Help

```
#include  "
href../../../mod/heshw2d/heshw2d_std/heshw2d_std_h_src.pdfheshw2d_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


/*Vettori*/
static double *vect_z;
static double ****P1;
static int current_index;
static double xa[3], ya[3];

static double **V, * *Q, * ** *P, * ** *P1;
static double **y, * *f, * *y_rf;
static int **f_down, * *f_up;
static int **y_down, * *y_up;
static int **rf_down, * *rf_up;
static double **pu_y, * *pd_y;
static double **pu_f, * *pd_f;
static double **pu_rf, * *pd_rf;
static double **r, * *rf, * *discount, * *discount_rf, *Pc;
static double *shift_r, *shift_rf;
static double *initial_yield;
static char *init_tr_rd, *init_tr_rf;
```

```
/*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


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

  shift_r = malloc((Nt + 1) * sizeof(double));
  shift_rf = malloc((Nt + 1) * sizeof(double));
  initial_yield = malloc((Nt + 2) * sizeof(double));
  Pc = 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;
    }

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

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

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;
  }

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;
  }

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

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

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

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;
  }
```

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

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

P = (double ** **)malloc((N + 1) * sizeof(double ** *));
for (i = 0; i <= N; i++)
  P[i] = (double ** *)malloc((Nt + 1) * sizeof(double **));
for (i = 0; i <= N; i++)
  for (j = 0; j <= Nt; j++)
    P[i][j] = (double **)malloc((Nt + 1) * sizeof(double *));
for (i = 0; i <= N; i++)
  for (j = 0; j <= Nt; j++)
    for (k = 0; k <= Nt; k++)
      P[i][j][k] = (double *)malloc((Nt + 1) * sizeof(double));

P1 = (double ** **)malloc((N + 1) * sizeof(double ** *));
for (i = 0; i <= N; i++)
  P1[i] = (double ** *)malloc((Nt + 1) * sizeof(double **));
for (i = 0; i <= N; i++)
  for (j = 0; j <= Nt; j++)
    P1[i][j] = (double **)malloc((Nt + 1) * sizeof(double *));
for (i = 0; i <= N; i++)
  for (j = 0; j <= Nt; j++)
    for (k = 0; k <= Nt; k++)
```

```
        P1[i][j][k] = (double *)malloc((Nt + 1) * sizeof(double));

  return OK;
}

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

  free(shift_r);
  free(shift_rf);
  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(rf[i]);
  free(rf);

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

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

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

  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 < Nt + 1; i++)
  free(y_rf[i]);
free(y_rf);

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(pu_rf[i]);
free(pu_rf);

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

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);

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

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

  for (i = 0; i < N + 1; i++)
    for (j = 0; j < Nt + 1; j++)
      for (k = 0; k < Nt + 1; k++)
        free(P[i][j][k]);
  for (i = 0; i < N + 1; i++)
    for (j = 0; j < Nt + 1; j++)
      free(P[i][j]);
  for (j = 0; j < N + 1; j++)
    free(P[j]);
  free(P);

  for (i = 0; i < N + 1; i++)
    for (j = 0; j < Nt + 1; j++)
      for (k = 0; k < Nt + 1; k++)
        free(P1[i][j][k]);
  for (i = 0; i < N + 1; i++)
    for (j = 0; j < Nt + 1; j++)
      free(P1[i][j]);
  for (j = 0; j < N + 1; j++)
    free(P1[j]);
  free(P1);

  return ;
}
```

```
static int lecture_tr(int flag_rd)
{

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

  if (flag_rd)
    Entrees = fopen(init_tr_rd, "r");
  else
    Entrees = fopen(init_tr_rf, "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 = malloc(200 * sizeof(double));
  tm = 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 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 theta, double ome
{
  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 * (theta - 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;
}


/*Calibration of the tree  the interest rate r Hwicek model*/
static int tree_rd(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];
        }
    }

  return 1;
}

/*Calibration of the tree  the interest rate r foreign Hwicek model*/
static int tree_rf(double tt, double r0f, double kappa_rf, double omega_rf, int
{
  int i, j;
  int z;
  double Ru, Rd;
  double dt, sqrt_dt;
  double mu_rf, v_curr;

  y_rf[0][0] = 0.;

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

```
y_rf[1][0] = y_rf[0][0] - sqrt_dt;
y_rf[1][1] = y_rf[0][0] + sqrt_dt;

for (i = 1; i < Nt; i++)
  for (j = 0; j <= i; j++)
    {
      y_rf[i + 1][j] = y_rf[i][j] - sqrt_dt;
      y_rf[i + 1][j + 1] = y_rf[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_rf[i][j];

        mu_rf = -kappa_rf * v_curr;

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

            z = z + 1;
          }
        rf_down[i][j] = -z;
        Rd = y_rf[i + 1][j - z];

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

        while ((y_rf[i][j] + mu_rf * dt > y_rf[i + 1][j + z])
               && (j + z <= i))
          {
            z = z + 1;
          }
```

```
          Ru = y_rf[i + 1][j + z];

          rf_up[i][j] = z;
          pu_rf[i][j] = (y_rf[i][j] + mu_rf * dt - Rd) / (Ru - Rd);

          if ((Ru - 1.e-9 > y_rf[i + 1][i + 1]) || (j + rf_up[i][j] > i + 1))
            {
              pu_rf[i][j] = 1;

              rf_up[i][j] = i + 1 - j;
              rf_down[i][j] = i - j;
            }
          if ((Rd + 1.e-9 < y_rf[i + 1][0]) || (j + rf_down[i][j] < 0))
            {
              pu_rf[i][j] = 0.;
              y_up[i][j] = 1 - j;
              rf_down[i][j] = 0 - j;
            }
          pd_rf[i][j] = 1. - pu_rf[i][j];
        }
    }

  return 1;
}


static int 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

        }
    }

  return 1;
}


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

  dt = tt / (double)Nt;

  /*Initialilise Yield Curve*/
  if (flat_flag == 0)
    {
      for (i = 1; 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;

    flag_rd = 0;

    n_price = lecture_tr(flag_rd);

    /* 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 = 1; 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 a[i]*/
      if (i == 0)
        shift_r[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_r[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_r[i];
          discount[i][j] = exp(-r[i][j] * dt);
        }
    }
  return 1;
}


/*Calibration of the tree consistent with dynamic of the Hull-White Process*/
```

```
static int calibration_bond_rf(int flat_flag, double tt, double r0f, double omeg
{
  double sum;
  int i, j, jj, n_price;
  double dt;
  int flag_rd;

  dt = tt / (double)Nt;

  /*Initialilise Yield Curve*/
  if (flat_flag == 0)
    {
      for (i = 1; i <= Nt + 1; i++)
        initial_yield[i] = r0f;
    }
  else
    {
      double *t_vect;
      t_vect = (double *)malloc((Nt + 3) * sizeof(double));

      for (i = 0; i <= Nt + 2; i++)
        t_vect[i] = i * dt;

      flag_rd = 0;
      n_price = lecture_tr(flag_rd);

      /* 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 = 1; 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_rf[i - 1][jj] * discount_rf[i - 1][jj
              if (jj + y_down[i - 1][jj] == j)
                sum += Q[i - 1][jj] * pd_rf[i - 1][jj] * discount_rf[i - 1][jj
            }

          Q[i][j] = sum;
        }
    /*Compute a[i]*/
    if (i == 0)
      shift_rf[0] = -log(Pc[1]) / dt;
    else
      {
        sum = 0.;
        for (j = 0; j <= i; j++)
          {
            sum += Q[i][j] * exp(-omega_rf * y_rf[i][j] * dt);
          }
```

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

      /*Compute x,r and discount factor d*/
      for (j = 0; j <= i; j++)
        {
          rf[i][j] = omega_rf * y_rf[i][j] + shift_rf[i];
          discount_rf[i][j] = exp(-rf[i][j] * dt);
        }
    }

  return 1;
}


//Quadratic Interpolation for variable mesh
static double quadratic_interpolation(double val, int xi, int k1,int k2, int k3,
{
  double res;
  int l, Index, z;
  double AA, BB, CC;

  if (val <= vect_z[0])
    return P[0][k1][k2][k3];
  else if (val >= vect_z[N])
    return P[N][k1][k2][k3];
  else
    {
      l = current_index;
      while ((vect_z[l] < val) && (l < N)) l++;
      current_index = l;
      if (l == 0)
        res = P[0][k1][k2][k3];
      else if (l < N - 1)
        {
          //Index=l-1;
          for (z = 0; z < 3; z++)
            {
              Index = l - 1;
              xa[z] = vect_z[Index + z];
              ya[z] = P[Index + z][k1][k2][k3];
```

```
            }
          AA = ya[0];
          BB = (ya[1] - ya[0]) / (xa[1] - xa[0]);
          CC = (ya[2] - AA - BB * (xa[2] - xa[0])) / (xa[2] - xa[0]) / (xa[2] -
          res = AA + BB * (val - xa[0]) + CC * (val - xa[0]) * (val - xa[1]);
        }
      else
        {
          res = ((val - vect_z[l - 1]) * P[l][k1][k2][k3]
                 + (vect_z[l] - val) * P[l - 1][k1][k2][k3]) / (vect_z[l] - vect
        }
      return res;
    }
}



/*Compute Price Option*/
int  Fd_HybridTree_HesHw2d(int am, double s0, NumFunc_1  *p, double tt, double v
{
  int i, j;
  double puuu, puud, pudu, pudd, pduu, pdud, pddu, pddd;
  int fv_up, fv_down, yv_up, yv_down, rfv_up, rfv_down;
  double l;
  double alpha, beta, gamma, alpha1, beta1, gamma1;
  int PriceIndex;
  double dx;
  int k1, k2, k3;
  double sm, sp;
  double  *A, *B, *C, *A1, *B1, *C1, *Price, *S,*vect_s;
  double dt;
  double z, vv;
  double bound1, bound2;
  double y0;
  int Index;
  double kappa, omega, rho_Sr, kappa_rf, omega_rf, rho_Srf, kappa2, omega2, rho_
  double rho_vr, rho_vrf, rho_rrf; //UNUSED IN THIS CODE
  double val;
  double sigma=0.5;
  double PRECISION_FDH=1.0e-5;
```

```
//----------------------Initialisation of variable



kappa2 = GET(kappa_vect, 0);
kappa = GET(kappa_vect, 1);
kappa_rf = GET(kappa_vect, 2);

omega2 = GET(sigma_vect, 0);
omega = GET(sigma_vect, 1);
omega_rf = GET(sigma_vect, 2);

rho_Sv = GET(rho, 0);
rho_Sr = GET(rho, 1);
rho_Srf = GET(rho, 2);
rho_vr = GET(rho, 3);
rho_vrf = GET(rho, 4);
rho_rrf = GET(rho, 5);

if ((fabs(rho_vr) > 0) || (fabs(rho_vrf) > 0) || (fabs(rho_rrf) > 0))
  return UNTREATED_CASE;

if (SQR(rho_Sv) + SQR(rho_Sr) + SQR(rho_Srf) >= 1.)
  return UNTREATED_CASE;

init_tr_rd = curve_rd;
init_tr_rf = curve_rf;

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

//Tree construction for r_d
tree_rd(tt, r0, kappa, omega, Nt);
calibration_bond_rd(flat_flag_rd, tt, r0, omega, Nt);

///Tree construction for r_f
tree_rf(tt, r0f, kappa_rf, omega_rf, Nt);
calibration_bond_rf(flat_flag_rf, tt, r0f, omega_rf, Nt);

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

```
A = malloc((N + 1) * sizeof(double));
B = malloc((N + 1) * sizeof(double));
C = malloc((N + 1) * sizeof(double));
A1 = malloc((N + 1) * sizeof(double));
B1 = malloc((N + 1) * sizeof(double));
C1 = malloc((N + 1) * sizeof(double));
vect_z = (double *)malloc((N + 1) * sizeof(double));
vect_s = (double *)malloc((N + 1) * sizeof(double));
Price = (double *)malloc((N + 1) * sizeof(double));
S = (double *)malloc((N + 1) * sizeof(double));

dt = tt / (double)Nt;

y0 = log(s0);

 l=sigma*sqrt(tt)*sqrt(log(1.0/PRECISION_FDH))+fabs((r0-r0f-0.5*sigma)*tt);

dx = 2 * l / (double)N;

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

/*Maturity conditions for Call options*/
for (k1 = 0; k1 <= Nt; k1++)
  for (k2 = 0; k2 <= Nt; k2++)
    for (k3 = 0; k3 <= Nt; k3++)
      for (j = 0; j <= N; j++)
  P[j][k1][k2][k3] = (p->Compute)(p->Par,vect_s[j]);


/*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;
  }

bound1 = 0.;
bound2 = 0.;

/*Dynamic Programming*/
for (i = Nt - 1; i >= 0; i--)
  {
    for (k1 = 0; k1 <= i; k1++)
      {
        for (k2 = 0; k2 <= i; k2++)
          {
            for (k3 = 0; k3 <= i; k3++)
              {
                z = r[i][k1] - rf[i][k3] - 0.5 * V[i][k2] - rho_Sv * kappa2 *

                vv = 0.5 * V[i][k2] * (1. - SQR(rho_Sv) - SQR(rho_Sr) - SQR(rh

                yv_up = y_up[i][k1];
                yv_down = y_down[i][k1];
                fv_up = f_up[i][k2];
                fv_down = f_down[i][k2];
                rfv_up = rf_up[i][k3];
                rfv_down = rf_down[i][k3];

                puuu = pu_y[i][k1] * pu_f[i][k2] * pu_rf[i][k3];
                puud = pu_y[i][k1] * pu_f[i][k2] * pd_rf[i][k3];
                pudu = pu_y[i][k1] * pd_f[i][k2] * pu_rf[i][k3];
                pudd = pu_y[i][k1] * pd_f[i][k2] * pd_rf[i][k3];

                pduu = pd_y[i][k1] * pu_f[i][k2] * pu_rf[i][k3];
                pdud = pd_y[i][k1] * pu_f[i][k2] * pd_rf[i][k3];
                pddu = pd_y[i][k1] * pd_f[i][k2] * pu_rf[i][k3];
                pddd = pd_y[i][k1] * pd_f[i][k2] * pd_rf[i][k3];

P[0][k1][k2][k3] = bound1;
P[N][k1][k2][k3] = bound2;
P1[0][k1][k2][k3] = bound1;
```

```
P1[N][k1][k2][k3] = bound2;

                //Fully Implicit
                /*Lhs Factor of theta-schema*/
                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;

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

                //F_UUU
                //Initialise
 current_index=0;

                for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
                  {
    val=vect_z[PriceIndex]+rho_Sv/omega2*(V[i+1][k2+fv_up]-V[i][k2])+rho_Sr*sqr
Price[PriceIndex]=puuu*quadratic_interpolation(val,i + 1,k1+yv_up,k2 + fv_up,k3

                  }


                //F_UUD
                //Initialise
 current_index=0;

                for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
```

```
                    {
        val=vect_z[PriceIndex]+rho_Sv/omega2*(V[i+1][k2+fv_up]-V[i][k2])+rho_Sr*s
  Price[PriceIndex]+=puud*quadratic_interpolation(val,i + 1,k1+yv_up,k2 + fv_up,

                    }


                  //F_UDU
                  //Initialise
  current_index=0;

                  for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
                    {
     val=vect_z[PriceIndex]+rho_Sv/omega2*(V[i+1][k2+fv_down]-V[i][k2])+rho_Sr*sq
Price[PriceIndex]+=pudu*quadratic_interpolation(val,i + 1,k1+yv_up,k2 + fv_down,
                    }


                  //F_UDD
                  //Initialise
  current_index=0;
                  for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
                    {
    val=vect_z[PriceIndex]+rho_Sv/omega2*(V[i+1][k2+fv_down]-V[i][k2])+rho_Sr*sqr
Price[PriceIndex]+=pudd*quadratic_interpolation(val,i + 1,k1+yv_up,k2 + fv_down,
                    }

                  //F_DUU
                  //Initialise
  current_index=0;

                  for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
                    {
     val=vect_z[PriceIndex]+rho_Sv/omega2*(V[i+1][k2+fv_up]-V[i][k2])+rho_Sr*sqrt
Price[PriceIndex]+=pduu*quadratic_interpolation(val,i + 1,k1+yv_down,k2 + fv_up,
                    }



                  //F_DUD
                  //Initialise
```

```
  current_index=0;

                  for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
                    {
 val=vect_z[PriceIndex]+rho_Sv/omega2*(V[i+1][k2+fv_up]-V[i][k2])+rho_Sr*sqrt(V[
Price[PriceIndex]+=pdud*quadratic_interpolation(val,i + 1,k1+yv_down,k2 + fv_up,
                    }


                  //F_DDU
                  //Initialise
  current_index=0;

                  for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
                    {
 val=vect_z[PriceIndex]+rho_Sv/omega2*(V[i+1][k2+fv_down]-V[i][k2])+rho_Sr*sqrt(
Price[PriceIndex]+=pddu*quadratic_interpolation(val,i + 1,k1+yv_down,k2 + fv_dow
                    }



                  //F_DDD
                  //Initialise
  current_index=0;

                  for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
                    {
 val=vect_z[PriceIndex]+rho_Sv/omega2*(V[i+1][k2+fv_down]-V[i][k2])+rho_Sr*sqrt(
Price[PriceIndex]+=pddd*quadratic_interpolation(val,i + 1,k1+yv_down,k2 + fv_dow
                    }

                  /*Set Rhs*/
                  S[1] = B1[1] * Price[1] + C1[1] * Price[2] + A1[1] * bound1 -
                  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]

                  /*Solve the system*/
                  for (PriceIndex = N - 2; PriceIndex >= 1; PriceIndex--)
```

```
                    S[PriceIndex] = S[PriceIndex] - C[PriceIndex] * S[PriceIndex

                  Price[1] = S[1] / B[1];
                  for (PriceIndex = 2; PriceIndex < N; PriceIndex++)
                    Price[PriceIndex] = S[PriceIndex] / B[PriceIndex] - A[PriceI

                  for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
                    P1[PriceIndex][k1][k2][k3] = discount[i][k1] * Price[PriceIn

                  if (am)
                    for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
                      P1[PriceIndex][k1][k2][k3] = MAX(P1[PriceIndex][k1][k2][k3
                }//end k1
              }//end k2
            }//end k3

        //Copy
        for (k1 = 0; k1 <= Nt; k1++)
          for (k2 = 0; k2 <= Nt; k2++)
            for (k3 = 0; k3 <= Nt; k3++)
              for (j = 0; j <= N; j++)
P[j][k1][k2][k3]=P1[j][k1][k2][k3];
      }//end i

    Index = 0.;
    while (vect_z[Index] < y0)
      {
        Index++;
      }

    Index--;

    sm = vect_s[Index];
    sp = (vect_s[Index + 1]);

    *ptprice = P[Index][0][0][0] + (P[Index + 1][0][0][0] - P[Index][0][0][0]) * (

    free_memory(Nt, N);
    free(A);
    free(B);
    free(C);
```

```
  free(A1);
  free(B1);
  free(C1);
  free(vect_z);
  free(vect_s);
  free(S);
  free(Price);

  return OK;
}

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



  return Fd_HybridTree_HesHw2d(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,
                               ptMod->v0.Val.V_PDOUBLE,
                               ptMod->flat_flag_rd.Val.V_INT,
                               MOD(GetYield_rd)(ptMod),
                               MOD(GetCurve_rd)(ptMod),
                               ptMod->flat_flag_rf.Val.V_INT,
                               MOD(GetYield_rf)(ptMod),
                               MOD(GetCurve_rf)(ptMod),
                               ptMod->theta.Val.V_PDOUBLE,
                               ptMod->kappa.Val.V_PNLVECT,
                               ptMod->sigma.Val.V_PNLVECT,
                               ptMod->rho.Val.V_PNLVECT,
                               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_heshw2d";
      Met->Par[0].Val.V_INT = 30;
      Met->Par[1].Val.V_INT = 50;
    }

  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)
};
```