```
   Help
/* Monte Carlo Simulation with Antithetic Variables for a
    Call - Put
 - CallSpread or Digit option.
 In the case of Monte Carlo simulation, the program provid
    es estimations for price and delta with a confidence interv
    al.
 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 MCAntithetic(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)
{
  short flag;
  long i;
  double g;
  int simulation_dim= 1;
  int init_mc;
  double mean_price, mean_delta, var_price, var_delta,forw
    ard, forward_stock,exp_sigmaxwt1, exp_sigmaxwt2, S_T1, U_T1
    , S_T2, U_T2, price1, price2,
  price_sample, delta_sample=0., price_sample_plus1, s_plus
    , price_sample_minus1, s_minus,
  price_sample_plus2, price_sample_minus2, brown, K1, K2,si
    gma_sqrt;
  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 */
```

```
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 with Antithetic Variables */
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);
        brown= sigma_sqrt*g;

        /* Antithetic Variables */
        exp_sigmaxwt1=exp(brown);
        exp_sigmaxwt2= 1./exp_sigmaxwt1;

        S_T1= forward_stock*exp_sigmaxwt1;
        U_T1= forward*exp_sigmaxwt1;
        S_T2= forward_stock*exp_sigmaxwt2;
        U_T2= forward*exp_sigmaxwt2;
```

```
      /*Price*/
      price1= (p->Compute)(p->Par,S_T1);
      price2= (p->Compute)(p->Par,S_T2);
      price_sample= 0.5*(price1 + price2);

      /*Delta*/

      /*Digit*/
      if ((p->Compute) ==  &Digit)
        {
          if (delta_met==1){
            price_sample_plus1= (p->Compute)(p->Par, U_
T1*s_plus);
            price_sample_minus1= (p->Compute)(p->Par,
U_T1*s_minus);
            price_sample_plus2= (p->Compute)(p->Par, U_
T2*s_plus);
            price_sample_minus2= (p->Compute)(p->Par,
U_T2*s_minus);
            delta_sample= (price_sample_plus1-price_sam
ple_minus1+price_sample_plus2-price_sample_minus2)/(4.*s*
inc);
          }
          if (delta_met==2){
            /*Malliavin Global*/
            delta_sample=((price1*g*sqrt(t))/(s*sigma*
t)-(price2*g*sqrt(t))/(s*sigma*t))/2.;
          }
          if(delta_met==3){
            /*Malliavin Local*/
            g_reg=K2*exp(-r*t)*regular(eps,S_T1-K1);
            g_reg_der=K2*exp(-r*t)*der_regular(eps,S_T1
-K1);
            delta_sample+=((price_sample-g_reg)*g*sqrt(
t))/(s*sigma*t)+g_reg_der*S_T1/s;
            g_reg=K2*exp(-r*t)*regular(eps,S_T2-K1);
            g_reg_der=K2*exp(-r*t)*der_regular(eps,S_T2
-K1);
            delta_sample+=-((price_sample-g_reg)*g*sq
rt(t))/(s*sigma*t)+g_reg_der*S_T1/s;
            delta_sample*=0.5;
```

```
          }
        }

      /* CallSpread */
      else
        if ((p->Compute) ==  &CallSpread )
          {
            if(delta_met==1){
              delta_sample= 0.;
              if(S_T1 > K1)
                delta_sample += U_T1;
              if(S_T1 > K2)
                delta_sample -= U_T1;
              if(S_T2 > K1)
                delta_sample += U_T2;
              if(S_T2 > K2)
                delta_sample -= U_T2;
              delta_sample/= 2.;
            }
            if (delta_met==2)
              /*Malliavin Global*/
              delta_sample=((price1*g*sqrt(t))/(s*sigma
*t)-(price2*g*sqrt(t))/(s*sigma*t))/2.;
            if (delta_met==3){
              delta_sample=0.0;
              g_reg=reg_call(eps,S_T1,K1);
              g_reg_der=exp(-r*t)*F_reg_call(eps,S_T1,
K1);
              delta_sample+=g_reg*U_T1+g_reg_der*sqrt(
t)*g/(s*sigma*t);

              g_reg=reg_call(eps,S_T1,K2);
              g_reg_der=exp(-r*t)*F_reg_call(eps,S_T1,
K2);
              delta_sample-=g_reg*U_T1+g_reg_der*sqrt(
t)*g/(s*sigma*t);

              g_reg=reg_call(eps,S_T2,K1);
              g_reg_der=exp(-r*t)*F_reg_call(eps,S_T2,
K1);
              delta_sample+=g_reg*U_T2-g_reg_der*sqrt(
```

```
t)*g/(s*sigma*t);

              g_reg=reg_call(eps,S_T2,K2);
              g_reg_der=exp(-r*t)*F_reg_call(eps,S_T2,
K2);
              delta_sample-=g_reg*U_T2-g_reg_der*sqrt(
t)*g/(s*sigma*t);
              delta_sample/=2.;

            }
          }

      /*Call-Put*/
        else
          if ((p->Compute) ==  &Put)
            {
              if (delta_met==1){
                delta_sample= 0.0;
                if (price1 > 0.)
                  delta_sample+= -U_T1;
                if (price2 > 0.)
                  delta_sample+= -U_T2;
                delta_sample/= 2.;
              }
              if (delta_met==2)
                /*Malliavin Global*/
                delta_sample=((price1*g*sqrt(t))/(s*si
gma*t)-(price2*g*sqrt(t))/(s*sigma*t))/2.;
              if(delta_met==3){
                /*Malliavin Local*/
                delta_sample=0.0;
                g_reg=reg_put(eps,S_T1,K1);
                g_reg_der=exp(-r*t)*F_reg_put(eps,S_T1,
K1);
                delta_sample+=-(g_reg*U_T1)+g_reg_der*
g*sqrt(t)/(s*sigma*t);

                g_reg=reg_put(eps,S_T2,K1);
                g_reg_der=exp(-r*t)*F_reg_put(eps,S_T2,
K1);
                delta_sample+=-(g_reg*U_T2)-g_reg_der*
```

```
g*sqrt(t)/(s*sigma*t);

                delta_sample/=2.;
              }
            }

      /*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_Antithetic)(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 MCAntithetic(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_Antithetic)(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_Antithetic)=
{
  "MC_Antithetic",
  {{"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_Antithetic),
  {{"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_Antithetic),
  CHK_mc,
  MET(Init)
};
```

References