

[Help](#)

```
#include "
href../../../../mod/merhes1d/merhes1d_lim/merhes1d_lim_h_src.pdfhes1d_lim.h"
#include "
href../../../../common/enums_h_src.pdfenums.h"
#include "pnl/pnl_interpolation.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2014+2) //The "#els
static int CHK_OPT(FD_HYBRIDTREE_BarrierHeston)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(FD_HYBRIDTREE_BarrierHeston)(void *Opt, void *Mod, PricingMethod *Met)
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else
static double **V, * *Q, * *P_old, * *P_new, * *vect_y_old, * *vect_y_new;
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 int current_index;
static double xa[3], ya[3];

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

```

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

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

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

```

```

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

    return OK;
}

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

    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++)
    free(P_old[i]);
free(P_old);

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

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

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

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;
}

static double compute_S(double Y, double rv, double omega, double rho)
{
    double val;

    val = exp(Y) * exp(rho * rv / omega);

    return val;
}

/*Calibration of the tree the stochastic volatility v*/
static int tree_v(double tt, double v0, double kappa, double theta, double omega)
{
    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(v0, omega);

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

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

```

```

V[1][0] = compute_v(f[1][0], omega);
V[1][1] = compute_v(f[1][1], omega);
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], omega);
        V[i + 1][j + 1] = compute_v(f[i + 1][j + 1], omega);
    }

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 = kappa * (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];

                z = 0;
                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])
    {
        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])
    {
        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;
}

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

    if (val <= vect_y_old[0][k])
        return P_old[0][k];
    else if (val >= vect_y_old[N][k])
        return P_old[N][k];
    else
    {
        l = current_index;
        while ((vect_y_old[l][k] < val) && (l < N)) l++;
    }

```

```

    current_index = l;
    if (l == 0)
        res = P_old[0][k];
    else if (l < N - 1)
    {
        //Index=l-1;
        for (z = 0; z < 3; z++)
        {
            Index = l - 1;
            xa[z] = vect_y_old[Index + z][k];
            ya[z] = P_old[Index + z][k];
        }
        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_y_old[l - 1][k]) * P_old[l][k]
            + (vect_y_old[l][k] - val) * P_old[l - 1][k]) / (vect_y_old[l][k]
        }
    }
    return res;
}
}

```

```

static int FDHYBRIDTREE_BarrierHeston(int up_or_down, double rebate, double barr
{
    int i, j, k;
    double stock;
    int fv_up, fv_down;
    double l;
    double alpha, beta, gamma, alpha1, beta1, gamma1;
    double dx;
    double discount;
    double bound1, bound2;
    double z, vv;
    double dt;
    double *A, *B, *C, *A1, *B1, *C1, *Price, *S;
    int Index, PriceIndex;
    double log_barrier, y0;
}

```

```

double sm, s, sp, spp;
double val;
double pm, pp;
double y_min;
double K;
double a,b,c;
double sigma=0.5;
double PRECISION_FDH=1.0e-5;

/*Memory Allocation*/
memory_allocation(Nt, N);

//Tree construction for v
tree_v(tt, v0, kappa, theta, omega, Nt);

//Finite Difference algorithm
K = p->Par[0].Val.V_PDDOUBLE;

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));

dt = tt / (double)Nt;

l=sigma*sqrt(tt)*sqrt(log(1.0/PRECISION_FDH))+fabs((r_fisso-divid-0.5*sigma)*
if(tt>3)
l=log(s0/2.);
if(tt>10)
l=log(s0);

/*Maturity conditions*/
for (k = 0; k <= Nt; k++)
{
    y0 = log(s0) - rho / omega * V[Nt][k];

```

```

log_barrier = log(barrier) - rho / omega * V[Nt][k];
if (up_or_down)
{
    dx = (log_barrier - (y0 - 1)) / (double)N;
    y_min = y0 - 1;
}
else
{
    dx = ((y0 + 1) - log_barrier) / (double)N;
    y_min = log_barrier;
}
for (j = 0; j <= N; j++)
{
    vect_y_old[j][k] = y_min + (double)j * dx;
    stock = compute_S(vect_y_old[j][k], V[Nt][k], omega, rho);
    P_old[j][k] = (p->Compute)(p->Par, stock);
}
}

if (!am)
for (k = 0; k <= Nt; k++)
{
    if (up_or_down)
        P_old[N][k] = rebate;
    else
        P_old[0][k] = rebate;
}

discount = exp(-r_fisso * dt);
/*Dynamic Programming*/
for (i = Nt - 1; i >= 0; i--)
{
    for (k = 0; k <= i; k++)
    {
        z = r_fisso - divid - 0.5 * V[i][k] - rho * kappa * (theta - V[i][k])
        vv = 0.5 * V[i][k] * (1. - SQR(rho));

        y0 = log(s0) - rho / omega * V[i][k];
        log_barrier = log(barrier) - rho / omega * V[i][k];

        if (up_or_down)

```

```

    {
        dx = (log_barrier - (y0 - 1)) / (double)N;
        y_min = y0 - 1;
    }
else
    {
        dx = ((y0 + 1) - log_barrier) / (double)N;
        y_min = log_barrier;
    }

for (j = 0; j <= N; j++)
    vect_y_new[j][k] = y_min + (double)j * dx;

z = r_fisso - divid - 0.5 * V[i][k] - rho * kappa * (theta - V[i][k])
vv = 0.5 * V[i][k] * (1. - SQR(rho));

//Boundary
if (up_or_down)
{
    stock = compute_S(vect_y_new[0][k], V[i][k], omega, rho);
    bound1 = (p->Compute)(p->Par, stock);

    bound2 = rebate;
    if (am)
    {
        stock = compute_S(vect_y_new[N][k], V[i][k], omega, rho);
        bound2 = (p->Compute)(p->Par, stock);
    }
}
else
{
    stock = compute_S(vect_y_new[0][k], V[i][k], omega, rho);
    bound2 = (p->Compute)(p->Par, stock);

    bound1 = rebate;
    if (am)
    {
        stock = compute_S(vect_y_new[N][k], V[i][k], omega, rho);
        bound1 = (p->Compute)(p->Par, stock);
    }
}

```

```

}

P_new[0][k] = bound1;
P_new[N][k] = bound2;

//Fully Implicit
/*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;
}
/*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 = f_up[i][k];
fv_down = f_down[i][k];

```

```

//Initialise
current_index = 0;
for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
{
    val = vect_y_new[PriceIndex][k];
    //Price[PriceIndex]=P_old[PriceIndex][k+fv_up];
    Price[PriceIndex] = pu_f[i][k] *quadratic_interpolation(val, i
}

//Initialise
current_index = 0;
for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
{
    val = vect_y_new[PriceIndex][k];
    Price[PriceIndex] += pd_f[i][k] *quadratic_interpolation(val,
}

/*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++)
{
    P_new[PriceIndex][k] = discount* Price[PriceIndex];
}

if (am)

```

```

        for (PriceIndex = 0; PriceIndex <= N; PriceIndex++)
            P_new[PriceIndex][k] = MAX(P_new[PriceIndex][k], compute_S(vect_y_
    }//end k

    //Copy
    for (j = 0; j <= N; j++)
        for (k = 0; k <= i; k++)
            {
                vect_y_old[j][k] = vect_y_new[j][k];
                P_old[j][k] = P_new[j][k];
            }

    }//end i

/* //S0 Index */
y0 = log(s0) - rho / omega * V[0][0];

Index = 0;
while ((vect_y_new[Index][0] < y0) && (Index < N - 1))
    {
        Index++;
    }

Index--;

if (Index < 0) Index = 0;

sm = compute_S(vect_y_new[Index][0], V[0][0], omega, rho);
s = compute_S(y0, V[0][0], omega, rho);
sp = compute_S(vect_y_new[Index + 1][0], V[0][0], omega, rho);
pm = (P_new[Index + 1][0] - P_new[Index][0]);

//Price
*ptprice = P_new[Index][0] + pm * (s - sm) / (sp - sm);

//Delta
spp = compute_S(vect_y_new[Index + 2][0], V[0][0], omega, rho);

pp = (P_new[Index + 2][0] - P_new[Index + 1][0]);

a = (pp / (spp - sp) - pm / (sp - sm)) / (spp - sm);

```



```

b = pp / (spp - sp) - a * (spp + sp);
c = P_new[Index][0] - a * SQR(sm) - b * sm;

*ptdelta = 2 * a * s + b;

/*Memory Disallocation*/
free_memory(Nt, N);

free(A);
free(B);
free(C);
free(A1);
free(B1);
free(C1);
free(S);
free(Price);

return OK;
}

int CALC(FD_HYBRIDTREE_BarrierHeston)(void *Opt, void *Mod, PricingMethod *Met)
{
    TYPEOPT *ptOpt = (TYPEOPT *)Opt;
    TYPEMOD *ptMod = (TYPEMOD *)Mod;
    double r, divid;
    int upordown;
    double rebate, limit;

    if (ptMod->Sigma.Val.V_PDDOUBLE == 0.0)
    {
        Fprintf(TOSCREEN, "BLACK-SHOLES MODEL\ n\ n\ n");
        return WRONG;
    }
    else
    {
        r = log(1. + ptMod->R.Val.V_DOUBLE / 100.);
        divid = log(1. + ptMod->Divid.Val.V_DOUBLE / 100.);

        rebate = ((ptOpt->Rebate.Val.V_NUMFUNC_1)->Compute)((ptOpt->Rebate.Val.V_N

```

```

        limit = ((ptOpt->Limit.Val.V_NUMFUNC_1)->Compute)((ptOpt->Limit.Val.V_NUMF

    if ((ptOpt->DownOrUp).Val.V_BOOL == DOWN)
        upordown = 0;
    else upordown = 1;

    return FDHYBRIDTREE_BarrierHeston(upordown, rebate, limit, ptOpt->EuOrAm.V
        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_PINT,
        Met->Par[1].Val.V_PINT,
        &(Met->Res[0].Val.V_DOUBLE),
        &(Met->Res[1].Val.V_DOUBLE));
    }

}

static int CHK_OPT(FD_HYBRIDTREE_BarrierHeston)(void *Opt, void *Mod)
{
    Option *ptOpt = (Option *)Opt;
    TYPEOPT *opt = (TYPEOPT *) (ptOpt->TypeOpt);

    if ((opt->OutOrIn).Val.V_BOOL == OUT &&
        (opt->Parisian).Val.V_BOOL == FALSE)
        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_barrierheston";
        Met->Par[0].Val.V_INT = 100;
    }
}

```

```

        Met->Par[1].Val.V_INT = 300;
    }

    return OK;
}

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

```