Help

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

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2017+2) //The "#els
static int CHK_OPT(FD_CHIARELLA_BarrierHeston)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(FD_CHIARELLA_BarrierHeston)(void *Opt, void *Mod, PricingMethod *Met)
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else

//////////
// Functions to manage grids.
//////////
static int lower_index(double *grid, int size, double value)
{
    double value_nearest;
    int index_nearest;
    int i;

    value_nearest = fabs(grid[0]-value);
    index_nearest = -1;

    for (i=0; i<size; i++)
    {
        if (fabs(grid[i]-value) <= value_nearest)
        {
            value_nearest = fabs(grid[i]-value);
            index_nearest = i;
        }
    }
    if (grid[index_nearest] > value)
    {
        return index_nearest-1;
```

```
    }
    else
    {
        return index_nearest;
    }
}

static void grid_generation_uniform(double *agrid, double Amin, double Amax, int
{
    int i;
    double deltaxi;

    deltaxi = (Amax-Amin)/Na;

    // Definition of uniform grid.
    for (i=0; i<=Na; i++)
    {
        agrid[i] = Amin + i * deltaxi;
    }
}

static void grid_generation_asset(double *sgrid, double Smin, double Sleft, doub
{
    int i;
    int N1, N2, N3, N4;
    double step;

    // Definition of the spot grid

    N2 = Ns*Coeff_s/2.; //points in [Sleft,Sright]
    N4 = Ns*Coeff_s/2.; //points in [Srightright,Smax]
    N3 = (Ns - N2 - N4)/2.;
    N1 = Ns - N2 - N3 - N4;

    sgrid[0] = Smin;

    step = (Sleft-Smin)/N1;
    for (i=1; i<=N1; i++)
    {
        sgrid[i] = sgrid[i-1] + step;
    }
```

```
    step = (Sright-Sleft)/N2;
    for (i=1+N1; i<=N1+N2; i++)
    {
        sgrid[i] = sgrid[i-1]+ step;
    }

    step = (Srightright-Sright)/N3;
    for (i=1+N1+N2; i<=N1+N2+N3; i++)
    {
        sgrid[i] = sgrid[i-1]+ step;
    }

    step = (Smax-Srightright)/N4;
    for (i=1+N1+N2+N3; i<=Ns; i++)
    {
        sgrid[i] = sgrid[i-1]+ step;
    }
}

static double interpolation(double griddown, double valuedown,
                            double gridup, double valueup, double gridunknown)
{
    return (valueup-valuedown)/(gridup-griddown) * (gridunknown - griddown) + va
}

static double double_interpolation(double value_sd_vd, double value_sd_vu,
                                   double value_su_vd, double value_su_vu,
                                   double grid_sd, double grid_su,
                                   double grid_vd, double grid_vu,
                                   double s, double v)
{
    double value_sd,value_su;
    value_sd = interpolation (grid_vd, value_sd_vd, grid_vu, value_sd_vu, v);
    value_su = interpolation (grid_vd, value_su_vd, grid_vu, value_su_vu, v);
    return interpolation (grid_sd, value_sd, grid_su, value_su, s);
}

static double quadratic_interpolation(double v1, double v2, double v3,
                                      double value_v1, double value_v2, double v
                                      double v0)
```

```
{
    double a, b, c, det;
    // Solve the system

    //    a * v1 * v1 + b * v1 + c = value_v1;
    //    a * v2 * v2 + b * v2 + c = value_v2;
    //    a * v3 * v3 + b * v3 + c = value_v3;

    // Compute Det

    det = v1*v1*v2*1 + v1*1*v3*v3 + 1*v2*v2*v3
    - (v3*v3*v2*1 + v3*1*v1*v1 + 1*v2*v2*v1);

    //    (v1v1, v1, 1)
    //    (v2v2, v2, 1)
    //    (v3v3, v3, 1)

    //   Compute Comatrice

    //    (v2-v3, -v2v2+v3v3, v2v2v3-v2v3v3)
    //    (-v1+v3, v1v1-v3v3, -v1v1v3+v1v3v3)
    //    (v1-v2, -v1v1+v2v2, v1v1v2-v1v2v2)

    //    Transposition and signs

    //    (v2-v3, v3-v1, v1-v2)
    //    (v3v3-v2v2, v1v1-v3v3, v2v2-v1v1)
    //    (v2v2v3-v2v3v3, v3v3v1-v3v1v1, v1v1v2-v1v2v2)

    a = (v2-v3) * value_v1 + (v3-v1) * value_v2 + (v1-v2) * value_v3;
    b = (v3*v3-v2*v2) * value_v1 + (v1*v1-v3*v3) * value_v2 + (v2*v2-v1*v1) * va
    c = (v2*v2*v3-v2*v3*v3) * value_v1 + (v3*v3*v1-v3*v1*v1) * value_v2 + (v1*v1

    return (a * v0 * v0 + b * v0 + c)/det;
}

static int find_early_exercice(double K, double *sgrid, int Ns, int call_or_put,
                               double *sol_R, double *sol_W)
{
    int i;
```

```
    double sign;

    if (call_or_put==1)
    {
        // Since by hypothesis dC/dS = 1 at the early exercice boundary for the
        // Find the first index where (R*1+W - (S-K)) becomes negative.
        i=0;
        sign = 1.;
        while ((i<Ns+1) && (sign>0.))
        {
            sign = sol_R[i] + sol_W[i] - (sgrid[i]-K);
            i++;
        }
        return ((sign<=0.)?(i-1):Ns+1);
    }
    else
    {
        // Since by hypothesis dP/dS = -1 at the early exercicse boundary for th
        // Find backward the first index where (R*-1+W - (K-S)) becomes negative
        i=Ns;
        sign = 1.;
        while ((i>-1) && (sign>0.))
        {
            sign = -sol_R[i] + sol_W[i] - (K-sgrid[i]);
            i--;
        }
        return ((sign<=0.)?(i+1):-1);
    }
}

//////////
// Functions to solve ODE's.
//////////
static void solve_R_edo(double coeff_A, double coeff_B, double *sgrid, int Ns, d
{
    // Solve the R EDO for n and m fixed.
    int i;
    double a, b, r, ap1, bp1;
    double quad_a, quad_b, quad_c;
    double s, sp1, h;
```

```
    sol_R[0] = initial;

    for (i=0; i<Ns; i++)
    {

        s = sgrid[i];
        sp1 = sgrid[i+1];
        h = sp1-s;

        r = sol_R[i];

        a = coeff_A/(s*s);
        b = coeff_B/s;
        ap1 = coeff_A/(sp1*sp1);
        bp1 = coeff_B/sp1;

        // We have a quadratic equation quad_a r r + quad_b r + quad_c = 0
        quad_a = ap1 * h / 2.;
        quad_b = 1. + bp1 * h /2.;
        quad_c = - r - h + b * h * r / 2. + a * h * r * r / 2.;

        // Solution is -2 c / (b + sqrt(Delta)) since forward and R'>0
        sol_R[i+1] = -2. * quad_c / (quad_b + sqrt(quad_b*quad_b-4.*quad_a*quad_
    }
}

static void solve_R_edo_backward(double coeff_A, double coeff_B, double *sgrid,
{
    // Solve backward the R EDO for n and m fixed.
    int i;
    double a, b, ap1, bp1, rp1;
    double quad_a, quad_b, quad_c;
    double s, sp1, h;

    sol_R[Ns] = terminal;

    for (i=0; i<Ns; i++)
    {
        s = sgrid[Ns-i-1];
        sp1 = sgrid[Ns-i];
        h = sp1 - s;
```

```
        rp1 = sol_R[Ns-i];

        ap1 = coeff_A/(sp1*sp1);
        bp1 = coeff_B/sp1;
        a = coeff_A/(s*s);
        b = coeff_B/s;

        // We have a quadratic equation quad_a r r + quad_b r + quad_c = 0
        quad_a = a * h / 2.;
        quad_b = -1. + b * h /2.;
        quad_c = rp1 - h + bp1 * h * rp1 / 2. + ap1 * h * rp1 * rp1 / 2.;

        // Solution is -2 c / (b - sqrt(Delta)) since backward and R'>0
        sol_R[Ns-i-1] = -2. * quad_c / (quad_b - sqrt(quad_b*quad_b-4.*quad_a*qu
    }
}

static void solve_W_edo(double coeff_A, double *coeff_P, double *coeff_Q,
                        double *sgrid, int Ns, double initial, double *sol_R, do
{
    // Solve the W EDO for n and m fixed.
    int i;
    double a, p, q, r, w, ap1, pp1, qp1, rp1;
    double s, h, sp1;

    sol_W[0] = initial;

    for (i=0; i<Ns; i++)
    {
        s = sgrid[i];
        sp1 = sgrid[i+1];
        h = sp1 - s;

        w = sol_W[i];

        a = coeff_A/(s*s);
        p = coeff_P[i]/(s*s);
        q = coeff_Q[i]/s;
        r = sol_R[i];
```

```
        ap1 = coeff_A/(sp1*sp1);
        pp1 = coeff_P[i+1]/(sp1*sp1);
        qp1 = coeff_Q[i+1]/sp1;
        rp1 = sol_R[i+1];

        sol_W[i+1] = (2. * w - h * (a*r*w + r*p + r*q + rp1*pp1 + rp1*qp1))
        / (2. + h*ap1*rp1);
    }
}

static void solve_W_edo_backward(double coeff_A, double *coeff_P, double *coeff_
                                 double *sgrid, int Ns, double terminal, double
{
    // Solve backward the W EDO for n and m fixed.
    int i;
    double a, p, q, r, ap1, pp1, qp1, rp1, wp1;
    double s, h, sp1;

    sol_W[Ns] = terminal;

    for (i=0; i<Ns; i++)
    {
        s = sgrid[Ns-i-1];
        sp1 = sgrid[Ns-i];
        h = sp1 - s;

        wp1 = sol_W[Ns-i];

        ap1 = coeff_A/(sp1*sp1);
        pp1 = coeff_P[Ns-i]/(sp1*sp1);
        qp1 = coeff_Q[Ns-i]/sp1;
        rp1 = sol_R[Ns-i];

        a = coeff_A/(s*s);
        p = coeff_P[Ns-i-1]/(s*s);
        q = coeff_Q[Ns-i-1]/s;
        r = sol_R[Ns-i-1];

        sol_W[Ns-i-1] = (2. * wp1 + h * (ap1*rp1*wp1 + rp1*pp1 + rp1*qp1 + r*p +
        / (2. - h * a * r);
    }
```

```
}

static void solve_V_edo_backward(double coeff_A, double coeff_B,
                                 double *coeff_P, double *coeff_Q,
                                 double *sgrid, int Ns, double terminal,
                                 double *sol_R, double *sol_W, double *sol_V)
{
    // Solve backward the V EDO for n and m fixed.
    int i;
    double a, b, p, q, r, w, vp1, ap1, bp1, pp1, qp1, rp1, wp1;
    double s, h, sp1;

    sol_V[Ns] = terminal; // Terminal condition

    for (i=0; i<Ns; i++)
    {
        s = sgrid[Ns-i-1];
        sp1 = sgrid[Ns-i];
        h = sp1 - s;

        vp1 = sol_V[Ns-i];

        ap1 = coeff_A/(sp1*sp1);
        bp1 = coeff_B/sp1;
        pp1 = coeff_P[Ns-i]/(sp1*sp1);
        qp1 = coeff_Q[Ns-i]/sp1;
        rp1 = sol_R[Ns-i];
        wp1 = sol_W[Ns-i];

        a = coeff_A/(s*s);
        b = coeff_B/s;
        p = coeff_P[Ns-i-1]/(s*s);
        q = coeff_Q[Ns-i-1]/s;
        r = sol_R[Ns-i-1];
        w = sol_W[Ns-i-1];

        sol_V[Ns-i-1] = (2.*vp1 - h* ( ap1*(rp1*vp1 + wp1) + bp1*vp1 + pp1 + qp1
        / (2. + h * (a * r + b));
    }
}
```

```
static void solve_V_edo_call_american(double coeff_A, double coeff_B,
                                      double *coeff_P, double *coeff_Q,
                                      double *sgrid, int Ns, double terminal, in
                                      double *sol_R, double *sol_W, double *sol_
{
    // Solve backward the V EDO for n and m fixed.
    int i;
    double a, b, p, q, r, w, vp1, ap1, bp1, pp1, qp1, rp1, wp1;
    double s, h, sp1;


    if (index>Ns) // No early exercice
    {
        sol_V[Ns] = terminal; // Terminal condition
        for (i=0; i<Ns; i++)
        {
            s = sgrid[Ns-i-1];
            sp1 = sgrid[Ns-i];
            h = sp1 - s;

            vp1 = sol_V[Ns-i];

            ap1 = coeff_A/(sp1*sp1);
            bp1 = coeff_B/sp1;
            pp1 = coeff_P[Ns-i]/(sp1*sp1);
            qp1 = coeff_Q[Ns-i]/sp1;
            rp1 = sol_R[Ns-i];
            wp1 = sol_W[Ns-i];

            a = coeff_A/(s*s);
            b = coeff_B/s;
            p = coeff_P[Ns-i-1]/(s*s);
            q = coeff_Q[Ns-i-1]/s;
            r = sol_R[Ns-i-1];
            w = sol_W[Ns-i-1];

            sol_V[Ns-i-1] = (2.*vp1 - h* ( ap1*(rp1*vp1 + wp1) + bp1*vp1 + pp1 +
            / (2. + h * (a * r + b));
        }
    }
    else // Early exercice
```

```
    {
        for (i=index;i<Ns+1; i++)
            sol_V[i] = 1.; // Since Call(S) = S-K

        sol_V[index] = 1.; // Terminal condition is 1 (Neumann boundary conditio
        for (i=Ns-index; i<Ns; i++)
        {
            s = sgrid[Ns-i-1];
            sp1 = sgrid[Ns-i];
            h = sp1 - s;

            vp1 = sol_V[Ns-i];

            ap1 = coeff_A/(sp1*sp1);
            bp1 = coeff_B/sp1;
            pp1 = coeff_P[Ns-i]/(sp1*sp1);
            qp1 = coeff_Q[Ns-i]/sp1;
            rp1 = sol_R[Ns-i];
            wp1 = sol_W[Ns-i];

            a = coeff_A/(s*s);
            b = coeff_B/s;
            p = coeff_P[Ns-i-1]/(s*s);
            q = coeff_Q[Ns-i-1]/s;
            r = sol_R[Ns-i-1];
            w = sol_W[Ns-i-1];

            sol_V[Ns-i-1] = (2.*vp1 - h* ( ap1*(rp1*vp1 + wp1) + bp1*vp1 + pp1 +
            / (2. + h * (a * r + b));
        }
    }
}

static void solve_V_edo_put_american(double coeff_A, double coeff_B,
                                     double *coeff_P, double *coeff_Q,
                                     double *sgrid, int Ns, double initial, int
                                     double *sol_R, double *sol_W, double *sol_V
{
    // Solve forward the V EDO for n and m fixed.
    int i;
    double a, b, p, q, r, w, v, ap1, bp1, pp1, qp1, rp1, wp1;
```

```
double s, h, sp1;

if (index<0) // No early exercice
{
    sol_V[0] = initial; // Initial condition
    for (i=0; i<Ns; i++)
    {
        s = sgrid[i];
        sp1 = sgrid[i+1];
        h = sp1 - s;

        v = sol_V[i];

        a = coeff_A/(s*s);
        b = coeff_B/s;
        p = coeff_P[i]/(s*s);
        q = coeff_Q[i]/s;
        r = sol_R[i];
        w = sol_W[i];

        ap1 = coeff_A/(sp1*sp1);
        bp1 = coeff_B/sp1;
        pp1 = coeff_P[i+1]/(sp1*sp1);
        qp1 = coeff_Q[i+1]/sp1;
        rp1 = sol_R[i+1];
        wp1 = sol_W[i+1];

        sol_V[i+1] = (2. * v + h* ( a*(r*v + w) + b*v + p + q + ap1*wp1 + pp
        / (2. - h * (ap1 * rp1 + bp1));
    }
}
else    // Early exercice
{
    for (i=0;i<index+1; i++)
        sol_V[i] = -1.; // Since Put(S) = K-S

    sol_V[index] = -1.; // Initial condition is -1 (Neumann boundary conditi
    for (i=index; i<Ns; i++)
    {
        s = sgrid[i];
        sp1 = sgrid[i+1];
```

```
            h = sp1 - s;

            v = sol_V[i];

            a = coeff_A/(s*s);
            b = coeff_B/s;
            p = coeff_P[i]/(s*s);
            q = coeff_Q[i]/s;
            r = sol_R[i];
            w = sol_W[i];

            ap1 = coeff_A/(sp1*sp1);
            bp1 = coeff_B/sp1;
            pp1 = coeff_P[i+1]/(sp1*sp1);
            qp1 = coeff_Q[i+1]/sp1;
            rp1 = sol_R[i+1];
            wp1 = sol_W[i+1];

            sol_V[i+1] = (2. * v + h* ( a*(r*v + w) + b*v + p + q + ap1*wp1 + pp
            / (2. - h * (ap1 * rp1 + bp1));
        }
    }
}

//////////
// Functions to make loops.
//////////
static void solve_lines_europ(double S0, double *sgrid, int Ns,
                              double V0, double sigma_v, double alpha_v, double
                              double *vgrid, int Nv,
                              double R0, double Q0, double rho_sv,
                              double K, double H, double maturity, double rebate
                              double *tgrid, int Nt, int index_time,
                              int call_or_put,
                              double **two_C_minus2,
                              double **two_C_minus1, double **two_V_minus1,
                              double **two_C_old, double **two_V_old,
                              double **two_C_new, double **two_V_new)
{
    int m, i;
    double deltav, deltatau, time_backward, scheme;
```

```
double coeff_A, coeff_B;
double *coeff_P, *coeff_Q;
double *sol_R, *sol_W, *sol_V;
double *ptprice, *ptdelta, initial, terminal;

coeff_P = (double*) malloc((Ns+1) * sizeof(double));
coeff_Q = (double*) malloc((Ns+1) * sizeof(double));
sol_R = (double*) malloc((Ns+1) * sizeof(double));
sol_W = (double*) malloc((Ns+1) * sizeof(double));
sol_V = (double*) malloc((Ns+1) * sizeof(double));

ptprice = malloc(sizeof(double));
ptdelta = malloc(sizeof(double));

*ptprice = 0.;
*ptdelta = 0.;
deltatau = tgrid[1]-tgrid[0];
deltav = vgrid[1]-vgrid[0];
time_backward = maturity-tgrid[Nt-index_time];

scheme = (index_time<=4) ? 1 : 0;

// Loop on m
for (m=1; m<=Nv; m++)
{
    // Compute the coefficients of the EDO for all S at fixed m.
    // coeff_A is independent of S and time
    coeff_A = 2./vgrid[m] *
    (
     2. * sigma_v * sigma_v * vgrid[m]/(2. * deltav * deltav)
     + R0
     + 2. * fabs(alpha_v - beta_v * vgrid[m])/(2.* deltav)
     );
    coeff_A = 2./vgrid[m] *
    (
     2. * sigma_v * sigma_v * vgrid[m]/(2. * deltav * deltav)
     + R0
     + 2. * fabs(alpha_v - beta_v * vgrid[m])/(2.* deltav)
     );
    if (scheme==1)
    {
```

```
    coeff_A += 2./(vgrid[m]*deltatau);
}
else
{
    coeff_A += 3./(vgrid[m] * deltatau);
}

// coeff_B is independent of S and time
coeff_B = -2./vgrid[m] * (R0-Q0);

// coeff_P and coeff_Q are neither independent of S nor time
// but they depend on old values
if (m==Nv)
{
    // Fix Neumann condition "out" of the memory
    for (i=0; i<Ns+1; i++)
    {
        two_C_old[Nv+1][i]=two_C_new[Nv-1][i];
        two_V_old[Nv+1][i]=two_V_new[Nv-1][i];
    }
}

for (i=0; i<Ns+1; i++)
{

    coeff_P[i] = -2./vgrid[m] *
    (
     sigma_v * sigma_v * vgrid[m]/(2. * deltav * deltav)
     * (two_C_old[m+1][i] + two_C_new[m-1][i])
     + (
        (alpha_v - beta_v * vgrid[m]) * (two_C_old[m+1][i] - two_C_new[m
        + fabs(alpha_v - beta_v * vgrid[m]) * (two_C_old[m+1][i] + two_C
        )
     /(2. * deltav)
     );

    coeff_Q[i] =  -(two_V_old[m+1][i]-two_V_new[m-1][i]) * rho_sv * sigm


    if (scheme==1)
    {
```

```
            coeff_P[i] += -2. * two_C_minus1[m][i]/(vgrid[m] *deltatau);
        }
        else
        {
            coeff_P[i] += (-4. * two_C_minus1[m][i] + two_C_minus2[m][i])/(v
        }
    }

    // Initial data are given by black-scholes and asymptotic analysis
    // Compute the R solution at fixed m for all S.
    initial = 0.;
    solve_R_edo(coeff_A, coeff_B, sgrid, Ns, initial, sol_R);

    // Compute the W solution at fixed m for all S.

    initial = call_or_put ? 0. : K*exp(-R0*time_backward)-sgrid[0]*exp(-Q0*t
    solve_W_edo(coeff_A, coeff_P, coeff_Q, sgrid, Ns, initial, sol_R, sol_W)

    // Compute the V solution at fixed m for all S.
    terminal = (rebate-sol_W[Ns])/sol_R[Ns];
    solve_V_edo_backward(coeff_A, coeff_B, coeff_P, coeff_Q, sgrid, Ns, term

    // Compute the C solution at fixed m for all S.
    for (i=0; i<Ns+1; i++)
    {
        two_C_new[m][i] = sol_R[i] * sol_V[i] + sol_W[i];
        two_V_new[m][i] = sol_V[i];
    }
}

// First value of m is given by quadratic interpolation.
for (i=0; i<Ns+1; i++)
{
    // Price must be positive
    two_C_new[0][i]=MAX(0.,quadratic_interpolation(vgrid[1], vgrid[2], vgrid
                                                   two_C_new[1][i], two_C_ne
                                                   vgrid[0]));
    // Delta can be positive or negative
    two_V_new[0][i]=quadratic_interpolation(vgrid[1], vgrid[2], vgrid[3],
                                            two_V_new[1][i], two_V_new[2][i]
                                            vgrid[0]);
```

```
    }

    free(coeff_P);
    free(coeff_Q);
    free(sol_R);
    free(sol_W);
    free(sol_V);
    free(ptprice);
    free(ptdelta);
}

static void solve_lines_american(double S0, double *sgrid, int Ns,
                                 double V0, double sigma_v, double alpha_v, doub
                                 double *vgrid, int Nv,
                                 double R0, double Q0, double rho_sv,
                                 double K, double H, double maturity, double reb
                                 double *tgrid, int Nt, int index_time,
                                 int call_or_put,
                                 double **two_C_minus2,
                                 double **two_C_minus1, double **two_V_minus1,
                                 double **two_C_old, double **two_V_old,
                                 double **two_C_new, double **two_V_new)
{
    int m, i, index;
    double deltav, deltatau, time_backward, scheme;
    double coeff_A, coeff_B;
    double *coeff_P, *coeff_Q;
    double *sol_R, *sol_W, *sol_V;
    double *ptprice, *ptdelta, initial, terminal, boundary_condition;

    coeff_P = (double*) malloc((Ns+1) * sizeof(double));
    coeff_Q = (double*) malloc((Ns+1) * sizeof(double));
    sol_R = (double*) malloc((Ns+1) * sizeof(double));
    sol_W = (double*) malloc((Ns+1) * sizeof(double));
    sol_V = (double*) malloc((Ns+1) * sizeof(double));

    ptprice = malloc(sizeof(double));
    ptdelta = malloc(sizeof(double));

    *ptprice = 0.;
    *ptdelta = 0.;
```

```
deltatau = tgrid[1]-tgrid[0];
deltav = vgrid[1]-vgrid[0];
time_backward = maturity-tgrid[Nt-index_time];

if (call_or_put==1)
{
    // The holder can choose to exercise the option or to take the rebate
    boundary_condition = MAX(rebate, H-K);
}
else
{
    // The holder has only the rebate since payoff is 0 (i.e. K < H)
    boundary_condition = rebate;
}
scheme = (index_time<=4) ? 1 : 0;

// Loop on m
for (m=1; m<=Nv; m++)
{
    // Compute the coefficients of the EDO for all S at fixed m.
    // coeff_A is independent of S and time
    coeff_A = 2./vgrid[m] *
    (
     2. * sigma_v * sigma_v * vgrid[m]/(2. * deltav * deltav)
     + R0
     + 2. * fabs(alpha_v - beta_v * vgrid[m])/(2.* deltav)
     );
    coeff_A = 2./vgrid[m] *
    (
     2. * sigma_v * sigma_v * vgrid[m]/(2. * deltav * deltav)
     + R0
     + 2. * fabs(alpha_v - beta_v * vgrid[m])/(2.* deltav)
     );
    if (scheme==1)
    {
        coeff_A += 2./(vgrid[m]*deltatau);
    }
    else
    {
        coeff_A += 3./(vgrid[m] * deltatau);
    }
```

```
// coeff_B is independent of S and time
coeff_B = -2./vgrid[m] * (R0-Q0);

// coeff_P and coeff_Q are neither independent of S nor time
// but they depend on old values
if (m==Nv)
{
    // Fix Neumann condition "out" of the memory
    for (i=0; i<Ns+1; i++)
    {
        two_C_old[Nv+1][i]=two_C_new[Nv-1][i];
        two_V_old[Nv+1][i]=two_V_new[Nv-1][i];
    }
}

for (i=0; i<Ns+1; i++)
{

    coeff_P[i] = -2./vgrid[m] *
    (
     sigma_v * sigma_v * vgrid[m]/(2. * deltav * deltav)
     * (two_C_old[m+1][i] + two_C_new[m-1][i])
     + (
        (alpha_v - beta_v * vgrid[m]) * (two_C_old[m+1][i] - two_C_new[m
        + fabs(alpha_v - beta_v * vgrid[m]) * (two_C_old[m+1][i] + two_C
        )
     /(2. * deltav)
     );

    coeff_Q[i] =  -(two_V_old[m+1][i]-two_V_new[m-1][i]) * rho_sv * sigm


    if (scheme==1)
    {
        coeff_P[i] += -2. * two_C_minus1[m][i]/(vgrid[m] *deltatau);
    }
    else
    {
        coeff_P[i] += (-4. * two_C_minus1[m][i] + two_C_minus2[m][i])/(v
    }
```

```
}

if (call_or_put==1)
{
    // Initial data are given by black-scholes and asymptotic analysis
    // Compute the R solution at fixed m for all S.
    initial = 0.;
    solve_R_edo(coeff_A, coeff_B, sgrid, Ns, initial, sol_R);

    // Compute the W solution at fixed m for all S.

    initial = MAX(sgrid[0]-K,0.);
    solve_W_edo(coeff_A, coeff_P, coeff_Q, sgrid, Ns, initial, sol_R, so
    // In American case, monitor the function R+W-(S-K).
    index = find_early_exercice(K, sgrid, Ns, call_or_put, sol_R, sol_W)
    // Compute backward the V solution at fixed m for all S.
    terminal = (MAX(H-K, rebate)-sol_W[Ns])/sol_R[Ns];
    solve_V_edo_call_american(coeff_A, coeff_B, coeff_P, coeff_Q, sgrid,
                              terminal, index, sol_R, sol_W, sol_V);
    // Compute the C solution at fixed m for all S. But monitor value.
    for (i=0; i<index; i++)
    {
        two_C_new[m][i] = sol_R[i] * sol_V[i] + sol_W[i];
        two_V_new[m][i] = sol_V[i];
    }
    for (i=index; i<Ns+1; i++)
    {
        two_C_new[m][i] = MAX(sgrid[i]-K,0.);
        two_V_new[m][i] = 1;
    }
}
else
{
    // Terminal data are given by black-scholes and asymptotic analysis
    // Compute the R solution at fixed m for all S.
    terminal = 0.;
    solve_R_edo_backward(coeff_A, coeff_B, sgrid, Ns, terminal, sol_R);

    // Compute the W solution at fixed m for all S.
    terminal = rebate;
    solve_W_edo_backward(coeff_A, coeff_P, coeff_Q, sgrid, Ns, terminal,
```

```
        // In American case, monitor the function -R+W-(K-S).
        index = find_early_exercice(K, sgrid, Ns, call_or_put, sol_R, sol_W)
        // Compute forward the V solution at fixed m for all S.
        initial = (MAX(K-sgrid[0],K*exp(-R0*time_backward)-sgrid[0]*exp(-Q0*
        solve_V_edo_put_american(coeff_A, coeff_B, coeff_P, coeff_Q, sgrid,
                                 initial, index, sol_R, sol_W, sol_V);

        // Compute the C solution at fixed m for all S. But monitor value.
        for (i=Ns; i>index; i--)
        {
            two_C_new[m][i] = sol_R[i] * sol_V[i] + sol_W[i];
            two_V_new[m][i] = sol_V[i];
        }
        for (i=index; i>-1; i--)
        {
            two_C_new[m][i] = MAX(K-sgrid[i],0.);
            two_V_new[m][i] = -1.;
        }
    }
}

// First value of m is given by quadratic interpolation.
for (i=0; i<Ns+1; i++)
{
    // Price must be positive
    two_C_new[0][i]=MAX(0.,quadratic_interpolation(vgrid[1], vgrid[2], vgrid
                                                   two_C_new[1][i], two_C_ne
                                                   vgrid[0]));
    // Delta can be positive or negative
    two_V_new[0][i]=quadratic_interpolation(vgrid[1], vgrid[2], vgrid[3],
                                            two_V_new[1][i], two_V_new[2][i]
                                            vgrid[0]);
}

free(coeff_P);
free(coeff_Q);
free(sol_R);
free(sol_W);
free(sol_V);
free(ptprice);
```

```
    free(ptdelta);
}


static int FDCHIARELLA_BarrierHeston(double rebate, double H, int am, NumFunc_1
                                     double *ptprice, double *ptdelta)
{
    double K;
    int call_or_put;
    double alpha_v,beta_v;
    double *sgrid, *vgrid, *tgrid;
    double Smin, Sleft, Sright, Srightright, Smax, Coeff_s, Vmax;
    int n, m, i;

    double error;
    int cpt_error;

    // Variables to compute price and delta.
    int IndexS, IndexV;
    double price_sd_vd, price_sd_vu;
    double price_su_vd, price_su_vu;

    // 2-vectors
    double **two_C_minus2;
    double **two_V_minus1;
    double **two_C_minus1;
    double **two_V;
    double **two_C;
    double **two_C_temp;
    double **two_V_temp;

    // 3-vectors
    double ***three_C;
    double ***three_V;

    K = p->Par[0].Val.V_PDOUBLE;
    if ((p->Compute) == &Call)
        call_or_put=1;
    else
        call_or_put=-1;
```

```
alpha_v = kappa_v*theta_v;
beta_v = kappa_v;

// Variance
Vmax = MIN(MAX(100.*V0,1.),5.);
// Asset
Coeff_s = 0.8; // Between 0 and 1 strictly.
Smin = 0.6 * S0;
Sleft = 0.95 * S0;
Sright = MIN(1.05*K,0.98*H);
Srightright = MAX(1.05*K,0.98*H);
Smax = H;

////////////////////////////////////
// Compute sgrid, vgrid and tgrid. //
////////////////////////////////////
// Memory allocation of 1-vectors.
sgrid=(double *)malloc((Ns+1)*sizeof(double));
vgrid=(double *)malloc((Nv+1)*sizeof(double));
tgrid=(double *)malloc((Nt+1)*sizeof(double));
grid_generation_asset(sgrid, Smin, Sleft, Sright, Srightright, Smax, Ns, Coe
grid_generation_uniform(vgrid, 0., Vmax, Nv);
grid_generation_uniform(tgrid, 0., maturity, Nt);

//////////////////////////
// Memory allocations. //
//////////////////////////
{
    // Memory allocation of 2-vectors.
    two_C_minus2 = (double**) malloc((Nv+1) * sizeof(double*));
    two_C_minus1 = (double**) malloc((Nv+1) * sizeof(double*));
    two_V_minus1 = (double**) malloc((Nv+1) * sizeof(double*));
    two_C = (double**) malloc((Nv+2) * sizeof(double*)); // Be careful, size
    two_V = (double**) malloc((Nv+2) * sizeof(double*)); // Be careful, size
    two_C_temp = (double**) malloc((Nv+1) * sizeof(double*));
    two_V_temp = (double**) malloc((Nv+1) * sizeof(double*));
    for (m=0; m<Nv+1; m++)
    {
        two_C_minus2[m] = (double*) malloc((Ns+1) * sizeof(double));
        two_C_minus1[m] = (double*) malloc((Ns+1) * sizeof(double));
        two_V_minus1[m] = (double*) malloc((Ns+1) * sizeof(double));
```

```
        two_C[m] = (double*) malloc((Ns+1) * sizeof(double));
        two_V[m] = (double*) malloc((Ns+1) * sizeof(double));
        two_C_temp[m] = (double*) malloc((Ns+1) * sizeof(double));
        two_V_temp[m] = (double*) malloc((Ns+1) * sizeof(double));
    }
    two_C[Nv+1] = (double*) malloc((Ns+1) * sizeof(double));
    two_V[Nv+1] = (double*) malloc((Ns+1) * sizeof(double));

    // Memory allocation of 3-vectors.
    three_C = (double***) malloc((Nt+1) * sizeof(double**));
    three_V = (double***) malloc((Nt+1) * sizeof(double**));
    for (n=0; n<Nt+1; n++)
    {
        three_C[n] = (double**) malloc((Nv+1) * sizeof(double*));
        three_V[n] = (double**) malloc((Nv+1) * sizeof(double*));
        for (m=0; m<Nv+1; m++)
        {
            three_C[n][m] = (double*) malloc((Ns+1) * sizeof(double));
            three_V[n][m] = (double*) malloc((Ns+1) * sizeof(double));
        }
    }
}// End of memory allocations.

// Initialization.
{
    for (m=0; m<Nv+1; m++)
    {
        for (i=0; i<Ns+1; i++)
        {
            two_C_minus2[m][i] = 0.;
            two_C_minus1[m][i] = 0.;
            two_V_minus1[m][i] = 0.;
            two_C[m][i] = 0.;
            two_V[m][i] = 0.;
            two_C_temp[m][i] = 0.;
            two_V_temp[m][i] = 0.;
        }
    }
    for (i=0; i<Ns+1; i++)
    {
        two_C[Nv+1][i] = 0.;
```

```
        two_V[Nv+1][i] = 0.;
    }
    for (n=0; n<Nt+1; n++)
    {
        for (m=0; m<Nv+1; m++)
        {
            for (i=0; i<Ns+1; i++)
            {
                three_C[n][m][i] = 0.;
                three_V[n][m][i] = 0.;
            }
        }
    }

    // Terminal values
    for (m=0; m<Nv+1; m++) {
        for (i=0; i<Ns+1; i++) {
            // Terminal condition for barrier
            if (call_or_put==1)
            {
                three_C[0][m][i] = MAX(sgrid[i]-K,0.);
                three_V[0][m][i] = (sgrid[i]<K) ? 0. : 1.;
            }
            else
            {
                three_C[0][m][i] = MAX(K-sgrid[i],0.);
                three_V[0][m][i] = (sgrid[i]<K) ? -1. : 0.;
            }
        }
    }
}

// Loop over the time
for (n=1; n<Nt+1; n++)
{
    // Record old values in corresponding arrays
    for (m=0; m<Nv+1; m++)
    {
        for (i=0; i<Ns+1; i++)
        {
            if (n>=2)
```

```
            two_C_minus2[m][i] = three_C[n-2][m][i];
        two_C_minus1[m][i] = three_C[n-1][m][i];
        two_V_minus1[m][i] = three_V[n-1][m][i];
        two_C[m][i] = three_C[n-1][m][i];
        two_V[m][i] = three_V[n-1][m][i];
        two_C_temp[m][i] = three_C[n-1][m][i];
        two_V_temp[m][i] = three_V[n-1][m][i];
    }
}

error = 1.;
cpt_error = 0;
while ((error > 0.00000001) && (cpt_error++<100))
{
    // One loop over all variance grid.
    if (am==1)
    {
        solve_lines_american (S0, sgrid, Ns,
                              V0, sigma_v, alpha_v, beta_v,
                              vgrid, Nv,
                              R0, Q0, rho_sv,
                              K, H, maturity, rebate,
                              tgrid, Nt, n,
                              call_or_put,
                              two_C_minus2,
                              two_C_minus1, two_V_minus1,
                              two_C, two_V,
                              two_C_temp, two_V_temp);
    }
    else
    {
        solve_lines_europ (S0, sgrid, Ns,
                           V0, sigma_v, alpha_v, beta_v,
                           vgrid, Nv,
                           R0, Q0, rho_sv,
                           K, H, maturity, rebate,
                           tgrid, Nt, n,
                           call_or_put,
                           two_C_minus2,
                           two_C_minus1, two_V_minus1,
                           two_C, two_V,
```

```
                                two_C_temp, two_V_temp);
            }

            // Compute the error and swap temporary arrays.
            error=0.;
            for (m=0; m<Nv+1; m++) {
                for (i=0; i<Ns+1; i++) {
                    error = MAX(error,fabs(two_C_temp[m][i]-two_C[m][i]));
                    error = MAX(error,fabs(two_V_temp[m][i]-two_V[m][i]));
                    two_C[m][i] = two_C_temp[m][i];
                    two_V[m][i] = two_V_temp[m][i];
                }
            }
        }// End of loop on error.

        // First value of m is given by quadratic interpolation.
        for (i=0; i<Ns+1; i++)
        {
            // Price must be positive
            two_C[0][i]=MAX(0.,quadratic_interpolation(vgrid[1], vgrid[2], vgrid
                                                      two_C[1][i], two_C[2][i],
                                                      vgrid[0]));
            // Delta can be positive or negative
            two_V[0][i]=quadratic_interpolation(vgrid[1], vgrid[2], vgrid[3],
                                               two_V[1][i], two_V[2][i], two_V[
                                               vgrid[0]);
        }

        // Accept the values and record them.
        for (m=0; m<Nv+1; m++) {
            for (i=0; i<Ns+1; i++) {
                three_C[n][m][i] = two_C[m][i] ;
                three_V[n][m][i] = two_V[m][i];
            }
        }
    }

    // Index in the domain for the price.
    // Find the index in sgrid corresponding to the value S0 of the asset.
    IndexS = lower_index(sgrid,Ns+1,S0);
    // Find the index in vgrid corresponding to the value V0 of the variance.
```

```
IndexV = lower_index(vgrid,Nv+1,V0);

// First compute the delta (using price as temporary variable). We do an int
price_sd_vd = three_C[Nt][IndexV][IndexS+1];
price_sd_vu = three_C[Nt][IndexV+1][IndexS+1];
price_su_vd = three_C[Nt][IndexV][IndexS+2];
price_su_vu = three_C[Nt][IndexV+1][IndexS+2];

*ptprice=double_interpolation(price_sd_vd, price_sd_vu, price_su_vd, price_s
                              sgrid[IndexS+1], sgrid[IndexS+2],
                              vgrid[IndexV], vgrid[IndexV+1],
                              sgrid[IndexS+1], V0);

price_sd_vd = three_C[Nt][IndexV][IndexS];
price_sd_vu = three_C[Nt][IndexV+1][IndexS];
price_su_vd = three_C[Nt][IndexV][IndexS+1];
price_su_vu = three_C[Nt][IndexV+1][IndexS+1];

*ptdelta =(*ptprice - double_interpolation(price_sd_vd, price_sd_vu, price_s
                                           sgrid[IndexS], sgrid[IndexS+1],
                                           vgrid[IndexV], vgrid[IndexV+1],
                                           sgrid[IndexS], V0)     )
/(sgrid[IndexS+1] - sgrid[IndexS]);

// Next compute the price. We do an interpolation.
*ptprice=double_interpolation(price_sd_vd, price_sd_vu, price_su_vd, price_s
                              sgrid[IndexS], sgrid[IndexS+1],
                              vgrid[IndexV], vgrid[IndexV+1],
                              S0, V0);

// Memory deallocations.
{
    // Memory deallocation of 3-vectors.
    for (n=0; n<Nt+1; n++)
    {
        for (m=0; m<Nv+1; m++)
        {
            free(three_C[n][m]);
            free(three_V[n][m]);
        }
        free(three_C[n]);
```

```
            free(three_V[n]);
        }
        free(three_C);
        free(three_V);

        // Memory deallocation of 2-vectors.
        for (m=0; m<Nv+1; m++)
        {
            free(two_C[m]);
            free(two_V[m]);
            free(two_C_temp[m]);
            free(two_V_temp[m]);
            free(two_C_minus1[m]);
            free(two_V_minus1[m]);
            free(two_C_minus2[m]);
        }
        free(two_C[Nv + 1]);
        free(two_V[Nv + 1]);
        free(two_C);
        free(two_V);
        free(two_C_temp);
        free(two_V_temp);
        free(two_C_minus1);
        free(two_V_minus1);
        free(two_C_minus2);
    }// End of memory deallocations.

    free(sgrid);
    free(vgrid);
    free(tgrid);

    return OK;

}

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

```
    if (ptMod->Sigma.Val.V_PDOUBLE == 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
        limit = ((ptOpt->Limit.Val.V_NUMFUNC_1)->Compute)((ptOpt->Limit.Val.V_NU


        return FDCHIARELLA_BarrierHeston(rebate, limit, ptOpt->EuOrAm.Val.V_BOOL
                                         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->Par[2].Val.V_PINT,
                                         &(Met->Res[0].Val.V_DOUBLE),
                                         &(Met->Res[1].Val.V_DOUBLE));
    }

}


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

    if ((opt->OutOrIn).Val.V_BOOL == OUT)
        if ((opt->DownOrUp).Val.V_BOOL == UP)
            if ((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_chiarella_barrierheston";
        Met->Par[0].Val.V_INT = 50;
        Met->Par[1].Val.V_INT = 500;
        Met->Par[2].Val.V_INT = 100;
    }

    return OK;
}

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