Help

```
#include  "merhes1d_std.h"
#include "enums.h"

#include "pnl/pnl_random.h"
#include "pnl/pnl_cdf.h"

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

static double **V;
static double **f;
static int **f_down,**f_up;
static double **pu_f,**pd_f;


/*Memory Allocation*/
static int memory_allocation(int Nt)
{
  int i;

  V=(double**)calloc(Nt+1,sizeof(double*));
  if (V==NULL)
    return MEMORY_ALLOCATION_FAILURE;
  for (i=0;i<Nt+1;i++)
    {
      V[i]=(double *)calloc(Nt+1,sizeof(double));
      if (V[i]==NULL)
        return MEMORY_ALLOCATION_FAILURE;
    }
```

```
pu_f=(double**)calloc(Nt+1,sizeof(double*));
if (pu_f==NULL)
  return MEMORY_ALLOCATION_FAILURE;
for (i=0;i<Nt+1;i++)
  {
    pu_f[i]=(double *)calloc(Nt+1,sizeof(double));
    if (pu_f[i]==NULL)
      return MEMORY_ALLOCATION_FAILURE;
  }

pd_f=(double**)calloc(Nt+1,sizeof(double*));
if (pd_f==NULL)
  return MEMORY_ALLOCATION_FAILURE;
for (i=0;i<Nt+1;i++)
  {
    pd_f[i]=(double *)calloc(Nt+1,sizeof(double));
    if (pd_f[i]==NULL)
      return MEMORY_ALLOCATION_FAILURE;
  }


f=(double**)calloc(Nt+1,sizeof(double*));
if (f==NULL)
  return MEMORY_ALLOCATION_FAILURE;
for (i=0;i<Nt+1;i++)
  {
    f[i]=(double *)calloc(Nt+1,sizeof(double));
    if (f[i]==NULL)
      return MEMORY_ALLOCATION_FAILURE;
  }

f_down=(int**)calloc(Nt+1,sizeof(int*));
if (f_down==NULL)
  return MEMORY_ALLOCATION_FAILURE;
for (i=0;i<Nt+1;i++)
  {
    f_down[i]=(int *)calloc(Nt+1,sizeof(int));
    if (f_down[i]==NULL)
      return MEMORY_ALLOCATION_FAILURE;
  }
```

```
  f_up=(int**)calloc(Nt+1,sizeof(int*));
  if (f_up==NULL)
    return MEMORY_ALLOCATION_FAILURE;
  for (i=0;i<Nt+1;i++)
    {
      f_up[i]=(int *)calloc(Nt+1,sizeof(int));
      if (f_up[i]==NULL)
        return MEMORY_ALLOCATION_FAILURE;
    }

  return OK;
}


static void free_memory(int Nt)
{
  int i;

  for (i=0;i<Nt+1;i++)
    free(V[i]);
  free(V);

  for (i=0;i<Nt+1;i++)
    free(pu_f[i]);
  free(pu_f);

  for (i=0;i<Nt+1;i++)
    free(pd_f[i]);
  free(pd_f);

  for (i=0;i<Nt+1;i++)
    free(f[i]);
  free(f);

  for (i=0;i<Nt+1;i++)
    free(f_up[i]);
  free(f_up);

  for (i=0;i<Nt+1;i++)
    free(f_down[i]);
  free(f_down);
```

```
  return;
}

static double compute_f(double r, double omega)
{
  return 2.*sqrt(r) / omega;
}

static double compute_v(double R, double omega)
{
  double val;

  val = SQR(R) * SQR(omega) / 4.;
  if (R > 0.)
    val = SQR(R) * SQR(omega) / 4.;
  else
    val = 0.0;
  return val;
}


/*Calibration of the tree  v*/
static int tree_v(double tt, double v02, double kappa2, double theta2, double om
{
  int i, j;
  int z;
  double Ru, Rd;
  double mu_r, v_curr;
  double dt, sqrt_dt;

  /*Fixed tree for R=f*/
  f[0][0] = compute_f(v02, omega2);

  dt = tt / (double)Nt;
  sqrt_dt = sqrt(dt);

  V[0][0] = compute_v(f[0][0], omega2);
  f[1][0] = f[0][0] - sqrt_dt;
  f[1][1] = f[0][0] + sqrt_dt;
  V[1][0] = compute_v(f[1][0], omega2);
```

```
V[1][1] = compute_v(f[1][1], omega2);
for (i = 1; i < Nt; i++)
  for (j = 0; j <= i; j++)
    {
      f[i + 1][j] = f[i][j] - sqrt_dt;
      f[i + 1][j + 1] = f[i][j] + sqrt_dt;
      V[i + 1][j] = compute_v(f[i + 1][j], omega2);
      V[i + 1][j + 1] = compute_v(f[i + 1][j + 1], omega2);
    }

for (i = 1; i < Nt; i++)
  for (j = 0; j <= i; j++)

    /*Evolve tree for f*/
    for (i = 0; i < Nt; i++)
      {
        for (j = 0; j <= i; j++)
          {
            /*Compute mu_f*/
            v_curr = V[i][j];

            mu_r = kappa2 * (theta2 - v_curr);

            z = 0;
            while ((V[i][j] + mu_r * dt < V[i + 1][j - z])
                   && (j - z >= 0))
              {

                z = z + 1;
              }
            f_down[i][j] = -z;
            Rd = V[i + 1][j - z];

            if (z > 0)
              z = 0;
            else z = 1;

            while ((V[i][j] + mu_r * dt > V[i + 1][j + z])
                   && (j + z <= i))
              {
                z = z + 1;
```

```
                }

              Ru = V[i + 1][j + z];

              f_up[i][j] = z;
              pu_f[i][j] = (V[i][j] + mu_r * dt - Rd) / (Ru - Rd);

              if ((Ru - 1.e-9 > V[i + 1][i + 1]) || (j + f_up[i][j] > i + 1))
                {
                  pu_f[i][j] = 1;

                  f_up[i][j] = i + 1 - j;
                  f_down[i][j] = i - j;
                }
              if ((Rd + 1.e-9 < V[i + 1][0]) || (j + f_down[i][j] < 0))
                {
                  pu_f[i][j] = 0.;
                  f_up[i][j] = 1 - j;
                  f_down[i][j] = 0 - j;
                }
              pd_f[i][j] = 1. - pu_f[i][j];
            }
        }
  return 1;
}

/*Compute Price Option*/
int  MCHybridTree_Bates(double s0, NumFunc_1  *p, double tt, double r,double div
{

  double price_sample, mean_price,var_price;
  int init_mc,ipath,i,k2,j,nj;
  double g,poisson_jump;
  int simulation_dim;
  double alpha, z_alpha;
  double pterror_price;
  double vol;
  double dt,sqrt_dt;
  double g1,w_t_1;
  double Yt,St;
  double rho3;
```

```
  double new_vol;
  double mu_z;
  double sigma_z;
  double Eu;

  Eu = exp(mu + 0.5 * gamma2) - 1.;

  /*Memory Allocation*/
  if (memory_allocation(Nt) != OK) return FAIL;

  //Tree construction for v
  tree_v(tt, v02, kappa2, theta2, omega2, Nt);

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

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

  dt=tt/(double)Nt;
  sqrt_dt=sqrt(dt);
  /* Test after initialization for the generator */
  simulation_dim =Nt;

  /* MC sampling */
  init_mc = pnl_rand_init(generator, simulation_dim, N_MC);
  rho3=sqrt(1.-SQR(rhoSv));
  for(ipath= 1;ipath<=N_MC;ipath++)
{
  vol=V[0][0];
  Yt=log(s0);
  k2=0;
  for(i=0;i<Nt;i++)
{
  g1=pnl_rand_normal(generator);//V
  w_t_1=sqrt_dt*g1;//V

  //Simulate V
  if(pnl_rand_uni(generator)<pu_f[i][k2])
```

```
k2+=f_up[i][k2];
  else
k2+=f_down[i][k2];

  new_vol=V[i+1][k2];

  mu_z=r-divid-0.5*vol-rhoSv/omega2*kappa2*(theta2-vol)-lambda*Eu;
  sigma_z=rho3*sqrt(vol);

  Yt+=mu_z*dt+sigma_z*w_t_1+rhoSv/omega2*(new_vol-vol);

   /* Jump */
          nj = pnl_rand_poisson(lambda * dt, generator);

          poisson_jump = 1.;
          for (j = 1; j <= nj; j++)
            {
              g = pnl_rand_normal(generator);
              poisson_jump *= exp(mu + sqrt(gamma2) * g);
            }

  Yt+=log(poisson_jump);

  St=exp(Yt);

  vol=new_vol;

}

  price_sample=exp(-r*tt)*(p->Compute)(p->Par,St);

  /* Sum */
  mean_price+=price_sample;

  /* Sum of squares */
  var_price+= SQR(price_sample);

}

  *ptprice=(mean_price/(double)N_MC);
  pterror_price= sqrt(var_price/(double)N_MC-SQR(*ptprice))/sqrt((double)N_MC-1)
```

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

  /*Memory Disallocation*/
  free_memory(Nt);

  return OK;
}

int CALC(MC_HybridTree_Bates)(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 MCHybridTree_Bates(ptMod->S0.Val.V_PDOUBLE,
                            ptOpt->PayOff.Val.V_NUMFUNC_1,
                            ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,r,
                            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,
  ptMod->Lambda.Val.V_PDOUBLE, ptMod->Mean.Val.V_PDOUBLE, ptMod->Variance.Val.V_
                            Met->Par[0].Val.V_PINT,
                            Met->Par[1].Val.V_PINT,
 Met->Par[2].Val.V_ENUM.value,
 Met->Par[3].Val.V_PDOUBLE,
                            &(Met->Res[0].Val.V_DOUBLE), &(Met->Res[1].Val.V_DO
}

static int CHK_OPT(MC_HybridTree_Bates)(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)
{
  if (Met->init == 0)
    {
      Met->init = 1;
      Met->HelpFilenameHint = "mc_hybridtree_bates";
      Met->Par[0].Val.V_INT = 100;
      Met->Par[1].Val.V_INT = 50000;
  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_HybridTree_Bates) =
{
  "MC_HybridTree",
  { {"N steps time", INT, {100}, ALLOW},
    {"N Iterations", INT, {100}, ALLOW},
{"RandomGenerator", ENUM, {100}, ALLOW},
    {"Confidence Value", DOUBLE, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_HybridTree_Bates),
  { {"Price", DOUBLE, {100}, FORBID},
 {"Inf Price", DOUBLE, {100}, FORBID},
    {"Sup Price", DOUBLE, {100}, FORBID} ,
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(MC_HybridTree_Bates),
  CHK_mc,
  MET(Init)
};
```