

## [Help](#)

```
extern "C" {
#include "
href../../../../mod/merhes1d/merhes1d_std/merhes1d_std_h_src.pdfmerhes1d_std.h"
#include "
href../../../../common/enums_h_src.pdfenums.h"
}
#include "
href../../../../common/math/levy_h_src.pdfmath/levy.h"
#include "
href../../../../common/math/fft_h_src.pdfmath/fft.h"
#include "
href../../../../common/math/numerics_h_src.pdfmath/numerics.h"
extern "C" {

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2014+2) //The "#els
    static int CHK_OPT(FD_HybridTree_Bates)(void *Opt, void *Mod)
    {
        return NONACTIVE;
    }
    int CALC(FD_HybridTree_Bates)(void *Opt, void *Mod, PricingMethod *Met)
    {
        return AVAILABLE_IN_FULL_PREMIA;
    }
#else

    static double **V, * *P_old, * *P_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;

    /*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;
}

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_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;

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

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);

    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);
        }

    /*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];

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;
}

static int HybridTree_Bates(int am, double S0, NumFunc_1 *Payoff, double T, d
{
    double K;
    int call_or_put;

    K = Payoff->Par[0].Val.V_PDOUBLE;

    if ((Payoff->Compute) == &Call)
        call_or_put = 1;
    else
        call_or_put = 0;

    //Sigma for boundary condition
    double sigma = 0.5;

    /*Construction of the model*/
    double delta = sqrt(gamma2);
    Merton_measure measure(mean, delta, lambda, sigma, dx);

    double km = 3;
    double A1 = log(2. / 3) + T * measure.espX1 - km * sqrt(T * measure.varX1);
    double Ar = log(2.) + r * T + km * sqrt(T * measure.varX1);
    if (A1 < -30) A1 = -30;
    if (Ar > 30) Ar = 30;
    int Nl = (int)ceil(-A1 / dx);
    int Nr = (int)ceil(Ar / dx);
    int NSpace = Nl + Nr;
    A1 = -Nl * dx;
    Ar = Nr * dx;
    A1 = log(S0) - rho / omega * v0 + A1;
    Ar = log(S0) - rho / omega * v0 + Ar;

    int k, i, j;
    int fv_up, fv_down;
    double stock;
    double dt;

```

```

double **P_Old, **P_New;
double *A, *B, *C, *S;
double *beta_p, *u, *v, *Pr;
int N_fft;
int p;
int NN_fft;
int Nz ;
double *mu;
double *mu_img ;
double *uaux;
double *uaux_img;
double *somme;
double *somme_img;
double *vect_t;
double *vect_y;
double y0;
double *Price;

/*Memory Allocation*/
memory_allocation(Ntime, NSpace);

//CIR Tree
Tree_v(T, v0, kappa, theta, omega, Ntime);

dt = T / (double)Ntime;

dx = (Ar - Al) / (NSpace);

const int Kmax = measure.Kmax;
const int Kmin = measure.Kmin;

vect_t = new double [Ntime + 1];
vect_y = new double [NSpace + 1];

for (int n = 0; n <= Ntime; n++)
    vect_t[n] = n * dt;

P_Old = new double*[Ntime + 1];
for (int i = 0; i <= Ntime; i++)
    P_Old[i] = new double [NSpace + 1];

```

```

P_New = new double*[Ntime + 1];
for (int i = 0; i <= Ntime; i++)
    P_New[i] = new double [NSpace + 1];

//Terminal Condition
for (j = 0; j <= NSpace; j++)
    vect_y[j] = A1 + (double)j * dx;

for (k = 0; k <= Ntime; k++)
    for (int i = 0; i <= NSpace; i++)
    {
        stock = compute_S(vect_y[i], V[Ntime][k], omega, rho);
        P_Old[k][i] = (Payoff->Compute)(Payoff->Par, stock);
    }

//some useful coefficients
/*matrix coefficients of the implicit part*/
A = new double[NSpace + 1];
B = new double[NSpace + 1];
C = new double[NSpace + 1];
S = new double[NSpace + 1];
Price=new double [NSpace+1];
Pr = new double[NSpace + 1];
beta_p = new double[NSpace + 1];
u = new double[NSpace + 1];
v = new double[NSpace + 1];

N_fft = NSpace + Kmax - Kmin; //number of non-zero values of u involved in c
//zero-padding to obtain NN = N + Nz = 2^p
p = 1;
NN_fft = 2; //size of auxiliary vectors
while (NN_fft < N_fft)
{
    p++;
    NN_fft = 2 * NN_fft;
}
Nz = NN_fft - N_fft; // number of extra zeros

mu = new double [NN_fft];
mu_img = new double [NN_fft];

```

```

uaux = new double [NN_fft];
uaux_img = new double [NN_fft];
somme = new double [NN_fft];
somme_img = new double [NN_fft];

/*computation of the Fourier transform of nu*/
for (int i = 0; i < Kmax - Kmin + 1; i++)
{
    mu[i] = (*measure.nu_array)[Kmax - Kmin - i];
    mu_img[i] = 0;
}
for (int i = Kmax - Kmin + 1; i < NN_fft; i++)
{
    mu[i] = 0;
    mu_img[i] = 0;
}
fft1d(mu, mu_img, NN_fft, -1);

double discount = exp(-r * dt);

for (int n = Ntime; n >= 1; n--) //time iterations
{
    for (k = 0; k <= n-1; k++)
    {
        fv_up = f_up[n - 1][k];
        fv_down = f_down[n - 1][k];

        double z, vv;
        z = r - divid - 0.5 * V[n - 1][k] - rho * kappa * (theta - V[n - 1][k]);
        vv = 0.5 * V[n - 1][k] * (1. - SQR(rho));

        double alpha, beta, gamma;
        int PriceIndex;
        double bound1, bound2;

        if (call_or_put == 1)
        {
            bound1 = 0;
            bound2 = compute_S(vect_y[NSpace], V[Ntime][k], omega, rho) * ex
        }
        else

```

```

    {
        bound1 = K * exp(-r * (Ntime - k) * dt) - compute_S(vect_y[0], V
        bound2 = 0;
    }

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 <= NSpace - 1; PriceIndex++)
{
    A[PriceIndex] = alpha;
    B[PriceIndex] = beta;
    C[PriceIndex] = gamma;
}

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

//Mixture of terminal conditions
for(int i=0;i<=NSpace;i++)
    Price[i]= pu_f[n - 1][k]*P_Old[k + fv_up][i]+pd_f[n - 1][k]*P_Old[k + fv_down]

// /*calculation of the discretized integral using FFT*/
for (int i = 0; i < NSpace - 1; i++)
{
    if ((Kmax + 1 + i < 0) || (Kmax + 1 + i >= NSpace))
    {
        uaux[i] = 0.;
        uaux[i] = MAX(0., compute_S(A1 + (double)(Kmax + 1 + i) * dx
    }
    else uaux[i] = Price[Kmax+1+i];
    uaux_img[i] = 0;
}

for (int i = NSpace - 1; i < NSpace + Nz - 1; i++)

```

```

        {
            uaux[i] = 0; //zero-padding
            uaux_img[i] = 0;
        }
    for (int i = NSpace + Nz - 1; i < NN_fft; i++)
    {
        if ((Kmin - NSpace - Nz + 1 + i < 0) || (Kmin - NSpace - Nz + 1
            {
                uaux[i] = 0.;
                uaux[i] = MAX(0., compute_S(A1 + (double)(Kmin - NSpace - Nz
            }
        else uaux[i] = Price[Kmin-NSpace-Nz+1+i];
        uaux_img[i] = 0;
    }

    fft1d(uaux, uaux_img, NN_fft, -1);

    for (int i = 0; i < NN_fft; i++)
    {
        somme[i] = mu[i] * uaux[i] - mu_img[i] * uaux_img[i];
        somme_img[i] = mu_img[i] * uaux[i] + mu[i] * uaux_img[i];
    }
    fft1d(somme, somme_img, NN_fft, 1);

    if(measure.alpha < 0)
    {
        S[0]=bound1+dt*(somme[NN_fft-1]-measure.alpha*(Price[1]-bound1)/dx-measu
;
        for(PriceIndex=1;PriceIndex<NSpace;PriceIndex++)
        S[PriceIndex]=Price[PriceIndex]+ dt*(somme[PriceIndex-1]-measure.alpha*(Pr

        S[NSpace]=bound2+dt*(somme[NN_fft-1]-measure.alpha*(bound2-Price[NSpace-

    }
else
    {
        S[0]=bound1+dt*(somme[NN_fft-1]-measure.alpha*(Price[1]-bound1)/dx-measu
;
        for(PriceIndex=1;PriceIndex<NSpace;PriceIndex++)
        S[PriceIndex]=Price[PriceIndex]+ dt*(somme[PriceIndex-1]-measure.alpha
        S[NSpace]=bound2+dt*(somme[NN_fft-1]-measure.alpha*(bound2-Price[NSpace-

```

```

    }

    /*Solve the system*/
    for (PriceIndex = NSpace - 1; PriceIndex >= 1; PriceIndex--)
        S[PriceIndex] = S[PriceIndex] - C[PriceIndex] * S[PriceIndex + 1];

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

    for (PriceIndex = 2; PriceIndex < NSpace; PriceIndex++)
        Pr[PriceIndex] = S[PriceIndex] / B[PriceIndex] - A[PriceIndex] * P

P_New[k] [0]=0;
P_New[k] [NSpace]=0;

    for (PriceIndex = 1; PriceIndex < NSpace; PriceIndex++)
    {
        P_New[k] [PriceIndex]= discount * Pr[PriceIndex];
        if (am)
        {
            stock = compute_S(vect_y[PriceIndex], V[n - 1] [k], omega, rh
            P_New[k] [PriceIndex] = MAX(P_New[k] [PriceIndex], (Payoff->Co
        }
    }

} //k iteration

//Copy
for (j = 1; j < NSpace; j++)
    for (k = 0; k <= n-1; k++)
        P_Old[k] [j] = P_New[k] [j];

} //end of time iterations

//s0 Index
int Index;
double sm, s, sp;

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

```

```

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

Index--;
if (Index < 0) Index = 0;

sm = compute_S(vect_y[Index], V[0][0], omega, rho);
s = compute_S(y0, V[0][0], omega, rho);
sp = compute_S(vect_y[Index + 1], V[0][0], omega, rho);

//Price
*ptprice = P_New[0][Index] + (P_New[0][Index + 1] - P_New[0][Index]) * (s -

//Delta
*ptdelta = (P_New[0][Index + 1] - P_New[0][Index]) / (s - sm);

delete [] beta_p;
delete [] u;
delete [] v;

delete [] mu;
delete [] mu_img;
delete [] uaux;
delete [] uaux_img;
delete [] somme;
delete [] somme_img;

delete [] A;
delete [] B;
delete [] C;
delete [] S;
delete [] Pr;

delete [] vect_t;
delete [] vect_y;
delete [] Price;

```



```

    for (i = 0; i <= Ntime; i++)
        delete[] P_Old[i];

    delete [] P_Old;

    for (i = 0; i <= Ntime; i++)
        delete[] P_New[i];

    delete [] P_New;

    /*Memory Disallocation*/
    free_memory(Ntime, NSpace);

    return OK;
}

int CALC(FD_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 HybridTree_Bates(ptOpt->EuOrAm.Val.V_BOOL, ptMod->SO.Val.V_PDOUBLE,
                            ptOpt->PayOff.Val.V_NUMFUNC_1, ptOpt->Maturity.Val.V_
                            , ptMod->MeanReversion.Val.V_PDOUBLE,
                            ptMod->LongRunVariance.Val.V_PDOUBLE,
                            ptMod->Sigma.Val.V_PDOUBLE,
                            ptMod->Rho.Val.V_PDOUBLE, ptMod->Lambda.Val.V_PDOUBL
}

static int CHK_OPT(FD_HybridTree_Bates)(void *Opt, void *Mod)
{
    if ((strcmp(((Option *)Opt)->Name, "CallEuro") == 0) || (strcmp(((Option *)O
        return OK;

    return WRONG;
}

```

```

    }
#endif //PremiaCurrentVersion

static int MET(Init)(PricingMethod *Met, Option *Opt)
{
    static int first = 1;

    if (first)
    {
        Met->HelpFilenameHint = "fd_hybridTree_bates";
        Met->Par[0].Val.V_PDOUBLE = 0.01;
        Met->Par[1].Val.V_INT2 = 50;
        first = 0;
    }

    return OK;
}

PricingMethod MET(FD_HybridTree_Bates) =
{
    "FD_BRIANICARAMELLINOZANETTE",
    { {"Space Discretization Step", DOUBLE, {500}, ALLOW}, {"TimeStepNumber", IN
        {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CALC(FD_HybridTree_Bates),
    {"Price", DOUBLE, {100}, FORBID}, {"Delta", DOUBLE, {100}, FORBID}, {" ", P
    CHK_OPT(FD_HybridTree_Bates),
    CHK_split,
    MET(Init)
};

}

```