

[Help](#)

```
#include "bshw1d_std.h"
#include "enums.h"
#include "error_msg.h"

#include "pnl/pnl_random.h"
#include "pnl/pnl_cdf.h"

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

static double **Q;
static double **y;
static int **y_down,**y_up;
static double **pu_y,**pd_y;
static double **r,**discount;
static double *initial_yield;

/*ZCB Data*/
static double* tm; /*Times T of maturities read in the file initialyield.dat */
static double* Pm; /*Values of the zero coupon P(0,tm) read in the file initialyi
int n_price1;

static double *shift,*Pc;
static char *init_tr;

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

```
    }

    return OK;
}

static void free_memory(int Nt)
{
    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);

    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] +
                                t[i] * Pm[j] / (tm[j + 1] - tm[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[iF] - t[i]) * (t[iF] - t[i]);
        }
    }
}

/*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;
}

/*Compute Price Option*/
int Mc_HybridTree_BsHw(double s0, NumFunc_1 *p, double tt, double divid, int f
{

    double price_sample, mean_price,var_price;
    int init_mc,ipath,i,k1,k2;
    int simulation_dim;
    double alpha, z_alpha;
    double pterror_price;
    double int_r,interest_rate,y_s;
    double dt,sqrt_dt;
    double g1,w_t_1;
    double Yt,St;
    double rho3;
    double new_y_s;
    double mu_z;
    double sigma_z;

    init_tr = curve;

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

    //Tree construction for r
    tree_r(tt, r0, kappa, omega, Nt);
    calibration_bond(flat_flag, tt, r0, omega, Nt);

    /* Value to construct the confidence interval */
    alpha= (1.- confidence)/2.;

```

```

z_alpha=pnl_inv_cdfnor(1.- alpha);

/*Initialisation*/
mean_price= 0.0;
var_price= 0.0;

dt=tt/(double)Nt;
sqrt_dt=sqrt(dt);
/* Test after initialization for the generator */
simulation_dim =Nt;

/* MC sampling */
init_mc = pnl_rand_init(generator, simulation_dim, N_MC);
rho3=sqrt(1-SQR(rhoSr));
for(ipath= 1;ipath<=N_MC;ipath++)
{
    interest_rate=r[0][0];
    y_s=y[0][0];

    Yt=log(s0);
    k1=0;
    k2=0;
    int_r=0;
    for(i=0;i<Nt;i++)
    {
        g1=pnl_rand_normal(generator);//V
        w_t_1=sqrt_dt*g1;//V
        interest_rate=r[i][k1];

        //Simulate r and y
        if(pnl_rand_uni(generator)<pu_y[i][k1])
            k1+=y_up[i][k1];
        else
            k1+=y_down[i][k1];

        new_y_s=y[i+1][k1];

        mu_z=interest_rate-divid-0.5*SQR(sigma)+sigma*rhoSr*kappa*y_s;
        sigma_z=rho3*sigma;

        Yt+=mu_z*dt+sigma_z*w_t_1+rhoSr*sigma*(new_y_s-y_s);
    }
}

```

[illegible]

```

        ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE, di
        ptMod->flat_flag.Val.V_INT,
        MOD(GetYield)(ptMod),
        MOD(GetCurve)(ptMod),
        ptMod->kr.Val.V_PDOUBLE,
        ptMod->Sigmar.Val.V_PDOUBLE,
    ptMod->SigmaS.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_ENUM.value,
    Met->Par[3].Val.V_PDOUBLE,
        &(Met->Res[0].Val.V_DOUBLE), &(Met->Res[1].Val.V_DO
}

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

static int MET(Init)(PricingMethod *Met, Option *Opt)
{
    if (Met->init == 0)
    {
        Met->init = 1;
        Met->HelpFilenameHint = "mc_hybridtree_bshw";
        Met->Par[0].Val.V_INT = 300;
        Met->Par[1].Val.V_INT = 100000;
        Met->Par[2].Val.V_ENUM.value = 0;
        Met->Par[2].Val.V_ENUM.members = &PremiaEnumMCRNGs;
        Met->Par[3].Val.V_DOUBLE = 0.95;
    }

    return OK;
}

PricingMethod MET(MC_HybridTree_BSHW) =
{

```

```

    "MC_HybridTree",
    { {"N steps time", INT, {100}, ALLOW},
      {"N Iterations", INT, {100}, ALLOW},
{"RandomGenerator", ENUM, {100}, ALLOW},
      {"Confidence Value", DOUBLE, {100}, ALLOW},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CALC(MC_HybridTree_BSHW),
    { {"Price", DOUBLE, {100}, FORBID},
{"Inf Price", DOUBLE, {100}, FORBID},
      {"Sup Price", DOUBLE, {100}, FORBID} ,
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CHK_OPT(MC_HybridTree_BSHW),
    CHK_ok,
    MET(Init)
};

```