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 */

#include <stdlib.h>
#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_barrierout(int *inside, double lnspot, double lastlnspot,
                            double barrier, double lastbarrier,
                            int *inside_increment,
                            double lnspot_increment, double lastlnspot_increment
                            double rap, double r, double time,
                            int *correction_active,
                            double rebate, int generator,
                            double *price_sample, double *price_sample_increment
{
  double proba = 0., uniform = 0.;
  if (*inside)
    {
      proba = exp(-2.*rap * ((lastlnspot - lastbarrier) * (lnspot - lastbarrier)
      uniform = pnl_rand_uni(generator);
      *correction_active = 1;
      if (uniform < proba)
        {
          *inside = 0;
          *price_sample = exp(-r * time) * rebate;
        }
    }
  if (*inside_increment)
    {
      proba = exp(-2.*rap * ((lastlnspot_increment - lastbarrier) * (lnspot_incr
      if (!*correction_active)
        uniform = pnl_rand_uni(generator);
```

```
      if (uniform < proba)
        {
          *inside_increment = 0;
          *price_sample_increment = exp(-r * time) * rebate;
        }
    }
  return OK;
}

double regul(double x)
{
  if (x <= -1.)
    return 0.0;
  else
    {
      if ((x > -1.) && (x <= 0))
        return (x + 1.) * exp(-1. / (x * x * (x - 1.) * (x - 1.)));
      else
        return 1.0;
    }
}
double der_regul(double x)
{
  if ((x <= -1) || (x >= 0))
    return 0.0;
  else
    return (1. + 2.*x * ((2.*x - 1) / (x * x * x * (x - 1.) * (x - 1.) * (x - 1.

}
static int MC_OutBaldi_97(int upordown, double s, NumFunc_1 *PayOff, double l, d
{
  double h = t / (double)M;
  double temps, lnspot, lastlnspot, lnspot_increment = 0., lastlnspot_increment,
  double rloc, sigmaloc, barrier, lastbarrier, rap, g;
  double mean_price, var_price, mean_delta, var_delta;
  long i;
  int k, inside, inside_increment, j;
  int correction_active;
  int init_mc;
  int simulation_dim;
```

```
double alpha, z_alpha, a, maxlnspot, minlnspot, temp, intder, intsto, intreg;
double *tauM, *taum, *domprocess;
tauM = malloc(sizeof(double) * (M + 1));
taum = malloc(sizeof(double) * (M + 1));
domprocess = malloc(sizeof(double) * (M + 1));

/* 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;

barrier = log(l);
lns = log(s);
a = l - 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, Nb);
/* Test after initialization for the generator */

if (init_mc == OK)
  {

    /* Begin N iterations */
    for (i = 1; i <= Nb; i++)
      {
        temps = 0.;
        lnspot = lns;
        intsto = 0.0;
        intreg = 0.0;
```

```
intder = 0.0;
taum[0] = 0.0;
tauM[0] = 0.0;
/*Barrier at time*/
barrier = log(l);
maxlnspot = lns;
domprocess[0] = 0.0;
minlnspot = lns;
/*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  || inside_increment) && (k < M))
  {
    correction_active = 0;

    lastlnspot = lnspot;
    lastbarrier = barrier;

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

    /* Tools for computation of Malliavin Weights*/
    if (delta_met > 1)
      {
        if (lnspot > maxlnspot)
          {
            tauM[k + 1] = temps;
            domprocess[k + 1] = domprocess[k] - maxlnspot;
            maxlnspot = lnspot;
            domprocess[k + 1] += maxlnspot;
          }
        else
          tauM[k + 1] = tauM[k];
        if (lnspot < minlnspot)
          {
            taum[k + 1] = temps;
```

```
        domprocess[k + 1] = domprocess[k] + minlnspot;
        minlnspot = lnspot;
        domprocess[k + 1] -= minlnspot;
      }
    else
      taum[k + 1] = taum[k];

    intsto += regul((a - 2.*exp(domprocess[k])) / a) * sqrt(h) * g
    intreg += regul((a - 2.*exp(domprocess[k])) / a) * h;
    temp = 0.0;
    for (j = 0; j <= k; j++)
      {
        if ((j * h <= tauM[k]) && (j * h >= taum[k]))
          temp += regul((a - 2.*exp(domprocess[k])) / a);
        if ((j * h >= tauM[k]) && (j * h <= taum[k]))
          temp += regul((a - 2.*exp(domprocess[k])) / a);
      }

    intder += der_regul((a - 2.*exp(domprocess[k])) / a) * temp *
  }

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;
      price_sample = exp(-r * temps) * rebate;
    }

if (inside_increment)
  if (((upordown == 0) && (lnspot_increment < barrier)) || ((upord
    {
      inside_increment = 0;
      price_sample_increment = exp(-r * temps) * rebate;
    }
```

```
    /*Check if the i-th path has reached the barrier during (temps-1,t
    if (upordown == 0)
      check_barrierout(&inside, lnspot, lastlnspot, barrier, lastbarri
                       &inside_increment, lnspot_increment, lastlnspot
                       rap, r, temps, &correction_active, rebate, gene
                       &price_sample, &price_sample_increment);
    else
      check_barrierout(&inside_increment, lnspot_increment, lastlnspot
                       barrier, lastbarrier, &inside, lnspot, lastlnsp
                       temps, &correction_active, rebate, generator,
                       &price_sample_increment, &price_sample);

    k++;
  }/*while*/

if (inside)
  {
    price_sample = exp(-r * t) * (PayOff->Compute)(PayOff->Par, exp(ln
  }

if (inside_increment)
  {
    price_sample_increment = exp(-r * t) * (PayOff->Compute)(PayOff->P
  }

/*Delta*/
if (delta_met == 1)
  delta_sample = (price_sample_increment - price_sample) / (increment
else
  {
    if (!inside)
      delta_sample = sigma * 10 * exp(-r * t) * price_sample * (intsto
    else
      delta_sample = 0.0;
    /*printf("%lf %lf %lf %lf %lf\ n",delta_sample,price_sample,intsto
    delta_sample = (price_sample_increment - price_sample) / (incremen
  }
/*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);
    }

  free(tauM);
  free(taum);
  free(domprocess);

  return init_mc;
}
```

```
int CALC(MC_OutBaldi)(void *Opt, void *Mod, PricingMethod *Met)
{
  TYPEOPT *ptOpt = (TYPEOPT *)Opt;
  TYPEMOD *ptMod = (TYPEMOD *)Mod;
  double r, divid, limit, rebate; /* increment=0.01; */
  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_OutBaldi_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->Par[5].Val.V_ENUM.value,
                        &(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_OutBaldi)(void *Opt, void *Mod)
{
  Option *ptOpt = (Option *)Opt;
  TYPEOPT *opt = (TYPEOPT *)(ptOpt->TypeOpt);

  if ((opt->OutOrIn).Val.V_BOOL == OUT)
```

```
    if ((opt->EuOrAm).Val.V_BOOL == EURO)
      if ((opt->Parisian).Val.V_BOOL == FALSE)

        return OK;

  return  WRONG;
}

static PremiaEnumMember DeltaMethodBaldiMembers[] =
{
  { "Finite Difference", 1 },
  { "Malliavin", 2 },
  { NULL, NULLINT }
};

static DEFINE_ENUM(DeltaMethodBaldi, DeltaMethodBaldiMembers)


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;
      Met->Par[5].Val.V_ENUM.value = 1;
      Met->Par[5].Val.V_ENUM.members = &DeltaMethodBaldi;

    }

  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_OutBaldi) =
{
  "MC_Baldi_Out",
  { {"N iterations", LONG, {100}, ALLOW},
    {"RandomGenerator", ENUM, {100}, ALLOW},
    {"TimeStepNumber M", INT2, {100}, ALLOW},
    {"Delta Increment Rel", DOUBLE, {100}, ALLOW},
    {"Confidence Value", DOUBLE, {100}, ALLOW},
    {"Delta Method", ENUM, {1}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_OutBaldi),
  { {"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_OutBaldi),
  CHK_mc,
  MET(Init)
} ;
```