```
   Help
#include  "purejump1d_pad.h"
#include "error_msg.h"
#include "enums.h"
#include "pnl/pnl_cdf.h"
#define INC 1.0e-5 /*Relative Increment for Delta-Hedging*/

/* --------------------------------------------------------
    --------- */
/* Pricing of a asian option by the Monte Carlo Privault
    method
 Estimator of the price and the delta.
 s et K are pseudo-spot and pseudo-strike. */
/* --------------------------------------------------------
    --------- */

/* Generation of Exponential Law.
 Inter Jump Times */
static double expdev(int generator)
{
  double dum;

  do dum=pnl_rand_uni(generator);
  while (dum == 0.0);
  return -log(dum);
}


static int  FixedAsian_Privault(double s,double K, double
    time_spent, NumFunc_2 *p, double T, double r, double sigma,
    double beta,double nu, int N,int generator, double confidence,
    int delta_met, double *ptprice,double *ptdelta, double *pt
    error_price, double *pterror_delta, double *inf_price, double *
    sup_price, double *inf_delta, double *sup_delta)

{
 long i,j;
 double mean_price, mean_delta, var_price, var_delta,
    price_sample, delta_sample=0., s_plus, s_minus;
 int init_mc;
 int simulation_dim= 1;
```

```
double alpha,gamma, z_alpha;
double *t, *sk,*jump_size;
double average,DwIntS,DDwIntS,Intw,DwG;
int k;
double wTk,wTk1,G;
int NUMBER_OF_JUMPS=1000;

/*Memory allocation*/
t= malloc((NUMBER_OF_JUMPS+1)*sizeof(double));
if (t==NULL)
  return MEMORY_ALLOCATION_FAILURE;

sk= malloc((NUMBER_OF_JUMPS+1)*sizeof(double));
if (sk==NULL)
  return MEMORY_ALLOCATION_FAILURE;

jump_size= malloc((NUMBER_OF_JUMPS+1)*sizeof(double));
if (jump_size==NULL)
  return MEMORY_ALLOCATION_FAILURE;

/* double inc=0.001;*/

/* Renormalized the sigma */
sigma=sigma/sqrt(nu);

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

/*Initialisation*/
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;

/*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(j=1 ; j<=N; j++)
      {
        average=0.;DwIntS=0;DDwIntS=0;Intw=0;
        /* Simulation of Poisson Jump Times */
        t[0]=0;
        k=0;
        while (t[k] < T)
          {
            k=k+1;
            t[k]=t[k-1]+expdev(generator)/nu;

          }
        if (k>1)
          {
            jump_size[0]=beta;
            sk[0]=1;

            /*Computation of Average and the Weight*/
            for (i=1;i<k;i++)
              {
                jump_size[i]=beta;
                sk[i]=sk[i-1]*(1.+sigma*jump_size[i-1]);

                average=average+  sk[i-1] * ( exp(gamma*t[
   i]) - exp(gamma*t[i-1] ))/gamma;

                /*Useful for computation of the weight*/
                if (delta_met==2)
                  {
                    wTk=sin(M_PI*t[i]/T);
                    wTk1=cos(M_PI*t[i]/T)*M_PI/T;

                    DwIntS=DwIntS + sigma * wTk * sk[i-1]
   * exp(gamma*t[i]) * jump_size[i-1];
                    DDwIntS=DDwIntS - sigma * wTk * sk[i-1
   ] * exp(gamma*t[i]) * jump_size[i-1] * (wTk1+gamma*wTk);
                    Intw=Intw+wTk1;
                  }
              }
            /*Average*/
            average=average + sk[k-1] * (exp(gamma*T) -
   exp(gamma*t[k-1]))/gamma;


            /*Price*/
            price_sample=(p->Compute)(p->Par, s, average*
   s/T);

            /*Delta*/
            /*Finite Difference*/
            if (delta_met==1)
              {
                delta_sample = ((p->Compute)(p->Par, s_pl
   us, s_plus*average/T)-(p->Compute)(p->Par, s_minus, s_minus
   *average/T))/(2.*s*INC);
              }
            /*Malliavin*/
            if (delta_met==2)
              {
                G= average/DwIntS/s;
                DwG=(1.-(average * DDwIntS / pow(DwIntS,2.
   ) ))/s;
                if (price_sample>0.)
                  delta_sample=(G*Intw-DwG)*(average*s/T-
   K);
                else delta_sample=0.;
              }

            /*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);


     /* 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(t);
 free(sk);
 free(jump_size);

 return init_mc;

}



int CALC(MC_FixedAsian_Privault)(void *Opt,void *Mod,Prici
    ngMethod *Met)
{
  TYPEOPT* ptOpt=(TYPEOPT*)Opt;
  TYPEMOD* ptMod=(TYPEMOD*)Mod;

  double T, t_0, T_0;
  double r, time_spent, pseudo_strike, true_strike, pseudo_
    spot;
```

```
int return_value;

r=log(1.+ptMod->R.Val.V_DOUBLE/100.);

/*divid=log(1.+ptMod->Divid.Val.V_DOUBLE/100.);*/
T= ptOpt->Maturity.Val.V_DATE;
T_0 = ptMod->T.Val.V_DATE;
t_0= (ptOpt->PathDep.Val.V_NUMFUNC_2)->Par[0].Val.V_PDOUB
  LE;
time_spent= (T_0-t_0)/(T-t_0);

if(T_0 < t_0)
  {
    Fprintf(TOSCREEN,"T_0 < t_0, untreated case{n{n{n");
    return_value = WRONG;
  }

/* Case t_0 <= T_0 */
else
  {
    pseudo_spot= (1.-time_spent)*ptMod->S0.Val.V_PDOUBLE;
    pseudo_strike= (ptOpt->PayOff.Val.V_NUMFUNC_2)->Par[0
  ].Val.V_PDOUBLE-time_spent*
    (ptOpt->PathDep.Val.V_NUMFUNC_2)->Par[4].Val.V_PDOUB
  LE;


    true_strike= (ptOpt->PayOff.Val.V_NUMFUNC_2)->Par[0].
  Val.V_PDOUBLE;

    (ptOpt->PayOff.Val.V_NUMFUNC_2)->Par[0].Val.V_PDOUB
  LE= pseudo_strike;

    return_value= FixedAsian_Privault(pseudo_spot,
                                      pseudo_strike,
                                      time_spent,
                                      ptOpt->PayOff.Val.
  V_NUMFUNC_2,
                                      T-T_0,
                                      r,
                                      /*divid,*/
```

```
                                        ptMod->Sigma.Val.V_
    PDOUBLE,
                                        ptMod->Beta.Val.V_
    DOUBLE,
                                        ptMod->Nu.Val.V_
    DOUBLE,
                                        Met->Par[0].Val.V_
    LONG,
                                        Met->Par[1].Val.V_
    ENUM.value,
                                        Met->Par[2].Val.V_
    DOUBLE,
                                        Met->Par[3].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));

      (ptOpt->PayOff.Val.V_NUMFUNC_2)->Par[0].Val.V_PDOUB
    LE=true_strike;
    }
  return return_value;
}



static int CHK_OPT(MC_FixedAsian_Privault)(void *Opt, void
    *Mod)
```

```
{

  if ( (strcmp( ((Option*)Opt)->Name,"AsianCallFixedEuro")=
    =0))
    return OK;

  return WRONG;
}

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

static DEFINE_ENUM(delta_method_privault, delta_method_priv
    ault_members)

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_DOUBLE= 0.95;
      Met->Par[3].Val.V_ENUM.value=1;
      Met->Par[3].Val.V_ENUM.members=&delta_method_privault
    ;

    }


  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_FixedAsian_Privault)=
{
  "MC_FixedAsian_Privault",
  {
    {"N iterations",LONG,{100},ALLOW},
    {"RandomGenerator",ENUM,{100},ALLOW},
    {"Confidence Value",DOUBLE,{100},ALLOW},
    {"Delta Method",ENUM,{100},ALLOW},
    {" ",PREMIA_NULLTYPE,{0},FORBID}},
  CALC(MC_FixedAsian_Privault),
  {{"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_FixedAsian_Privault),
  CHK_mc,
  MET(Init)
};
```

References