

## Help

```
#include "bshw1d_stda.h"
#include "enums.h"
#include "error_msg.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2015+2) //The "#els
static int CHK_OPT(FD_HybridTree_GLWB_BSHW)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(FD_HybridTree_GLWB_BSHW)(void *Opt, void *Mod, PricingMethod *Met)
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else

/*Vettori*/
static char *infilename;
static double gamma_i;
static int ratchet;
static int flat_flag;
static double tt, s0, r0, divid, kappa, omega, rho, sigma, theta_sc;
static double alpha_m;
static int Nt, N;
static double **r, * *discount, * *Q, * *P_old, * *P_old1, * *P_new;
static double **y;
static int **y_down, * *y_up;
static double **pu_y, * *pd_y;
static double lm_insurance[500];
static double lm_age[500];
static double Rt[500];
static double Mt[500];
static double Kt[500];
static double Bt[500];
static double Gt[500];
static int n_step_for_period;

/*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 *initial_yield;
static double **r, *Pc, * *discount;
static double *shift;
static char *init_tr;

#define MAXIT 30

static double rtsec(double (*func)(double), double x1, double x2, double xacc)
{
    int j;
    double fl, f, dx, swap, xl, rts;

    fl = (*func)(x1);
    f = (*func)(x2);
    if (fabs(fl) < fabs(f))
    {
        rts = x1;
        xl = x2;
        swap = fl;
        fl = f;
        f = swap;
    }
    else
    {
        xl = x1;
        rts = x2;
    }
    for (j = 1; j <= MAXIT; j++)
    {
        dx = (xl - rts) * f / (f - fl);
        xl = rts;
        fl = f;
        rts += dx;
        f = (*func)(rts);
        if (fabs(dx) < xacc || f == 0.0) return rts;
    }
    printf("Maximum number of iterations exceeded in rtsec\ n");
    return 0.0;
}

#undef MAXIT
```

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

    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;
    }

    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;
}

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

y_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_old1 = (double **)malloc((N + 1) * sizeof(double *));
for (i = 0; i <= N; i++)
    P_old1[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;

    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(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 < N + 1; i++)
        free(P_old[i]);
    free(P_old);

    for (i = 0; i < N + 1; i++)
        free(P_old1[i]);
    free(P_old1);

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

    return;
}
```

```

/*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(pligne, "%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;
}

static double compute_S(double Y, double rv, double sigma, double omega, double
{
    double val;

```

```

    val = exp(Y + rho * sigma * rv);

    return val;
}

/*Compute Price Bond*/
static double compute_price(double alpha_g)
{
    int i, j, k;
    double stock;
    int fv_up, fv_down;
    double l;
    double alpha, beta, gamma, alpha1, beta1, gamma1;
    int PriceIndex;
    double dx;
    double *A, *B, *C, *A1, *B1, *C1, *Price, *S, *vect_y;
    double dt;
    double z, vv;
    double bound1, bound2;
    int Index;
    double A_IN, new_A, new_P = 0.;
    int anno, flag_monit, i_s;
    double s_value, s_value1;
    int current_mt_year, current_mt_year1;
    double log_s0;
    double val, val1, val_s, val_sim;
    double price;

    A_IN = s0;

    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));

    vect_y = (double *)malloc((N + 1) * sizeof(double));
    Price = (double *)malloc((N + 1) * sizeof(double));
    S = (double *)malloc((N + 1) * sizeof(double));

```

```

dt = tt / (double)Nt;
double sigma2 = SQR(sigma);

double PRECISION_FDH=1.0e-5;
l=sigma*sqrt(tt)*sqrt(log(1.0/PRECISION_FDH))+fabs((r0-divid-0.5*sigma2)*tt);

dx = 2.0 * l / (double)N;
log_s0 = log(s0)-sigma*rho/omega*r[0][0];

for (j = 0; j <= N; j++)
{
    vect_y[j] = log_s0 - l + (double)j * dx;
}

/*Maturity conditions*/
for (k = 0; k <= Nt; k++)
    for (j = 0; j <= N; j++)
    {
        stock = compute_S(vect_y[j], y[Nt][k], sigma, omega, rho);
        P_new[j][k] = stock * Rt[56];
        P_new[j][k] = 0.;
    }
bound1 = 0.;
bound2 = 0.;

anno = (int)tt;

/*Dynamic Programming*/
for (i = Nt - 1; i >= 0; i--)
{

    current_mt_year1 = (int)((i + 1) * dt);
    current_mt_year = (int)((i) * dt);

    if (((i + 1) % n_step_for_period) == 0) && ((i + 1) != Nt) && (i > 0))
    {
        flag_monit = 1;
        anno = anno - 1;
    }
    else flag_monit = 0;
}

```

```

//Monitoring
if (flag_monit)
{
    for (k = 0; k <= i; k++)
    {
        for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
            Price[PriceIndex] = P_new[PriceIndex][k];

        if (ratchet)
        {
            for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
            {
                val_s = compute_S(vect_y[PriceIndex], y[i][k], sigma, omega);
                new_A = MAX(val_s, A_IN);
                s_value = A_IN / new_A * val_s;
                i_s = 0;
                while ((compute_S(vect_y[i_s], y[i][k], sigma, omega, rho)

                if (i_s == 0)
                    new_P = Price[1];
                else if (i_s >= N)
                    new_P = Price[N - 1];
                else
                {
                    val = Price[i_s];
                    val1 = Price[i_s - 1];
                    val_s = compute_S(vect_y[i_s], y[i][k], sigma, omega,
                    val_sim = compute_S(vect_y[i_s - 1], y[i][k], sigma, o
                    new_P = val + (val - val1) * (s_value - val_s) / (val_
                }

                P_new[PriceIndex][k] = new_A / A_IN * new_P;
            Price[PriceIndex]=P_new[PriceIndex][k];
            }
        }

        if (fabs(gamma_i) < 0.000001) //Bonus Event
        {
            for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
            {

```

```

new_A = A_IN * (1 + Bt[anno]);
val_s = compute_S(vect_y[PriceIndex], y[i][k], sigma, omega, rho);
s_value = A_IN / new_A * val_s;
if (i_s == 0)
    new_P = Price[1];
else if (i_s >= N)
    new_P = Price[N - 1];
else
{
    val = Price[i_s];
    val1 = Price[i_s - 1];
    val_s = compute_S(vect_y[i_s], y[i][k], sigma, omega, rho);
    val_sim = compute_S(vect_y[i_s - 1], y[i][k], sigma, omega, rho);
    new_P = val + (val - val1) * (s_value - val_s) / (val_sim - val1);
}
}
P_new[PriceIndex][k] = MAX(P_new[PriceIndex][k], new_A / A_IN);
}
else if (gamma_i <= 1.0) //Withdrawal not exceeding contract amount
{
    for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
    {
        val_s = compute_S(vect_y[PriceIndex], y[i][k], sigma, omega, rho);
        s_value = MAX(val_s - gamma_i * Gt[anno] * A_IN, 0.);
        i_s = 0;
        while ((compute_S(vect_y[i_s], y[i][k], sigma, omega, rho) > s_value))
        {
            if (i_s == 0)
                new_P = Price[1];
            else if (i_s >= N)
                new_P = Price[N - 1];
            else
            {
                val = Price[i_s];
                val1 = Price[i_s - 1];
                val_s = compute_S(vect_y[i_s], y[i][k], sigma, omega, rho);
                val_sim = compute_S(vect_y[i_s - 1], y[i][k], sigma, omega, rho);
                new_P = val + (val - val1) * (s_value - val_s) / (val_sim - val1);
            }
        }
        P_new[PriceIndex][k] = new_P + Rt[anno] * gamma_i * Gt[anno];
    }
}

```

```

    }
    else if (gamma_i > 1.) //Partial or full surrender
        //gamma_i=2 is full surrender
        {
            for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
            {
                new_A = A_IN * (2 - gamma_i);
                val_s = compute_S(vect_y[PriceIndex], y[i][k], sigma, omega);
                s_value1 = MAX(val_s - Gt[anno] * A_IN, 0.);
                s_value = s_value1 * (2 - gamma_i);
                if (i_s == 0)
                    new_P = Price[1];
                else if (i_s >= N)
                    new_P = Price[N - 1];
                else
                {
                    val = Price[i_s];
                    val1 = Price[i_s - 1];
                    val_s = compute_S(vect_y[i_s], y[i][k], sigma, omega);
                    val_sim = compute_S(vect_y[i_s - 1], y[i][k], sigma, omega);
                    new_P = val + (val - val1) * (s_value - val_s) / (val_s - val_sim);
                }
                P_new[PriceIndex][k] = MAX(P_new[PriceIndex][k], new_A / A_IN);
            }
        }
    } //end k
} //End Monitoring

for (k = 0; k <= i; k++)
{
    z = r[i][k] - alpha_g - alpha_m - 0.5 * SQR(sigma) + sigma * rho * kap;
    vv = 0.5 * SQR(sigma) * (1. - SQR(rho));

    //Fully Implicit Scheme

    /*Lhs Factor of the fully implicit scheme*/
    alpha = theta_sc * (-vv * dt / SQR(dx) + z * dt / (2.*dx));
    beta = 1 + theta_sc * vv * 2 * dt / SQR(dx);
    gamma = theta_sc * (-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;

/*Rhs Factors*/
alpha1 = (1. - theta_sc) * (vv * dt / SQR(dx) - z * dt / (2.*dx));
beta1 = 1 - (1. - theta_sc) * vv * 2 * dt / SQR(dx);
gamma1 = (1. - theta_sc) * (vv * dt / SQR(dx) + z * dt / (2 * dx));

for (PriceIndex = 1; PriceIndex <= N - 1; PriceIndex++)
{
    A1[PriceIndex] = alpha1;
    B1[PriceIndex] = beta1;
    C1[PriceIndex] = gamma1;
}

B1[1] = beta1 + alpha1;
B1[N - 1] = beta1 + 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 = y_up[i][k];
fv_down = y_down[i][k];

//Initialise
for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
{
    Price[PriceIndex] = pu_y[i][k] * P_new[PriceIndex][k + fv_up] + pd_y[
}

```

```

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

//Add M(t)S
for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
{
    S[PriceIndex] += compute_S(vect_y[PriceIndex], y[i][k], sigma, ome

}

/*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_old[PriceIndex][k] = discount[i][k] * Price[PriceIndex];
}

} //end k

for (j = 0; j <= N; j++)
    for (k = 0; k <= i; k++)
        P_new[j][k] = P_old[j][k];

} //end i

Index = (int) floor((double)N / 2.0);
price = P_new[Index][0] - s0;

```

```

//Memory Desallocation
free(A);
free(B);
free(C);
free(A1);
free(B1);
free(C1);
free(vect_y);
free(S);
free(Price);

return price;

}

/*Compute Price Option*/
int Fd_HybridTree_GLWB_BsHw(double s, double t_fixed, double divid_fixed, double
{

    int i,j, iter;
    double val, sum, sum1;
    double pr1, pr2, pracc;
    FILE *f_in;

    init_tr = curve;

    n_step_for_period = n_step_for_year;
    Nt = (int)(t_fixed * n_step_for_year);
    N = N_space;
    theta_sc = theta;

    tt = t_fixed;
    s0 = s;
    divid = divid_fixed;
    sigma = sigma_fixed;
    alpha_m = alpha_m_fixed;
    ratchet = ratchet_fixed;
    gamma_i = gamma_i_fixed;
    r0 = r0_fixed;
    kappa = kappa_fixed;
    omega = omega_fixed;

```

```

rho = rho_fixed;
flat_flag = flat_flag_fixed;

f_in = fopen(infile_name, "rb");

for (i = 0; i <= 60; i++)
{
    fscanf(f_in, "%lf\ n", &val);
    lm_insurance[i] = val;
    lm_age[i] = 65. + i;
}

sum = 1.;
Mt[0] = lm_insurance[0];
for (i = 1; i <= 60; i++)
{
    sum *= (1. - lm_insurance[i - 1]);
    Mt[i] = lm_insurance[i] * sum;
}

Rt[0] = 1.;
sum1 = 0;
for (i = 1; i <= 60; i++)
{
    sum1 += Mt[i - 1];
    Rt[i] = 1. - sum1;
}

//Surrender fee Kt
for (i = 0; i <= 60; i++)
{
    j=0;
    while((pnl_vect_get(timesKt_fixed,j)<(double)i)&&(j<=4))
j++;
    Kt[i] = pnl_vect_get(Kt_fixed,j);
}

for (i = 1; i <= 60; i++)
    Bt[i] = B_i_fixed;

```

```

    for (i = 1; i <= 60; i++)
        Gt[i] = Gt_i_fixed;

    /*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);
    iter = 1;

    if (iter)
    {
        pracc = 1.e-8;
        pr1 = 0.;
        pr2 = 0.1;
        val = rtsec(compute_price, pr1, pr2, pracc);
    }

    /*Price*/
    *ptprice = val;

    fclose(f_in);

    free_memory(Nt, N);

    return OK;
}

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

    divid = log(1. + ptMod->divid.Val.V_DOUBLE / 100.);
    infilename = ptOpt->MortalityData.Val.V_FILENAME;

```

```

return Fd_HybridTree_GLWB_BsHw(ptMod->S0.Val.V_PDOUBLE,
                                ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
                                ptMod->flat_flag.Val.V_INT,
                                MOD(GetYield)(ptMod),
                                MOD(GetCurve)(ptMod),
                                ptMod->kr.Val.V_PDOUBLE,
                                ptMod->Sigmar.Val.V_PDOUBLE,
                                ptMod->RhoSr.Val.V_PDOUBLE,
                                Met->Par[0].Val.V_PINT,
                                Met->Par[1].Val.V_PINT,
                                Met->Par[2].Val.V_RGDOUBLE051,
                                &(Met->Res[0].Val.V_DOUBLE));
}

static int CHK_OPT(FD_HybridTree_GLWB_BSHW)(void *Opt, void *Mod)
{
    if ((strcmp(((Option *)Opt)->Name, "GLWB") == 0))
        return OK;
    else
        return WRONG;
}
#endif //PremiaCurrentVersion

static int MET(Init)(PricingMethod *Met, Option *Opt)
{
    if (Met->init == 0)
    {
        Met->init = 1;
        Met->HelpFilenameHint = "fd_hybridtree_glwb_bshw";
        Met->Par[0].Val.V_INT = 10;
        Met->Par[1].Val.V_INT = 200;
        Met->Par[2].Val.V_RGDOUBLE = 0.5;
    }

    return OK;
}

PricingMethod MET(FD_HybridTree_GLWB_BSHW) =
{
    "FD_HybridTree_GLWB_BSHW",
    {

```

```

        {"Timestep_for_year", INT2, {100}, ALLOW}, {"SpaceStepNumber", INT2, {100},
        {"Theta", RGDOUBLE051, {100}, ALLOW},
        {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CALC(FD_HybridTree_GLWB_BSHW),
    { {"Fair Fee", DOUBLE, {100}, FORBID},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CHK_OPT(FD_HybridTree_GLWB_BSHW),
    CHK_ok,
    MET(Init)
};

```