```
   Help
#include  "hes1d_std.h"
#include "enums.h"
#include "math/ESM_func.h"
#include "pnl/pnl_random.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_Smith_Heston)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(MC_Smith_Heston)(void*Opt,void *Mod,PricingMethod
    *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else

int MCSmith(double S0, NumFunc_1  *pf, double T, double r,
    double divid, double v0,double K_heston,double Theta,double si
    gma,double rho, long N_sample,int N_t_grid,int generator,
    double threshold,double confidence, double *ptprice, double *pt
    delta, double *pterror_price, double *pterror_delta ,
    double *inf_price, double *sup_price, double *inf_delta, double
    *sup_delta)
{
  double delta;
  int i,M;
  long k;
  double g1,g2;
  double price_sample, delta_sample, mean_price, mean_delt
    a, var_price, var_delta;
  double alpha, z_alpha;
  double u;
  double d, ekd, nekd, C0,B;
  double sq_rho, KTD,RS,KRS;
  double a,KKK;
  double Vi;
  double erT;
```

```
double V,log_S;
double *mean,*variance,*h;
double **values;
int *N_vect, NS, j;
double Vmax, inv, Act_i, Nact_i, b, hh, m, p, Vmax_i, z,
  Vst, var, omega, lambda, gen,eps1,epsilon,pois;

delta = T/N_t_grid;
erT=exp((r-divid)*T);
M=10000;

//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);
B=ekd/C0;
sq_rho=sqrt(1-rho*rho);
KTD=K_heston*Theta*delta;
RS=rho/sigma;
KRS=K_heston*RS-0.5;
ESM_update_const_char(K_heston, sigma, delta, d);

a=0.5*d;
KKK=2.*K_heston*v0/pow(sigma,2.);
 eps1=1e-5;
inv=- pnl_inv_cdfnor(eps1);
Vmax=v0;
for (i=1;i<N_t_grid+1;i++){
  Act_i=pow(ekd,(double)i);
  Nact_i=1. - Act_i;
  b=KKK*Act_i/Nact_i;
  hh=1. - 2./3.*(a+b)*(a+3.*b)/pow(a+2*b,2.);
  m=(hh-1.)*(1-3*hh);
  p=.5*(a+2*b)/pow(a+b,2);

  Vmax_i=(a+b)*(.5*pow(sigma,2)*Nact_i/K_heston)*pow(inv*
  hh*sqrt(2*p*(1.+m*p))+1.-hh*p*(1-hh+(2.-hh)*m*p/2.),1./hh);
  Vmax=MAX(Vmax_i,Vmax);
}
```

```
NS=2000;
omega=0.5;

//Memory allocation
mean= malloc((NS)*sizeof(double));
if (mean==NULL)
return MEMORY_ALLOCATION_FAILURE;

variance= malloc((NS)*sizeof(double));
if (variance==NULL)
return MEMORY_ALLOCATION_FAILURE;

h= malloc((NS)*sizeof(double));
if (h==NULL)
return MEMORY_ALLOCATION_FAILURE;

N_vect= malloc((NS)*sizeof(int));
if (N_vect==NULL)
return MEMORY_ALLOCATION_FAILURE;

values=(double**)calloc(NS,sizeof(double*));
 if (values==NULL)
 return MEMORY_ALLOCATION_FAILURE;
for (i=0;i<NS;i++)
  {
    values[i]=(double *)calloc(M,sizeof(double));
    if (values[i]==NULL)
    return MEMORY_ALLOCATION_FAILURE;
 }

epsilon=1.e-6;

values_all_AESM(M,Vmax, NS , K_heston, sigma, delta, d,ep
   silon, mean, variance,  h, N_vect, values);

/* 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);

for(k=0; k<N_sample; k++ )
  {      // N_path Paths

    V=v0;
    log_S=log(S0);
    for(i=0; i<N_t_grid; i++)
      {
        u=pnl_rand_uni(generator);
        g2=pnl_rand_normal(generator);

        Vi=V;
        lambda=B*Vi;


        if(d>1){
        g1=pnl_rand_normal(generator);
          gen=pow(g1+sqrt(lambda),2.)+pnl_rand_chi2(d-1.,
          generator);
        }
        else{
          pois=pnl_rand_poisson(lambda*0.5,generator);
          gen=pnl_rand_chi2(d+2*pois, generator);
        }

        V=C0*gen;

        z=omega*Vi+(1.-omega)*V;
        j=floor(z*NS/Vmax);
        if(j+1>NS)
          j=NS-1;

        Vst= inverse_ESM( u, h[j], N_vect[j], values[j]);

        Moments_ESM( Vi,V, K_heston, sigma, delta, d, &m,
```

```
    &var);

         Vst = sqrt(var/variance[j])*(Vst-mean[j])+m;//moment
         //matching
         Vst =MAX(Vst,0.);
         log_S += RS *( V - Vi - KTD)+ KRS*Vst+sq_rho*sq
  rt(Vst)*g2;
      }

    /*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*/                    {
  free(mean);
  free(variance);
  free(h);
  free(N_vect);
  for (i=0;i<NS;i++)
    free(values[i]);
  free(values);

  return OK;
}

int CALC(MC_Smith_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 MCSmith(ptMod->S0.Val.V_PDOUBLE,
                 ptOpt->PayOff.Val.V_NUMFUNC_1,
                 ptOpt->Maturity.Val.V_DATE-ptMod->T.Val.V_DA
    TE,
                 r,
                 divid, ptMod->Sigma0.Val.V_PDOUBLE
                 ,ptMod->MeanReversion.hal.V_PDOUBLE,
                 ptMod->LongRunVariance.Val.V_PDOUBLE,
                 ptMod->Sigma.Val.V_PDOUBLE,
```

```
                ptMod->Rho.Val.V_PDOUBLE,
                Met->Par[0].Val.V_LONG,
                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_Smith_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_LONG=10000;
      Met->Par[1].Val.V_INT=1;
      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_Smith_Heston)=
{
  "MC_Smith",
  {{"N iterations",LONG,{100},ALLOW},
   {"TimeStepNumber",LONG,{100},ALLOW},
   {"RandomGenerator",ENUM,{100},ALLOW},
   {"THRESHOLD",DOUBLE,{100},ALLOW},
   {"Confidence Value",DOUBLE,{100},ALLOW},
   {" ",PREMIA_NULLTYPE,{0},FORBID}},
  CALC(MC_Smith_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_Smith_Heston),
  CHK_mc,
  MET(Init)
};
```

References