```
   Help
/* Standard Monte Carlo simulation for a Call - Put -
    CallSpread or
   Digit option.
   In the case of Monte Carlo simulation, the program prov
    ides estimations for price and delta with a confidence
    interval.
   In the case of Quasi-Monte Carlo simulation, the program
     just provides estimations for price and delta.

   For a Call, the implementation is based on the Call-Put
    Parity
   relationship. */

#include  "bs1d_std.h"
#include "enums.h"
static double reg_put(double eps, double s,double H)
{
  if (s<=H-eps)
    return 1.;
  else{
    if ((s>H-eps)&&(s<=H+eps))
      return (-s+H+eps)/2*eps;
    else
      return 0.0;
  }
}
static double F_reg_put(double eps,double s, double H)
{
  if (s<=H-eps)
    return 0.0;
  else{
    if ((s>=H-eps)&& (s<H))
      return H-s - (SQR(-s+H+eps))/(4.*eps);
    else{
      if ((s>=H) && (s<H+eps))
  return 0.0 - (SQR(-s+H+eps))/(4.*eps);
      else
  return 0.0;
    }
  }
```

```
}
static double reg_call(double eps, double s,double H)
{
  if (s<=H-eps)
    return 0.;
  else{
    if ((s>H-eps)&&(s<=H+eps))
      return (s-H+eps)/2*eps;
    else
      return 1.0;
  }
}
static double F_reg_call(double eps,double s, double H)
{
  if (s<=H-eps)
    return 0.0;
  else{
    if ((s>=H-eps)&& (s<H))
      return 0.0 - (SQR(s-H+eps))/(4.*eps);
    else{
      if ((s>=H) && (s<H+eps))
  return s-H - (SQR(s-H+eps))/(4.*eps);
      else
  return 0.0;
    }
  }
}

static double regular(double eps,double s)
{
  if ((s>-eps)&&(s<=0))
    return 0.5*SQR(1+s/eps);
  else if  ((s<eps)&&(s>0)) return (1-0.5*SQR(1-s/eps));
  else if (s>eps) return 1;
  else return 0.;
}

static double der_regular(double eps,double s)
{
  if ((s>-eps)&&(s<=0))
    return (1+s/eps)*1./eps;
```

```
  else if  ((s<eps)&&(s>0)) return (1-s/eps)*1./eps;
  else return 0.;
}

static int MCStandard(double s, NumFunc_1  *p, double t,
    double r, double divid, double sigma, long N, int generator,
    double inc, double confidence,int delta_met, double *ptprice,
    double *ptdelta, double *pterror_price, double *pterror_delta ,
    double *inf_price, double *sup_price, double *inf_delta, double
    *sup_delta)
{
  int flag;
  long i;
  double mean_price, mean_delta, var_price, var_delta, forw
    ard, forward_stock,
    exp_sigmaxwt, S_T, U_T, price_sample, delta_sample=0.,
    s_plus, s_minus, K1, K2,sigma_sqrt;
  double g;
  int init_mc;
  int simulation_dim= 1;
  double alpha, z_alpha;
  double g_reg,g_reg_der,eps=1.0;

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

  /*Initialisation*/
  flag= 0;
  s_plus= s*(1.+inc);
  s_minus= s*(1.-inc);
  mean_price= 0.0;
  mean_delta= 0.0;
  var_price= 0.0;
  var_delta= 0.0;

  /* CallSpread */ /*For digit option K2 is the rebate*/
  K1= p->Par[0].Val.V_PDOUBLE;
  K2= p->Par[1].Val.V_PDOUBLE;

  /*Median forward stock and delta values*/
```

```
sigma_sqrt=sigma*sqrt(t);
forward= exp(((r-divid)-SQR(sigma)/2.0)*t);
forward_stock= s*forward;

/* Change a Call into a Put to apply the Call-Put parity
  */
if((p->Compute) == &Call)
  {
    (p->Compute) = &Put;
    flag= 1;
  }

/*MC sampling*/
init_mc= pnl_rand_init(generator,simulation_dim,N);

/* Test after initialization for the generator */
if(init_mc == OK)
  {


    /* Begin N iterations */
    for(i=1 ; i<=N ; i++)
{
  /* Simulation of a gaussian variable according to the     generator type,
     that is Monte Carlo or Quasi Monte Carlo. */
  g= pnl_rand_normal(generator);

  exp_sigmaxwt=exp(sigma_sqrt*g);
  S_T= forward_stock*exp_sigmaxwt;
  U_T= forward*exp_sigmaxwt;

  /*Price*/
  price_sample=(p->Compute)(p->Par,S_T);

  /*Delta*/

  /*Digit*/
  if ((p->Compute) ==  &Digit)
    {
      if (delta_met==1)
  delta_sample = ((p->Compute)(p->Par,U_T*s_plus)-
```

```
    (p->Compute)(p->Par,U_T*s_minus))/(2.*s*inc);
    if (delta_met==2)
/*Malliavin Global*/
delta_sample=(price_sample*g*sqrt(t))/(s*sigma*t);
    if(delta_met==3){
/*Malliavin Local*/
g_reg=K2*exp(-r*t)*regular(eps,S_T-K1);
g_reg_der=K2*exp(-r*t)*der_regular(eps,S_T-K1);
//f_loc=price_sample-g_reg;
delta_sample=((price_sample-g_reg)*g*sqrt(t))/(s*si
gma*t)+g_reg_der*S_T/s;
    }
  }

/* CallSpread */
else
  if ((p->Compute) ==  &CallSpread )
    {
if(delta_met==1){
  delta_sample= 0;
  if(S_T > K1)
    delta_sample += U_T;
  if(S_T > K2)
    delta_sample -= U_T;
}
if (delta_met==2)
  /*Malliavin Global*/
  delta_sample=(price_sample*g*sqrt(t))/(s*sigma*t);
if (delta_met==3){
  delta_sample=0.0;
  g_reg=reg_call(eps,S_T,K1);
  g_reg_der=exp(-r*t)*F_reg_call(eps,S_T,K1);
  delta_sample+=g_reg*U_T+g_reg_der*sqrt(t)*g/(s*si
gma*t);
  g_reg=reg_call(eps,S_T,K2);
  g_reg_der=exp(-r*t)*F_reg_call(eps,S_T,K2);
  delta_sample-=g_reg*U_T+g_reg_der*sqrt(t)*g/(s*si
gma*t);
}
    }
```

```
  /*Call-Put*/
    else
      if ((p->Compute) ==  &Put)
  {
    if (delta_met==1)
      {
        if (price_sample>0.)
    delta_sample= -U_T;
        else
    delta_sample= 0.0;
      }
    if (delta_met==2)
      /*Malliavin Global*/
      delta_sample=(price_sample*g*sqrt(t))/(s*sigma*t)
  ;
    if(delta_met==3){
      /*Malliavin Local*/
      g_reg=reg_put(eps,S_T,K1);
      g_reg_der=exp(-r*t)*F_reg_put(eps,S_T,K1);
      //f_loc=price_sample-g_reg;
      delta_sample=-(g_reg*U_T)+g_reg_der*g*sqrt(t)/(s*
  sigma*t);
    }

  }

  /*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= exp(-r*t)*(mean_price/(double) N);
    *pterror_price= sqrt(exp(-2.0*r*t)*var_price/(double)
  N - SQR(*ptprice))/sqrt(N-1);
```

```
      /*Delta*/
      *ptdelta= exp(-r*t)*mean_delta/(double) N;
      *pterror_delta= sqrt(exp(-2.0*r*t)*(var_delta/(
    double)N-SQR(*ptdelta)))/sqrt((double)N-1);

      /* Call Price and Delta with the Call Put Parity */
      if(flag == 1)
  {
    *ptprice+= s*exp(-divid*t)- p->Par[0].Val.V_DOUBLE*exp
    (-r*t);
    *ptdelta+= exp(-divid*t);
    (p->Compute)= &Call;
    flag = 0;
  }

      /* 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_Standard)(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 MCStandard(ptMod->S0.Val.V_PDOUBLE,
        ptOpt->PayOff.Val.V_NUMFUNC_1,
```

```
        ptOpt->Maturity.Val.V_DATE-ptMod->T.Val.V_DATE,
        r,
        divid,
        ptMod->Sigma.Val.V_PDOUBLE,
        Met->Par[0].Val.V_LONG,
        Met->Par[1].Val.V_ENUM.value,
        Met->Par[2].Val.V_PDOUBLE,
        Met->Par[3].Val.V_DOUBLE,
        Met->Par[4].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_Standard)(void *Opt, void *Mod)
{
  Option* ptOpt=(Option*)Opt;
  TYPEOPT* opt=(TYPEOPT*)(ptOpt->TypeOpt);

  if ((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=100000;
      Met->Par[1].Val.V_ENUM.value=0;
      Met->Par[1].Val.V_ENUM.members=&PremiaEnumRNGs;
      Met->Par[2].Val.V_PDOUBLE=0.01;
      Met->Par[3].Val.V_DOUBLE= 0.95;
      Met->Par[4].Val.V_ENUM.value=2;
      Met->Par[4].Val.V_ENUM.members=&PremiaEnumDeltaMC;


    }

  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_Standard)=
{
  "MC_Standard",
  {{"N iterations",LONG,{100},ALLOW},
```

```
   {"RandomGenerator",ENUM,{100},ALLOW},
   {"Delta Increment  Rel (Digit)",PDOUBLE,{100},ALLOW},
   {"Confidence Value",DOUBLE,{100},ALLOW},
   {"Delta Method",ENUM,{100},ALLOW},
   {" ",PREMIA_NULLTYPE,{0},FORBID}},
  CALC(MC_Standard),
  {{"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_Standard),
  CHK_mc,
  MET(Init)
};
```

References