

## Help

```

#include "heshwld_std.h"
#include "enums.h"
#include "error_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 ***P1;
static double *vect_z;
static int current_index;
static double xa[3], ya[3];

static double **V, * *Q, * **P, * **P1;
static double **y, * *f;
static int **f_down, * *f_up;
static int **y_down, * *y_up;
static double **pu_y, * *pd_y;
static double **pu_f, * *pd_f;
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, j;

    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;
    }

    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_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;
}

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;
    }

    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));

    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));

    return OK;
}

```

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

    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(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(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);

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

```

    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;
}

```

```

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

    }

    return 1;
}

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 FICHER N'A PU ETRE OUVERT. VERIFIER LE CHEMIN\ n");
    }

    /* i is the number of libe 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 interpolate(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 int calibration_bond(int flat_flag, double tt, double r0, double omega, i
{
    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
        }
        interpolate(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);
    }
}

return 1;
}

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

    if (val <= vect_z[0])
        return P[0][k1][k2];
    else if (val >= vect_z[N])
        return P[N][k1][k2];
    else
    {

```



```

    l = current_index;
    while ((vect_z[l] < val) && (l < N)) l++;
    current_index = l;
    if (l == 0)
        res = P[0][k1][k2];
    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];
        }
        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]
            + (vect_z[l] - val) * P[l - 1][k1][k2]) / (vect_z[l] - vect_z[l - 1]);
    }
    return res;
}

/*Compute Price Option*/
int Fd_HybridTree_HesHw(int am, double s0, NumFunc_1 *p, double tt, double div)
{
    int i, j;
    double puu, pud, pdu, pdd;
    int fv_up, fv_down, yv_up, yv_down;
    double l;
    double alpha, beta, gamma, alpha1, beta1, gamma1;
    int PriceIndex;
    double dx;
    int k1, k2;
    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 val;

init_tr = curve;

if ((fabs(rhorv) > 0))
    return UNTREATED_CASE;

if (SQR(rhoSv) + SQR(rhoSr) >= 1.)
    return UNTREATED_CASE;

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

//Tree construction for r
tree_r(tt, r0, kappa, omega, Nt);
calibration_bond(flat_flag, tt, r0, omega, Nt);
//Tree construction for v
tree_v(tt, v02, kappa2, theta2, omega2, 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));
Price = (double *)malloc((N + 1) * sizeof(double));
S = (double *)malloc((N + 1) * sizeof(double));
vect_z=(double *)malloc((N+1)*sizeof(double));
vect_s = (double *)malloc((N + 1) * sizeof(double));

dt = tt / (double)Nt;

y0 = log(s0);

```

```

double sigma=0.5;
double PRECISION_FDH=1.0e-5;
l=sigma*sqrt(tt)*sqrt(log(1.0/PRECISION_FDH))+fabs((r0-divid-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 (j = 0; j <= N; j++)
            P[j][k1][k2] = (p->Compute)(p->Par,vect_s[j]);

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 (k2 = 0; k2 <= i; k2++)
    {
        for (k1 = 0; k1 <= i; k1++)
        {
            z = r[i][k1] - divid - 0.5 * V[i][k2] - rhoSv * kappa2 * (theta2 -

```

```

vv = 0.5 * V[i][k2] * (1. - SQR(rhoSv) - SQR(rhoSr));

//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 + 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];

yv_up = y_up[i][k1];
yv_down = y_down[i][k1];
fv_up = f_up[i][k2];
fv_down = f_down[i][k2];

puu = pu_y[i][k1] * pu_f[i][k2];
pud = pu_y[i][k1] * pd_f[i][k2];
pdu = pd_y[i][k1] * pu_f[i][k2];
pdd = pd_y[i][k1] * pd_f[i][k2];

P[0][k1][k2] = bound1;
P[N][k1][k2] = bound2;

```

```

P1[0][k1][k2] = bound1;
P1[N][k1][k2] = bound2;

//F_UU
//Initialise
current_index=0;
for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
{
val=vect_z[PriceIndex]+rhoSv/omega2*(V[i+1][k2+fv_up]-V[i][k2])+rhoSr*sqrt(V[i]
Price[PriceIndex]=puu*quadratic_interpolation(val,i + 1,k1+yv_up,k2 + fv_up,N);
}

//F_UD
//Initialise
current_index=0;
for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
{
val=vect_z[PriceIndex]+rhoSv/omega2*(V[i+1][k2+fv_down]-V[i][k2])+rhoSr*sqrt(V[
Price[PriceIndex]+=pud*quadratic_interpolation(val,i + 1,k1+yv_up,k2 + fv_dow
}

//F_DU
//Initialise
current_index=0;
for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
{
val=vect_z[PriceIndex]+rhoSv/omega2*(V[i+1][k2+fv_up]-V[i][k2])+rhoSr*sqrt(V[i]
Price[PriceIndex]+=pdu*quadratic_interpolation(val,i + 1,k1+yv_down,k2 + fv_u
}

//F_DD
//Initialise
current_index=0;
for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
{
val=vect_z[PriceIndex]+rhoSv/omega2*(V[i+1][k2+fv_down]-V[i][k2])+rhoSr*sqrt(V[
Price[PriceIndex]+=pdd*quadratic_interpolation(val,i + 1,k1+yv_down,k2 + fv_d
}

```

```

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

/*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++)
    P1[PriceIndex][k1][k2] = discount[i][k1] * Price[PriceIndex];

if (am)
    for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
        P1[PriceIndex][k1][k2] = MAX(P1[PriceIndex][k1][k2], (p->Compu

    }//end k2
} //end k1

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

} //end i

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

```



```

        , ptMod->kV.Val.V_PDOUBLE,
        ptMod->thetaV.Val.V_PDOUBLE,
        ptMod->SigmaV.Val.V_PDOUBLE,
        ptMod->RhoSr.Val.V_PDOUBLE,
        ptMod->RhoSV.Val.V_PDOUBLE,
        ptMod->RhorV.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)->Name, "PutEuro") == 0))
        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_heshw";
        Met->Par[0].Val.V_INT = 50;
        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)  
};
```