBLE, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_InBaldi),
  { {"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_InBaldi),
  CHK_mc,
  MET(Init)
} ;
```