Help

```
extern "C" {
#include  "
href../../../mod/doublehes1d/doublehes1d_std/doublehes1d_std_h_src.pdfhes1d_std.
#include "pnl/pnl_random.h"
#include "pnl/pnl_cdf.h"
#include "pnl/pnl_specfun.h"
#include "
href../../../common/enums_h_src.pdfenums.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2015+2) //The "#els
static int CHK_OPT(MC_TseWan)(void *Opt, void *Mod)
{
  return NONACTIVE;
}

int CALC(MC_TseWan)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else

static double algo1_v3(double V_0, double sigma, double delta, double k,
   double t2, int generator)
{
  double V_t2, chi2;
  double cte1 = k*t2;
  double n_t1_t2 = (4.*k*pow(exp(1.), -cte1)) / (sigma*sigma*(1.-pow(exp(1.), -c

  if (delta > 1)
    {
  double Z = pnl_rand_normal(generator) + sqrt(V_0*n_t1_t2);
  chi2 = Z*Z + pnl_rand_chi2(delta - 1., generator);
    }
  else
    {
  long int Np;
  Np = pnl_rand_poisson(0.5*V_0*n_t1_t2, generator);
  chi2 = pnl_rand_chi2(2*Np + delta, generator);
    }
```

```
  double ekd, nekd, C0;
  ekd = exp(-cte1);
  nekd = 1. - ekd;
  C0 = sigma*sigma*nekd/(4.*k);
  V_t2 = C0 * chi2;

  return V_t2;
}


static double algo3(double m, double s, int generator)
{
  // We generate a standard normal variate N and a uniform variate U.
  double N = pnl_rand_normal(generator); //N = normrnd(m, s);
  double U = pnl_rand_uni(generator);

  double x = 1. + N*N/(2.*s/m) - sqrt((2.*s/m + 2.*s/m)*(N*N) + N*N*N*N)/(2.*s/m

  double Ic;
  if ((U*(1. + x)) > 1.)
Ic = m/x;
  else
Ic = m*x;

  return Ic;
}


static void binarySearchAlgo4(double* PreE_IcV, double* PreVar_IcV, double* PreE
  int Nu, double searchedValue, double* v)
{
  int j;
  double acc, accAux;
  int infm, sup, middle;

  // Search the index of v which approach better v[j] to searchedValue
  acc = 100000000000.;
  infm = 0;
  sup = Nu-1;
  j = 0;
```

```
  while (infm <= sup)
    {
  middle = floor((sup+infm)/2.);
  accAux = v[middle] - searchedValue;
  if (fabs(accAux) < acc)
        {
  acc = fabs(accAux);
  j = middle;
  if (accAux == 0.)
break;
  else if (accAux < 0) // accAux is negative
infm = middle + 1;
  else
sup = middle - 1;
        }
  else
break;
    }
  *PreE_IcV   = PreE_Ic[j];
  *PreVar_IcV = PreVar_Ic[j];

  return;
}


static void algo4(double Inc_t, int Nu, double k, double V_t1, double V_t2, doub
  double* PreE_Ic, double* PreVar_Ic, double* E_Ic, double* Var_Ic, double* v)
{
  double C1 = cosh(k*Inc_t/2.) / sinh(k*Inc_t/2.);
  double C2 = 1. / pow(sinh(k*Inc_t/2.), 2.);

  double E_X2 = delta*sigma*sigma*(-2. + k*Inc_t*C1)/(4.*k*k);
  double sigmaSq_X2 = delta*sigma*sigma*sigma*sigma*(-8. + 2.*k*Inc_t*C1
 + k*k*Inc_t*Inc_t*C2)/(8.*k*k*k*k*k);

  double E_X1;
  double sigmaSq_X1;

  E_X1 = (V_t1 + V_t2)*(C1/k - Inc_t*C2/2.);
  sigmaSq_X1 = (V_t1 + V_t2)*(sigma*sigma*C1/(k*k*k) + sigma*sigma*Inc_t*C2/(2.*
  - sigma*sigma*Inc_t*Inc_t*C1*C2/(2.*k));
```

```
  if ((V_t1 == 0.) || (V_t2 == 0.))
    {
  *E_Ic   = E_X1 + E_X2;
  *Var_Ic = sigmaSq_X1 + sigmaSq_X2;
    }
  else // use nearest neighbor interpolation to approximate E_Ic and Var_Ic
    {
  double PreE_IcV, PreVar_IcV;
  double searchedValue = sqrt(V_t1*V_t2);
  binarySearchAlgo4(&PreE_IcV, &PreVar_IcV, PreE_Ic, PreVar_Ic, Nu, searchedValu
  *E_Ic   = E_X1 + PreE_IcV;
  *Var_Ic = sigmaSq_X1 + PreVar_IcV;
    }

  return;
}

static void algo4Precomputing(double Inc_t, double* v, int Nu, double k, double
  double delta, double* PreE_Ic, double* PreVar_Ic)
{
  int i;

  double C1 = cosh(k*Inc_t/2.) / sinh(k*Inc_t/2.);
  double C2 = 1. / (sinh(k*Inc_t/2.)*sinh(k*Inc_t/2.));

  double E_eta;
  double E_etaSq;

  double E_X2 = delta*sigma*sigma*(-2. + k*Inc_t*C1)/(4.*k*k);
  double E_Z = 4.*E_X2/delta;
  double sigmaSq_X2 = delta*sigma*sigma*sigma*sigma*(-8. + 2.*k*Inc_t*C1
 + k*k*Inc_t*Inc_t*C2)/(8.*k*k*k*k);
  double sigmaSq_Z = 4.*sigmaSq_X2/delta;
  double nu = delta/2. - 1.;
  double Cz = 2.*k/(sigma*sigma*sinh(k*Inc_t/2.));

  double z;
  for (i = 0; i<Nu; i++)
    {
  z = Cz*v[i]; // v[i] = sqrt(V_ti*V_t(i+1))
```

```
  E_eta = z*pnl_bessel_i(nu + 1., z)/(2.*pnl_bessel_i(nu, z)); // modified Besse
  E_etaSq = z*z*pnl_bessel_i(nu + 2., z)/(4.*pnl_bessel_i(nu, z)) + E_eta; // mo

  PreE_Ic[i] = E_X2 + E_eta*E_Z;
  PreVar_Ic[i] = sigmaSq_X2 + E_eta*sigmaSq_Z + (E_etaSq-E_eta*E_eta)*E_Z*E_Z;
    }

  return;
}


/** BEFORE SIMULATION, precompute E[I_c]_V(t1)V(t2) and Var[I_c]_V(t1)V(t2) as
specified in Algorithm 4 **/
static void precomputing(double* v, int Nu, double k, double T, double sigma,
 double delta, double* PreE_Ic, double* PreVar_Ic)
{

  algo4Precomputing(T, v, Nu, k, sigma, delta, PreE_Ic, PreVar_Ic);

  return;
}


static double HestonModelIG(int Nu, double k, double T, double sigma,
                     double theta, double V_0, double delta, double* v,
                     double rho, double r, double X_0, double* PreVar_Ic,
                     double* PreE_Ic, int generator)
{
  /** DURING SIMULATION at time t2, when (X(t1), V(t1)) is known, the IPZ-IG
  scheme samples (X(t2), V(t2)) as follows : */

  /** Sample V(t2) by the formule **/
  double V_T = algo1_v3(V_0, sigma, delta, k,T, generator);

  /** Calculate E[Ic] and Var[Ic] using Algorithm 4 **/
  double E_Ic, Var_Ic;
  algo4(T, Nu, k, V_0, V_T, sigma, delta, PreE_Ic, PreVar_Ic, &E_Ic, &Var_Ic, v)

  /** Sample Ic using the moment-matched IG distribution by Algorithm 3 **/
  double Ic = algo3(E_Ic, (E_Ic*E_Ic*E_Ic)/Var_Ic, generator);
```

```
  /** Conditional on V(t1), V(t2) and Ic, sample X(t2) using equation (4) **/
  /* Note that the last step is actually irrelevant to the IPZ-IG scheme, but
 nevertheless necessary for getting a sample of X(t2). */
  double approach;
  double X_T;
  approach = pnl_rand_normal(generator)*sqrt((1.-rho*rho)*Ic) +
(r*T - 0.5*Ic + (rho/sigma)*(V_T-V_0-k*theta*T+k*Ic));
  X_T = exp(approach) * X_0;

  return X_T;
}

int MCTseWan(double X_0, NumFunc_1  *p, double T, double r, double q, double V_0
{
  int N;
  int Nu;
  int i;
  double *v;
  double v_min = 0.0001;
  double v_max = 8*sigma;
  int init_mc;
  double alpha, z_alpha;
  long dummy = 0;

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

  N=4;
  Nu = pow(2, 15 + ceil(log(N*1.)/log(2.))) + 1;

  v = (double*) malloc(Nu*sizeof(double));
  for (i = 0; i < Nu; i++)
v[i] = v_min + i*(v_max-v_min)/(Nu-1.);

  /* Replace Nu*iters by dummy because the 3rd input
     parameter is not used but may cause overflow with gcc */
  init_mc=pnl_rand_init(generator, 1, dummy);

  double delta = (4.*k*theta)/(sigma*sigma);
```

```
  double* PreE_Ic = (double*) malloc(Nu*sizeof(double));
  double* PreVar_Ic = (double*) malloc(Nu*sizeof(double));

  precomputing(v, Nu, k, T, sigma, delta, PreE_Ic, PreVar_Ic);

  double S_T;
  double sum = 0.;
  double sum2=0.;
  double payoff;
  for (i = 0; i < iters; i++)
    {
  S_T = HestonModelIG(Nu, k, T, sigma, theta, V_0, delta, v, rho,
 r, X_0, PreVar_Ic, PreE_Ic, generator);
  payoff = (p->Compute)(p->Par,S_T);
  sum += payoff;
  sum2+=SQR(payoff);
    }

  *ptprice = (sum / (double)iters);
  *pterror_price = exp(-r * T) * sqrt(sum2 / (double)iters - SQR(*ptprice)) / sq
  *ptprice = exp(-r * T) * (*ptprice);

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

  free(v);
  free(PreE_Ic);
  free(PreVar_Ic);


  return init_mc;
}

int CALC(MC_TseWan)(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 MCTseWan(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_LONG, Met->Par[1].Val.V_ENUM.value,Met->Par[2].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
));

}

static int CHK_OPT(MC_TseWan)(void *Opt, void *Mod)
{

  if ((strcmp(((Option *)Opt)->Name, "CallEuro") == 0) || (strcmp(((Option *)Opt
    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 = 100000;
      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->HelpFilenameHint = "mc_tsewan";
```

```
    }
  return OK;
}

PricingMethod MET(MC_TseWan) =
{
  "MC_TseWan",
  { {"N iterations", LONG, {100}, ALLOW},
    {"RandomGenerator", ENUM, {100}, ALLOW},
    {"Confidence Value", DOUBLE, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_TseWan),
  { {"Price", DOUBLE, {100}, FORBID},
{"Error Price", DOUBLE, {100}, FORBID},
   {"Inf Price", DOUBLE, {100}, FORBID},
    {"Sup Price", DOUBLE, {100}, FORBID} ,
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(MC_TseWan),
  CHK_mc,
  MET(Init)
};
}
```