Help

```
#include <stdlib.h>
#include  "
href../../../mod/doublehes1d/doublehes1d_std/doublehes1d_std_h_src.pdfhes1d_std.
#include "
href../../../common/enums_h_src.pdfenums.h"

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


// Function that will be used in L2
static double psi_function(double h, double k)
{
  if (k == 0) return h;
  else return (1. - exp(-k * h)) / k;
}


// Simulation according to the generator L_1
static  double L1(double x, double h, int n, double a, double sigma)
{
  double x_new;
  x_new = x + (a - (double)n * sigma * sigma / 4.) * h;
  return x_new;
}

// Simulation according to the generator L_2
static  double L2(double x, double h, int n, double sigma, double k, int generat
{
  double *Y, *Z, psi, x_new = 0.;
  int i;
```

```
  Y = malloc(n * sizeof(double *));
  Z = malloc(n * sizeof(double *));

  Y[0] = sqrt(x);

  for (i = 1; i < n; i++)
    {
      Y[i] = 0.;
    }

  psi = psi_function(h, k);

  for (i = 0; i < n; i++)
    {
      Z[i] = Y[i] * exp(-k * 0.5 * h) + sigma * 0.5 * sqrt(psi) * pnl_rand_norma
    }

  for (i = 0; i < n; i++)
    {
      x_new = x_new + Z[i] * Z[i];
    }

  free(Y);
  free(Z);

  return x_new;
}


//Simulation CIR at time t_i+1 starting from a simulation at time t_i
//following the composition rule p_2(t/2)*p_1(t)*p_2(t/2)
//that is according to the scheme q_2 in the paper
static double CIR(double x, double h, int n, double a, double k, double sigma, i
{
  double x_new, x_tmp1, x_tmp2;
  x_tmp1 = L2(x, h * 0.5, n, sigma, k, generator);
  x_tmp2 = L1(x_tmp1, h, n, a, sigma);
  x_new = L2(x_tmp2, h * 0.5, n, sigma, k, generator);
  return x_new;
}
```

```
//Function associated to the generator W in [Alfonsi] paper
static void HW(double *x1, double *x2, double *x3, double *x4, double *delta_x1,
{
  double app;
  app = CIR(*x1, h, n, a, k, sigma, generator);
  (*delta_x1) = -(*x1) + app; //here I "save" the value x1[i+1]-x[i]
  (*x2) = (*x2) + (*x1 + 0.5 * (*delta_x1)) * h;
  (*x4) = (*x4) + 0.5 * (*x3) * h;
  (*x3) = (*x3) * exp((r - divid - rho * a / sigma) * h + rho * (*delta_x1) / si
  (*x4) = (*x4) + 0.5 * (*x3) * h;
  (*x1) = (*x1) + (*delta_x1);
}

//Function associated to the generator Z in [Alfonsi] paper
static void Zgenerator( double *x1, double *x3, double h,double rho,int generato
{
  (*x3) = (*x3) * exp(sqrt((1. - rho * rho) * (*x1) * h) * pnl_rand_normal(gener
}


int MCBaldiPisani(double S_0, NumFunc_1  *p, double T, double r, double divid, d
{
  long i, ipath;
  double price_sample, delta_sample, mean_price, mean_delta, var_price, var_delt
  int init_mc;
  int simulation_dim;
  double alpha, z_alpha;
  double x1, x2, x3, x4, delta_x1, S_T;
  double h;
  double a;
  int n;

  //Time Step
  h = T / nn;

  a = theta * k;
  n = floor(4.*a / (sigma * sigma));

  /* 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;

/* Size of the random vector we need in the simulation */
simulation_dim = nn;

/* MC sampling */
init_mc = pnl_rand_init(generator, simulation_dim, nb);
/* Test after initialization for the generator */
if (init_mc == OK)
  {

    for (ipath = 1; ipath <= nb; ipath++)
      {
        //initialization
        x1 = v_0;
        x2 = 0.;
        x3 = S_0;
        x4 = 0.;

        //simulation of X_T
        for (i = 0; i < nn; i++)
          {

            if (pnl_rand_uni(generator) < 0.5)
              {
                HW(&x1, &x2, &x3, &x4, &delta_x1, h, n, a, r, divid, k, sigma,
                Zgenerator(&x1, &x3, h, rho, generator);
              }
            else
              {
                Zgenerator(&x1, &x3, h, rho, generator);
                HW(&x1, &x2, &x3, &x4, &delta_x1, h, n, a, r, divid, k, sigma,
              }
          }
```

```
      /*Price*/
      S_T = x3;
      price_sample = (p->Compute)(p->Par, x3);

      /* Delta */
      if (price_sample > 0.0)
        delta_sample = (S_T / S_0);
      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)nb);
  *pterror_price = exp(-r * T) * sqrt(var_price / (double)nb - SQR(*ptprice)
  *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)nb);
  if ((p->Compute) == &Put)
    *ptdelta *= (-1);
  *pterror_delta = sqrt(exp(-2.0 * r * T) * (var_delta / (double)nb - SQR(*p

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

```
  return init_mc;
}

int CALC(MC_BaldiPisani_Heston)(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 MCBaldiPisani(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_INT,
                       Met->Par[2].Val.V_ENUM.value,
                       Met->Par[3].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_BaldiPisani_Heston)(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_INT = 20;
      Met->Par[2].Val.V_ENUM.value = 0;
      Met->Par[2].Val.V_ENUM.members = &PremiaEnumMCRNGs;
      Met->Par[3].Val.V_DOUBLE = 0.95;
    }


  return OK;
}


PricingMethod MET(MC_BaldiPisani_Heston) =
{
  "MC_BaldiPisani",
  { {"N iterations", LONG, {100}, ALLOW},
    {"TimeStepNumber", LONG, {100}, ALLOW},
    {"RandomGenerator", ENUM, {100}, ALLOW},
    {"Confidence Value", DOUBLE, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_BaldiPisani_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_BaldiPisani_Heston),
  CHK_mc,
  MET(Init)
};
```