Help

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

#include "bs1d_doublim.h"
#include "enums.h"

static int MC_ParisianIn(NumFunc_1  *L,NumFunc_1  *U,
    double s,NumFunc_1   *PayOff,double t,double delay,double r,
    double divid,double sigma,int generator,long M,int N,double inc
    rement,double confidence,double *ptprice,double *ptdelta,
    double *pterror_price,double *pterror_delta,double *inf_price,
    double *sup_price, double *inf_delta,double *sup_delta)
{
  double g, h;
  double time,lnspot,lastlnspot,price_sample=0.,delta_sampl
    e;
  double lnspot_increment,lastlnspot_increment,price_sampl
    e_increment=0.;
  double rloc,sigmaloc,up,low,lastup,lastlow,proba=0.,rap,
    gt,hd;
  double gt_increment,hd_increment;
  double mean_price,var_price,mean_delta,var_delta;
  double uniform=0.,proba_increment;
  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);

  /*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 simu
  lation */
simulation_dim= N;

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=log(s);
```

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

time=0.;
k=0;

/*Up and Down Barrier at time*/
up=log((U->Compute)(U->Par,time));
low=log((L->Compute)(L->Par,time));

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

    lastlnspot=lnspot;
    lastup=up;
    lastlow=low;

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

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

    up=log((U->Compute)(U->Par,time));
    low=log((L->Compute)(L->Par,time));

    /*Check if the i-th path has reached the barriers
at time*/
    /*Otherwise there is no extinction*/

    if (inside)
if (lnspot>up)
  {
```

```
    if (lastlnspot>up)
      {
  proba=exp(-2.*rap*((lastlnspot-lastup)*(lnspot-
lastup)-(lastlnspot-lastup)*(up-lastup)));
  correction_active=1;
  uniform=pnl_rand_uni(generator);
  if (uniform<proba)
    gt=time;
      }
    else gt=(time-h)+(up-lastlnspot)/(lnspot-lastlns
pot)*h;
  }

    if (inside_increment)
if (lnspot_increment>up)
  {
    if (lastlnspot_increment>up)
      {
  proba_increment=exp(-2.*rap*((lastlnspot_increment
-lastup)*(lnspot_increment-lastup)-(lastlnspot_increment-
lastup)*(up-lastup)));
  if (!correction_active)
    uniform=pnl_rand_uni(generator);
  if (uniform<proba)
    gt_increment=time;
      }
    else gt_increment=(time-h)+(up-lastlnspot_increm
ent)/(lnspot_increment-lastlnspot_increment)*h;
  }

    if (inside_increment)
if (lnspot_increment<low)
  {
    if (lastlnspot_increment<low)
      {
  proba_increment=exp(-2.*rap*((lastlnspot_increment
-lastlow)*(lnspot_increment-lastlow)+(lastlnspot_increment
-lastlow)*(low-lastlow)));
  correction_active=1;
  uniform=pnl_rand_uni(generator);
  if (uniform<proba_increment)
```

```
    gt_increment=time;
      }
    else gt_increment=(time-h)+(low-lastlnspot_increm
ent)/(lnspot_increment-lastlnspot_increment)*h;
  }

    if (inside)
if (lnspot <low)
  {
    if (lastlnspot <low)
      {
  proba =exp(-2.*rap*((lastlnspot -lastlow)*(lnspot
-lastlow)+(lastlnspot -lastlow)*(low-lastlow)));
  if (!correction_active)
    uniform=pnl_rand_uni(generator);
  if (uniform<proba)
    gt =time;
      }
    else gt =(time-h)+(low-lastlnspot )/(lnspot -
lastlnspot)*h;
  }

    if (inside) {
if ((lnspot<=up)&&(lnspot>=low))
  gt=time;
hd=time-gt;
if(hd>delay)
  {
    inside=0;
    if(t-time<0)
      time=t;
    price_sample=exp(-r*time)*Boundary(exp(lnspot),
PayOff,t-time,r,divid,sigma);
  }
    }

    if (inside_increment) {
if ((lnspot_increment<=up)&&(lnspot_increment>=low))
  gt_increment=time;

hd_increment=time-gt_increment;
```

```
    if(hd_increment>delay)
      {
        inside_increment=0;
        if(t-time<0)
          time=t;
        price_sample_increment=exp(-r*time)*Boundary(exp(
    lnspot_increment),PayOff,t-time,r,divid,sigma);
      }
        }

        k++;
      }
    if (inside)
      price_sample=0.;

    if (inside_increment)
      price_sample_increment=0.;

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

      /*Price*/
      *ptprice=mean_price/(double)M;
      *pterror_price= sqrt(var_price/(double)M - SQR(*pt
    price))/sqrt(M-1);
      /*Delta*/
      *ptdelta=mean_delta/(double) M;
      *pterror_delta= sqrt(var_delta/(double)M-SQR(*ptdelt
    a))/sqrt((double)M-1);
```

```
      /* 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_ParisianIn)(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 MC_ParisianIn(ptOpt->LowerLimit.Val.V_NUMFUNC_1,
           ptOpt->UpperLimit.Val.V_NUMFUNC_1,
           ptMod->S0.Val.V_PDOUBLE,
           ptOpt->PayOff.Val.V_NUMFUNC_1,
           ptOpt->Maturity.Val.V_DATE-ptMod->T.Val.V_DA
    TE,
           (ptOpt->LowerLimit.Val.V_NUMFUNC_1)->Par[1].
    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_ParisianIn)(void *Opt, void *Mod)
{
  Option* ptOpt=(Option*)Opt;
  TYPEOPT* opt=(TYPEOPT*)(ptOpt->TypeOpt);

  if ((opt->RebOrNo).Val.V_BOOL==NOREBATE)
    if ((opt->Parisian).Val.V_BOOL==OK)
      if (((opt->OutOrIn).Val.V_BOOL==IN)&&((opt->EuOrAm).
    Val.V_BOOL==EURO))
  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_ParisianIn)=
{
  "MC_ParisianIn",
  {{"Iterations",LONG,{100},ALLOW},
   {"RandomGenerator",ENUM,{100},ALLOW},
   {"TimeStepNumber",INT2,{100},ALLOW},
   {"Delta Increment Rel",PDOUBLE,{100},ALLOW},
   {"Confidence Value",PDOUBLE,{100},ALLOW},
   {" ",PREMIA_NULLTYPE,{0},FORBID}},
  CALC(MC_ParisianIn),
  {{"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_ParisianIn),
  CHK_mc,
  MET(Init)
} ;
```

References