Help

```
#include  "hes1d_std.h"
#include "enums.h"
#include "pnl/pnl_cdf.h"


#if defined(PremiaCurrentVersion) && PremiaCurrentVersion <
     (2009+2) //The "#else" part of the code will be freely av
    ailable after the (year of creation of this file + 2)
static int CHK_OPT(MC_Pelsser_Heston)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(MC_Pelsser_Heston)(void*Opt,void *Mod,Pricing
    Method *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else

static double ran_gamma_pdf (const double x, const double
    a)
{
  double p,lngamma_a;
  if (x < 0)
    {
      return 0 ;
    }
  else if (x == 0)
    {
      if (a == 1)
        return 1. ;
      else
        return 0 ;
    }
  else if (a == 1)
    {
      return exp(-x) ;
    }
  else
    {
```

```
      lngamma_a = lgamma (a);
      p = exp ((a - 1) * log (x) - x - lngamma_a);
      return p;
    }
}

static double cdf_gamma_Q (double x,  double a)
{
  int which;
  double scale, bound;
  int status;
  double p, q;
  which = 1;
  scale = 1.;

  pnl_cdf_gam (&which, &p, &q, &x, &a, &scale, &status, &bo
    und);
  return q;
}

void cdf_gamma_Qinv_unif_grid (unsigned int N, double * inv
    erse, const double a, const double b)
{
  //we would like to map the inverse of the gamma(a,b) dis
    tribution on the grid {k/N,k=0,...,N-1}.
  // at step k, a close guess to the inverse of k/N is the
    invere of (k-1)/N that we just computed !!!
  double lambda, dQ, phi;
  double Q, x;
  double step0,step1,step;
  int i, j;

  inverse[0]=0.0;
  {
    Q=1.-1./N;
    x=exp ((lgamma (a) + log (1.-Q)) / a);

    for (i=0;i<35;i++)// maximum numer of iteration is 35
      {
        dQ = Q - cdf_gamma_Q (x, a);
        phi = ran_gamma_pdf (x, a);
```

```
      if (dQ == 0.0) break;
      else{
        lambda = -dQ / MAX (2 * fabs (dQ / x), phi);
        step0 = lambda;
        step1 = -((a - 1) / x - 1) * lambda * lambda / 4.
  0;
        step = step0;
        if (fabs (step1) < fabs (step0)) step += step1;

        if (x + step > 0) x += step;
        else   x /= 2.0;

        if (fabs (step0) < 1e-10 * x) break;
      }
    }
  inverse[1]= b*x;
}
for ( j=2;j<N;j++)
  {
    Q=1.-(double)j/N;
    x=inverse[j-1];

    for (i=0;i<35;i++)// maximum numer of iteration is 35
      {
        dQ = Q - cdf_gamma_Q (x, a);
        phi = ran_gamma_pdf (x, a);

        if (dQ == 0.0) break;
        else{
          lambda = -dQ / MAX (2 * fabs (dQ / x), phi);
          step0 = lambda;
          step1 = -((a - 1) / x - 1) * lambda * lambda /
  4.0;
          step = step0;
          if (fabs (step1) < fabs (step0)) step += step1;

          if (x + step > 0) x += step;
          else   x /= 2.0;

          if (fabs (step0) < 1e-10 * x) break;
```

```
          }
        }
      inverse[j] =b*x;
    }
}

//we would like to map the inverse of the chi_squre(nu) dis
    tribution on the grid {k/N,k=0,...,N-1}.

static void cdf_chisq_Qinv_unif_grid (unsigned int N,
    double * inverse, const double nu)
{
  cdf_gamma_Qinv_unif_grid ( N, inverse, nu / 2, 2.0);
}

static double cdf_chisq_Qinv(double q, double nu)
{
  int which;
  double x, p, bound;
  int status;
  p = 1. - q;
  which = 2;

  pnl_cdf_chi (&which, &p, &q, &x, &nu, &status, &bound);
  return x;
}


int MCPelsser(double S0, NumFunc_1  *pf, double T, double
    r, double divid, double v0,double K_heston,double Theta,
    double sigma,double rho, int N_sample,int N_t_grid,int     generator,  doubl
    double *ptdelta, double *pterror_price, double *pterror_delta ,
    double *inf_price, double *sup_price, double *inf_delta, double
    *sup_delta)
{
  double delta;
  int i;
  int j;
  double g1,unif;
  double price_sample, delta_sample, mean_price, mean_delt
    a, var_price, var_delta;
```

```
double alpha, z_alpha;
double V,log_S;
double erT;
double d, ekd, nekd, C0,NN;
double K1,K2,K3,K4,A,B;
double C6,C7,C8;
double K000;
int Nmax;// nombre maximum de poisson simul
int N_grid; //Number of grid points nombre d'inveverse
  calcul
double gamma1, gamma2; // CENTRAL DISCRETIZATION
double **Cache;
int N,k;
double Vi, lambda, gen;
int pois;

delta= T/N_t_grid;;
erT=exp((r-divid)*T);
Nmax=40;
N_grid=10000;
gamma1=0.5;
gamma2=0.5;
gen=0.;

//Memory allocation
Cache=malloc((Nmax+1)*sizeof(double*));
if (Cache==NULL)
  return MEMORY_ALLOCATION_FAILURE;
for (i=0;i<Nmax+1;i++)
  {
    Cache[i]=malloc(N_grid*sizeof(double));
    if (Cache[i]==NULL) return MEMORY_ALLOCATION_FAILURE;
  }

//Useful constants
d=4*K_heston*Theta/(sigma*sigma);
ekd=exp(-K_heston*delta);
nekd= 1.- ekd;
C0=pow(sigma,2.)*nekd/(4*K_heston);
NN=ekd/C0;
K1=gamma1*delta*(K_heston*rho/sigma-0.5)-rho/sigma;
```

```
K2=gamma2*delta*(K_heston*rho/sigma-0.5)+rho/sigma;
K3=gamma1*delta*(1-pow(rho,2.));
K4=gamma2*delta*(1-pow(rho,2.));

A=K2+0.5*K4;
B=K1+0.5*K3;

C6=-C0*A/(1.-2.*C0*A);
C7=0.5*d*log(1.-2.*C0*A);
C8=-B;


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

/*Initialisation*/
mean_price= 0.0;
mean_delta= 0.0;
var_price= 0.0;
var_delta= 0.0;

pnl_rand_init(generator,1,N_sample);

//Pre-compute inverse Chi2 on an array
for(N=0;N<Nmax+1;N++)
  cdf_chisq_Qinv_unif_grid(N_grid,Cache[N],d+2*N);

for(j=0;j<N_sample;j++)
  {
    // N_path Paths
    V=v0;
    log_S=log(S0);
    for(i=0; i<N_t_grid; i++)
      {
        unif=pnl_rand_uni(generator);
        g1=pnl_rand_normal(generator);
        Vi=V;
        lambda=NN*Vi;
        pois= pnl_rand_poisson(lambda*0.5,generator);
        if(pois<=Nmax) {
```

```
          k=(int)floor(unif*N_grid);
          if(k>N_grid-2) k=N_grid-2;
          gen=((k+1.)-N_grid*unif)*Cache[pois][k]
            +(N_grid*unif-k)*Cache[pois][k+1];
        }
        else{
          gen=cdf_chisq_Qinv(unif,d+2*pois);
        }

        V=C0*gen;
        K000=C6*lambda+C7+C8*Vi;
        log_S += K000 + K1*Vi + K2*V+ sqrt(K3*Vi+K4*V)*g1
  ;

      }

    /*Price*/
    price_sample=(pf->Compute)(pf->Par,erT*exp(log_S));

    /* Delta */
    if(price_sample >0.0)
      delta_sample=(erT*exp(log_S)/S0);
    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 of the N iterations */

/* Price estimator */
*ptprice=(mean_price/(double)N_sample);
*pterror_price= exp(-r*T)*sqrt(var_price/(double)N_sampl
  e-SQR(*ptprice))/sqrt((double)N_sample-1);
*ptprice= exp(-r*T)*(*ptprice);
```

```
  /* Price Confidence Interval */
  *inf_price= *ptprice - z_alpha*(*pterror_price);
  *sup_price= *ptprice + z_alpha*(*pterror_price);


  /* Delta estimator */
  *ptdelta=exp(-r*T)*(mean_delta/(double)N_sample);
  if((pf->Compute) == &Put)
    *ptdelta *= (-1);
  *pterror_delta= sqrt(exp(-2.0*r*T)*(var_delta/(double)N_
    sample-SQR(*ptdelta)))/sqrt((double)N_sample-1);

  /* Delta Confidence Interval */
  *inf_delta= *ptdelta - z_alpha*(*pterror_delta);
  *sup_delta= *ptdelta + z_alpha*(*pterror_delta);

  /*Memory desallocation*/
  for (i=0;i<=Nmax;i++)
    free(Cache[i]);
  free(Cache);

  return OK;
}

int CALC(MC_Pelsser_Heston)(void *Opt, void *Mod, Pricing
    Method *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 MCPelsser(ptMod->S0.Val.V_PDOUBLE,
                   ptOpt->PayOff.Val.V_NUMFUNC_1,
                   ptOpt->Maturity.Val.V_DATE-ptMod->T.Val.
    V_DATE,
                   r,
                   divid, ptMod->Sigma0.Val.V_PDOUBLE
                   ,ptMod->MeanReversion.Val.V_PDOUBLE,
```

```
                    ptMod->LongRunVariance.Val.V_PDOUBLE,
                    ptMod->Sigma.Val.V_PDOUBLE,
                    ptMod->Rho.Val.V_PDOUBLE,
                    Met->Par[0].Val.V_INT,
                    Met->Par[1].Val.V_INT,
                    Met->Par[2].Val.V_ENUM.value,
                    Met->Par[3].Val.V_RGDOUBLE12,
                    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_Pelsser_Heston)(void *Opt, void *Mod)
{

  if ((strcmp( ((Option*)Opt)->Name,"CallEuro")==0)||(strc
    mp( ((Option*)Opt)->Name,"PutEuro")==0))
    return OK;

  return  WRONG;
}

#endif //PremiaCurrentVersion
static int MET(Init)(PricingMethod *Met,Option *Opt)
{
  //int type_generator;
  if ( Met->init == 0)
    {
      Met->init=1;

      Met->Par[0].Val.V_INT=10000;
      Met->Par[1].Val.V_INT=4;
      Met->Par[2].Val.V_ENUM.value=0;
      Met->Par[2].Val.V_ENUM.members=&PremiaEnumMCRNGs;
      Met->Par[3].Val.V_RGDOUBLE12= 1.5;
```

```
      Met->Par[4].Val.V_DOUBLE= 0.95;
    }

  return OK;
}

PricingMethod MET(MC_Pelsser_Heston)=
{
  "MC_Pelsser",
  {{"N iterations",INT,{100},ALLOW},
   {"TimeStepNumber",INT,{100},ALLOW},
   {"RandomGenerator",ENUM,{100},ALLOW},
   {"THRESHOLD",DOUBLE,{100},ALLOW},
   {"Confidence Value",DOUBLE,{100},ALLOW},
   {" ",PREMIA_NULLTYPE,{0},FORBID}},
  CALC(MC_Pelsser_Heston),
  {{"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_Pelsser_Heston),
  CHK_mc,
  MET(Init)
};
```

References