Help

```
#include  "
href../../../mod/bshw1d/bshw1d_std/bshw1d_std_h_src.pdfbshw1d_std.h"
#include "
href../../../common/enums_h_src.pdfenums.h"
#include "
href../../../common/error_msg_h_src.pdferror_msg.h"

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

static double **Q, * *P_old, * *P_new;
static double **y;
static int **y_down, * *y_up;
static double **pu_y, * *pd_y;
static double **r, * *discount;
static double *initial_yield;

/*ZCB Data*/
static double *tm; /*Times T of maturities read in the file initialyield.dat */
static double *Pm;/*Values of the zero coupon P(0,tm) read in the file initialyi

static double *shift, *Pc;
static char *init_tr;

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

  shift = (double *)malloc((Nt + 1) * sizeof(double));
  initial_yield = (double *)malloc((Nt + 2) * sizeof(double));
```

```
Pc = (double *)malloc((Nt + 2) * sizeof(double));

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

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

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

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

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

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

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

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

```
  P_old = (double **)malloc((N + 1) * sizeof(double *));
  for (i = 0; i <= N; i++)
    P_old[i] = (double *)malloc((Nt + 1) * sizeof(double));

  P_new = (double **)malloc((N + 1) * sizeof(double *));
  for (i = 0; i <= N; i++)
    P_new[i] = (double *)malloc((Nt + 1) * sizeof(double));
  return OK;
}


static void free_memory(int Nt, int N)
{
  int i;

  free(shift);
  free(initial_yield);
  free(Pc);

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

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

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

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

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

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

```
  free(y);

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

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

  for (i = 0; i < N + 1; i++)
    free(P_old[i]);
  free(P_old);

  for (i = 0; i < N + 1; i++)
    free(P_new[i]);
  free(P_new);

  return;
}


/*Calibration of the tree  the interest rate r Hwicek model*/
static void tree_r(double tt, double r0, double kappa, double omega, int Nt)
{
  int i, j;
  int z;
  double Ru, Rd;
  double dt, sqrt_dt;
  double mu_r, v_curr;

  y[0][0] = 0.;

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

  y[1][0] = y[0][0] - sqrt_dt;
  y[1][1] = y[0][0] + sqrt_dt;

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

```
    {
      y[i + 1][j] = y[i][j] - sqrt_dt;
      y[i + 1][j + 1] = y[i][j] + sqrt_dt;
    }

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

        mu_r = -kappa * v_curr;

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

            z = z + 1;
          }
        y_down[i][j] = -z;
        Rd = y[i + 1][j - z];


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

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

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

        y_up[i][j] = z;

        pu_y[i][j] = (y[i][j] + mu_r * dt - Rd) / (Ru - Rd);
```

```
          if ((Ru - 1.e-9 > y[i + 1][i + 1]) || (j + y_up[i][j] > i + 1))
            {
              pu_y[i][j] = 1;

              y_up[i][j] = i + 1 - j;
              y_down[i][j] = i - j;
            }
          if ((Rd + 1.e-9 < y[i + 1][0]) || (j + y_down[i][j] < 0))
            {
              pu_y[i][j] = 0.;
              y_up[i][j] = 1 - j;
              y_down[i][j] = 0 - j;
            }
          pd_y[i][j] = 1. - pu_y[i][j];
        }
    }
}

static int lecture_tr()
{

  int i;
  char ligne[30];
  char *pligne;
  double p, tt_value;
  FILE *Entrees;

  Entrees = fopen(init_tr, "r");

  if (Entrees == NULL)
    {
      printf("Le FICHIER N'A PU ETRE OUVERT. VERIFIER LE CHEMIN\ n");
    }

  /* i is the number oflibe that has been read */
  i = 0;
  pligne = ligne;
  Pm = (double *)malloc(200 * sizeof(double));
  tm = (double *)malloc(200 * sizeof(double));
```

```
  while (1)
    {
      pligne = fgets(ligne, sizeof(ligne), Entrees);
      if (pligne == NULL) break;
      else
        {
          sscanf(ligne, "%lf t=%lf", &p, &tt_value);
          /* The line read must be written "0.943290 t=0.5" where 0.943290 is a
          Pm[i] = p; /*save the price of the zero coupon*/
          tm[i] = tt_value; /*save the corresponding time*/
          i++;

        }
    }
  fclose(Entrees);

  return i;
}

static void interpole(int n_price, int imax, double *t)
{
  int i, iF, j;

  n_price--;

  i = 0;
  while (t[i] <= tm[1])
    {
      initial_yield[i] = (tm[1] - t[i]) / tm[1] + t[i] * Pm[1] / tm[1];
      i++;
    }

  for (j = 0; j < n_price; j++)
    {
      while (t[i] < tm[j + 1] && i <= imax + 1)
        {
          initial_yield[i] = (tm[j + 1] - t[i]) / (tm[j + 1] - tm[j]) * Pm[j] +
          i++;
        }
    }
```

```
  if (t[i] > tm[n_price] && i <= imax + 1)
    {
      for (iF = i ; iF <= imax ; iF++)
        {
          initial_yield[iF] = Pm[n_price] + (Pm[n_price] - Pm[n_price - 1]) / (t
        }
    }
}

/*Calibration of the tree consistent with dynamic of the Hull-White Process*/
static void calibration_bond(int flat_flag, double tt, double r0, double omega,
{
  double sum;
  int i, j, jj, n_price;
  double dt;

  dt = tt / (double)Nt;

  /*Initialilise Yield Curve*/
  if (flat_flag == 0)
    {
      for (i = 0; i <= Nt + 1; i++)
        initial_yield[i] = r0;
    }
  else
    {
      double *t_vect;
      t_vect = (double *)malloc((Nt + 3) * sizeof(double));

      for (i = 0; i <= Nt + 2; i++)
        t_vect[i] = i * dt;
      n_price = lecture_tr();
      /* We search in initialyield.dat the biggest value before time T */
      if (tt > tm[n_price - 1])
        {
          printf("\ nError : time bigger than the last time value entered in ini
        }
      interpole(n_price, Nt, t_vect);

      free(tm);
      free(Pm);
```

```
    free(t_vect);
  }

for (i = 0; i <= Nt + 1; i++)
  {
    if (flat_flag == 0)
      {
        Pc[i] = exp(-initial_yield[i] * i * dt);
      }
    else
      {
        Pc[i] = initial_yield[i];
      }
  }

/*Initalise first node*/
Q[0][0] = 1.;

/*Evolve tree for the x=ln r*/
for (i = 0; i <= Nt; i++)
  {
    /*Update pure security prices*/
    if (i > 0)
      for (j = 0; j <= i; j++)
        {
          sum = 0.;

          for (jj = 0; jj <= i - 1; jj++)
            {
              if (jj + y_up[i - 1][jj] == j)
                sum += Q[i - 1][jj] * pu_y[i - 1][jj] * discount[i - 1][jj];
              if (jj + y_down[i - 1][jj] == j)
                sum += Q[i - 1][jj] * pd_y[i - 1][jj] * discount[i - 1][jj];
            }

          Q[i][j] = sum;
        }

    /*Compute shift a[i]*/
    if (i == 0)
      shift[0] = -log(Pc[1]) / dt;
```

```
      else
        {
          sum = 0.;
          for (j = 0; j <= i; j++)
            sum += Q[i][j] * exp(-omega * y[i][j] * dt);

          shift[i] = (log(sum) - log(Pc[i + 1])) / dt;
        }

      /*Compute x,r and discount factor d*/
      for (j = 0; j <= i; j++)
        {
          r[i][j] = omega * y[i][j] + shift[i];
          discount[i][j] = exp(-r[i][j] * dt);
        }
    }
}

static double compute_S(double Y, double rv, double sigma, double omega, double
{
  double val;

  val = exp(Y + rho * sigma * rv);

  return val;
}

/*Compute Price Option*/
int  Fd_HybridTree_BsHw(int am, double s0, NumFunc_1  *p, double tt, double divi
{
  int i, j;
  double stock;
  int fv_up, fv_down;
  double l;
  double alpha, beta, gamma, alpha1, beta1, gamma1;
  int PriceIndex;
  double dx;
  int k1;
  double  *A, *B, *C, *A1, *B1, *C1, *Price, *S, *vect_y;
  double dt;
  double z, vv, sigma2;
```

```
double bound1, bound2;
double y0;
int Index;
int dummy;
double PRECISION_FDH=1.0e-5;

init_tr = curve;

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

//Tree construction for r
tree_r(tt, r0, kappa, omega, Nt);
calibration_bond(flat_flag, tt, r0, omega, Nt);

//Memory allocation
A = (double *)malloc((N + 1) * sizeof(double));
B = (double *)malloc((N + 1) * sizeof(double));
C = (double *)malloc((N + 1) * sizeof(double));
A1 = (double *)malloc((N + 1) * sizeof(double));
B1 = (double *)malloc((N + 1) * sizeof(double));
C1 = (double *)malloc((N + 1) * sizeof(double));
vect_y = (double *)malloc((N + 1) * sizeof(double));
Price = (double *)malloc((N + 1) * sizeof(double));
S = (double *)malloc((N + 1) * sizeof(double));


dt = tt / (double)Nt;
sigma2 = SQR(sigma);

 l=sigma*sqrt(tt)*sqrt(log(1.0/PRECISION_FDH))+fabs((r0-divid-0.5*sigma2)*tt);

 y0=log(s0)-sigma*rho/omega*r[0][0];
dx = 2 * l / (double)N;

for (j = 0; j <= N; j++)
  {
    vect_y[j] = y0 - l + (double)j * dx;
  }
```

```
/*Maturity conditions for Call options*/
for (k1 = 0; k1 <= Nt; k1++)
  for (j = 0; j <= N; j++)
    {
      stock = compute_S(vect_y[j], y[Nt][k1], sigma, omega, rho);
      P_old[j][k1] = (p->Compute)(p->Par, stock);
    }
bound1 = 0.;
bound2 = 0.;

/*Rhs Factor of theta-schema*/
alpha1 = 0.;
beta1 = 1.;
gamma1 = 0.;

for (PriceIndex = 1; PriceIndex <= N - 1; PriceIndex++)
  {
    A1[PriceIndex] = alpha1;
    B1[PriceIndex] = beta1;
    C1[PriceIndex] = gamma1;
  }

/*Dynamic Programming*/
for (i = Nt - 1; i >= 0; i--)
  {
    for (k1 = 0; k1 <= i; k1++)
      {
        z = r[i][k1] - divid - 0.5 * sigma2 + sigma * rho * kappa * y[i][k1];
        vv = 0.5 * SQR(sigma) * (1. - SQR(rho));

        /*Lhs Factor of the fully implicit scheme*/
        alpha = -vv * dt / SQR(dx) + z * dt / (2.*dx);
        beta = 1 + vv * 2 * dt / SQR(dx);
        gamma = -vv * dt / SQR(dx) - z * dt / (2 * dx);

        for (PriceIndex = 1; PriceIndex <= N - 1; PriceIndex++)
          {
            A[PriceIndex] = alpha;
            B[PriceIndex] = beta;
            C[PriceIndex] = gamma;
          }
```

```
B[1] = beta + alpha;
B[N - 1] = beta + gamma;

/*Rhs Factors*/
alpha1 = 0.;
beta1 = 1.;
gamma1 = 0.;

for (PriceIndex = 1; PriceIndex <= N - 1; PriceIndex++)
  {
    A1[PriceIndex] = alpha1;
    B1[PriceIndex] = beta1;
    C1[PriceIndex] = gamma1;
  }

/*Set Gauss*/
for (PriceIndex = N - 2; PriceIndex >= 1; PriceIndex--)
  B[PriceIndex] = B[PriceIndex] - C[PriceIndex] * A[PriceIndex + 1] /
for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
  A[PriceIndex] = A[PriceIndex] / B[PriceIndex];
for (PriceIndex = 1; PriceIndex < N - 1; PriceIndex++)
  C[PriceIndex] = C[PriceIndex] / B[PriceIndex + 1];

fv_up = y_up[i][k1];
fv_down = y_down[i][k1];

//Initialise
for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
  {
    Price[PriceIndex] = pu_y[i][k1] *P_old[PriceIndex][k1 + fv_up]+pd_
  }

/*Set Rhs*/
S[1] = B1[1] * Price[1] + C1[1] * Price[2] + A1[1] * bound1 - alpha *
for (PriceIndex = 2; PriceIndex < N - 1; PriceIndex++)
  S[PriceIndex] = A1[PriceIndex] * Price[PriceIndex - 1] +
                  B1[PriceIndex] * Price[PriceIndex] +
                  C1[PriceIndex] * Price[PriceIndex + 1];
S[N - 1] = A1[N - 1] * Price[N - 2] + B1[N - 1] * Price[N - 1] + C1[N

/*Solve the system*/
```

```
        for (PriceIndex = N - 2; PriceIndex >= 1; PriceIndex--)
          S[PriceIndex] = S[PriceIndex] - C[PriceIndex] * S[PriceIndex + 1];

        Price[1] = S[1] / B[1];

        for (PriceIndex = 2; PriceIndex < N; PriceIndex++)
          Price[PriceIndex] = S[PriceIndex] / B[PriceIndex] - A[PriceIndex] *

        for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
          {
            P_new[PriceIndex][k1] = discount[i][k1] *  Price[PriceIndex];
          }

        if (am)
          for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
            P_new[PriceIndex][k1] = MAX(P_new[PriceIndex][k1], (p->Compute)(p-


      }//end k1

    //Copy
    for (k1 = 0; k1 <= i; k1++)
      for (j = 0; j <= N; j++)
        {
          P_old[j][k1] = P_new[j][k1];
        }
  }//end i


Index = (int) floor((double)N / 2.0);
//Price
*ptprice = P_new[Index][0];

/*Memory Disallocation*/
free_memory(Nt, N);
free(A);
free(B);
free(C);
free(A1);
free(B1);
free(C1);
```

```
  free(vect_y);
  free(S);
  free(Price);

  return OK;
}

int CALC(FD_HybridTree)(void *Opt, void *Mod, PricingMethod *Met)
{
  TYPEOPT *ptOpt = (TYPEOPT *)Opt;
  TYPEMOD *ptMod = (TYPEMOD *)Mod;
  double divid;

  divid = log(1. + ptMod->divid.Val.V_DOUBLE / 100.);
  return Fd_HybridTree_BsHw(ptOpt->EuOrAm.Val.V_BOOL, ptMod->S0.Val.V_PDOUBLE,
                            ptOpt->PayOff.Val.V_NUMFUNC_1,
                            ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE, di
                            ptMod->flat_flag.Val.V_INT,
                            MOD(GetYield)(ptMod),
                            MOD(GetCurve)(ptMod),
                            ptMod->kr.Val.V_PDOUBLE,
                            ptMod->Sigmar.Val.V_PDOUBLE,

                            ptMod->RhoSr.Val.V_PDOUBLE,
                            Met->Par[0].Val.V_PINT,
                            Met->Par[1].Val.V_PINT,
                            &(Met->Res[0].Val.V_DOUBLE));
}

static int CHK_OPT(FD_HybridTree)(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 = "fd_hybridtree_bshw";
      Met->Par[0].Val.V_INT = 100;
      Met->Par[1].Val.V_INT = 100;
    }

  return OK;
}

PricingMethod MET(FD_HybridTree) =
{
  "FD_HybridTree",
  { {"N steps time", INT, {100}, ALLOW},
    {"N steps space", INT, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(FD_HybridTree),
  { {"Price", DOUBLE, {100}, FORBID},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(FD_HybridTree),
  CHK_ok,
  MET(Init)
};
```