```
   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 */
#define WITH_boundary 1
#include "bs1d_lim.h"
#include "enums.h"


static int check_parisianin(double *gt,double lnspot,
    double lastlnspot,
          double barrier, double lastbarrier,
          double *gt_increment,
          double lnspot_increment,double lastlnspot_inc
    rement,
          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>barrier)))
    {
      if (((lastlnspot>barrier)&&(upordown==1))||((lastlns
    pot<barrier)&&(upordown==0)))
  {
    proba=exp(-2.*rap*((lastlnspot-lastbarrier)*(lnspot-
    lastbarrier)-(lastlnspot-lastbarrier)*(barrier-lastbarrier)));
    *correction_active=1;
    uniform=pnl_rand_uni(generator);
    if (uniform<proba)
      *gt=time;
  }
      else *gt=(time-h)+(barrier-lastlnspot)/(lnspot-lastl
    nspot)*h;
    }
  else *gt=time;
```

```
  if (((upordown==0)&&(lnspot_increment<barrier))||((upor
    down==1)&&(lnspot_increment>barrier)))
    {
      if (((lastlnspot_increment>barrier)&&(upordown==1))||
    ((lastlnspot_increment<barrier)&&(upordown==0)))
  {
    proba=exp(-2.*rap*((lastlnspot_increment-lastbarrier)*
    (lnspot_increment-lastbarrier)-(lastlnspot_increment-lastb
    arrier)*(barrier-lastbarrier)));
    if (!*correction_active)
      uniform=pnl_rand_uni(generator);
    if (uniform<proba)
      *gt_increment=time;
  }
      else *gt_increment=(time-h)+(barrier-lastlnspot_inc
    rement)/(lnspot_increment-lastlnspot_increment)*h;
    }
  else *gt_increment=time;

  return OK;

}

static int MC_ParisianIn(int upordown,double s,NumFunc_1 *
    PayOff,double l,double t,double delay,double r,double divid,
    double sigma,int generator,long M,int N,double increment,
    double confidence,double *ptprice,double *ptdelta,double *pt
    error_price,double *pterror_delta,double *inf_price, double *su
    p_price, double *inf_delta, double *sup_delta)
{
  double g, h;
  double time,lnspot,lastlnspot,price_sample=0.,delta_sampl
    e,lns;
  double lnspot_increment,lastlnspot_increment,price_sampl
    e_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 simu
  lation */
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.;
  lnspot=lns;

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

  time=0;
  k=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*/
    /*Otherwise there is no extinction*/
    if (upordown==0)
check_parisianin(&gt,lnspot,lastlnspot,barrier,lastb
arrier,
           &gt_increment,lnspot_increment,lastlnspo
t_increment,
           rap,upordown,h,time,&correction_active,    generator);
    else
check_parisianin(&gt_increment,lnspot_increment,lastl
nspot_increment,barrier,lastbarrier,&gt,lnspot,lastlnspot,
rap,upordown,h,time,&correction_active,generator);

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

    if(hd>delay)
{
  inside=0;
  if (time>t) time=t;
  price_sample=exp(-r*time)*Boundary(exp(lnspot),PayO
ff,t-time,r,divid,sigma);
}

    if(hd_increment>delay)
{
  inside_increment=0;
  if (time>t) time=t;
  price_sample_increment=exp(-r*time)*Boundary(exp(ln
spot_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,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_NUMFUN

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

  return MC_ParisianIn(upordown,
           ptMod->S0.Val.V_PDOUBLE,ptOpt->PayOff.Val.V_
    NUMFUNC_1,
           limit,
           ptOpt->Maturity.Val.V_DATE-ptMod->T.Val.V_DA
    TE,
           (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_PDOUB
    LE,
           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->OutOrIn).Val.V_BOOL==IN)
      if ((opt->EuOrAm).Val.V_BOOL==EURO)
  if ((opt->Parisian).Val.V_BOOL==OK)

    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 (Quasi Random not allowed)",ENUM,{100}
    ,ALLOW},
   {"TimeStepNumber",INT2,{100},ALLOW},
   {"Delta Increment Rel",DOUBLE,{100},ALLOW},
   {"Confidence Value",DOUBLE,{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