

[Help](#)

```
#include <stdio.h>
#include <stdlib.h>
#include <
href../../../../common/math/cdo/cdo_math_h_src.pdfmath.h>

#include "
href../../../../mod/copula/copula_stdndc/copula_stdndc_h_src.pdfcopula_stdndc.h"
#include "pnl/pnl_matrix.h"
#include "pnl/pnl_cdf.h"
#include "pnl/pnl_random.h"
#include "
href../../../../common/math/cdo/cdo_h_src.pdfmath/cdo/cdo.h"
#include "
href../../../../mod/copula/copula_stdndc/price_cdo_h_src.pdfprice_cdo.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2007+2) //The "#els
static int CHK_OPT(HullWhite)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(HullWhite)(void *Opt, void *Mod, PricingMethod *Met)
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else
double          **hw_numdef(const CDO          *cdo,
                           const copula        *cop,
                           const grid          *t,
                           const cond_prob     *cp)
{
    double        **nd;
    double        *U;
    double        *V;
    double        p0;
    double        w_jn;
    int            jv;
    int            jV;
    int            jt;
    int            jn;
```

```

int          jk;
FILE         *data;

nd = malloc(t->size * sizeof(double *));
U = malloc((cdo->n_comp + 1) * sizeof(double));
V = malloc((cdo->n_comp + 1) * sizeof(double));
for (jt = 0; jt < t->size; jt++)
{
    nd[jt] = malloc((cdo->n_comp + 1) * sizeof(double));
    for (jV = 0; jV < (cdo->n_comp + 1); jV++) nd[jt][jV] = 0.;
    for (jv = 0; jv < cop->size; jv++)
    {
        p0 = 1.;
        for (jn = 0; jn < cdo->n_comp; jn++) p0 *= (1. - cp->p[jn][jt][jv]);
        nd[jt][0] += p0 * cop->weights[jv];
        U[0] = 1.;
        for (jV = 1; jV < (cdo->n_comp + 1); jV++)
        {
            V[jV] = 0;
            for (jn = 0; jn < cdo->n_comp; jn++)
            {
                w_jn = cp->p[jn][jt][jv] / (1. - cp->p[jn][jt][jv]);
                V[jV] += pnl_pow_i(w_jn, jV); /* explosion! */
            }
            U[jV] = 0;
            for (jk = jV; jk >= 1; jk--)
            {
                U[jV] += (PNL_ALTERNATE(jk + 1) * V[jk] * U[jV - jk]);
            }
            U[jV] /= jV;
            if (p0 > 0.)
                nd[jt][jV] += p0 * U[jV] * cop->weights[jv];
        }
    }
}

data = fopen("cp-hw.dat", "w");
for (jn = 0; jn < cdo->n_comp; jn++)
{
    for (jt = 0; jt < t->size; jt++)
    {

```

```

        for (jv = 0; jv < cop->size; jv++)
        {
            fprintf(data, "%i %i %i %.15f\ n", jn, jt, jv, cp->p[jn][jt][jv]);
        }
    }
    fclose(data);
    free(U);
    free(V);

    return (nd);
}

```

```

double      **hw_numdef1(const CDO      *cdo,
                        const copula    *cop,
                        const grid      *t,
                        const cond_prob *cp)
{
    double      ***nd;
    double      **nd1;
    int         jw;
    double      *U;
    double      *V;
    double      p0;
    double      w_jn;
    int         jv;
    int         jV;
    int         jt;
    int         jn;
    int         jk;

    nd1 = malloc(t->size * sizeof(double *));
    nd = malloc(t->size * sizeof(double **));
    U = malloc((cdo->n_comp + 1) * sizeof(double));
    V = malloc((cdo->n_comp + 1) * sizeof(double));
    for (jt = 0; jt < t->size; jt++)
    {
        nd[jt] = malloc((cdo->n_comp + 1) * sizeof(double *));
        nd1[jt] = malloc((cdo->n_comp + 1) * sizeof(double));
        for (jV = 0; jV < (cdo->n_comp + 1); jV++)

```

```

{
    nd[jt][jV] = malloc((cop->size) * sizeof(double));
    nd1[jt][jV] = 0.;
}

for (jv = 0; jv < cop->size; jv++)
{
    for (jV = 0; jV < cdo->n_comp + 1; jV++)
    {
        nd[jt][jV][jv] = 0.0;
    }

    for (jw = 0; jw < cop->size; jw++)
    {
        p0 = 1.;
        for (jn = 0; jn < cdo->n_comp; jn++)
        {
            p0 = p0 * (1. - cp->p[jn][jt][jv + jw * cop->size]);
        }

        nd[jt][0][jv] += p0 * cop->weights[jw + cop->size];
        U[0] = 1.;
        for (jV = 1; jV < (cdo->n_comp + 1); jV++)
        {
            V[jV] = 0;
            for (jn = 0; jn < cdo->n_comp; jn++)
            {
                w_jn = cp->p[jn][jt][jv + jw * cop->size] / (1. - cp->p[jn][jt][jv + jw * cop->size]);
                V[jV] += pow(w_jn, jV);
            }
            U[jV] = 0;
            for (jk = 1; jk <= jV; jk++)
            {
                U[jV] += (PNL_ALTERNATE(jk + 1) * V[jk] * U[jV - jk]);
            }
            U[jV] = U[jV] / jV;
            nd[jt][jV][jv] += p0 * U[jV] * cop->weights[jw + cop->size];
        }
    }
}

for (jV = 1; jV < (cdo->n_comp + 1); jV++)
{

```

```

        nd1[jt][jV] = nd1[jt][jV] + nd[jt][jV][jv] * (cop->weights[jv]);
    }

    }
}
for (jt = 0; jt < t->size; jt++)
{
    for (jV = 0; jV < (cdo->n_comp + 1); jV++)
    {
        free(nd[jt][jV]);
    }
    free(nd[jt]);
}
free(nd);
free(U);
free(V);
/*    free(nd);*/
return (nd1);
}

```

```

double          **hw_losses_h(const CDO      *cdo,
                              const copula   *cop,
                              const grid     *t,
                              const grid     *x,
                              const cond_prob *cp)
{
    double      **cond_losses;
    double      **losses;
    double      *delta;
    double      p_default;
    double      sum;
    int         jt;
    int         jx;
    int         jv;
    int         jn;

    cond_losses = malloc(x->size * sizeof(double *));
    for (jx = 0; jx < x->size; jx++)
        cond_losses[jx] = malloc(cop->size * sizeof(double));
    delta = malloc(x->size * sizeof(double));
}

```

```

losses = malloc(t->size * sizeof(double *));
for (jt = 0; jt < t->size; jt++)
{
    for (jv = 0; jv < cop->size; jv++)
        cond_losses[0][jv] = 1.;
    for (jx = 1; jx < x->size; jx++)
    {
        for (jv = 0; jv < cop->size; jv++)
            cond_losses[jx][jv] = 0.;
    }
    for (jv = 0; jv < cop->size; jv++)
    {
        for (jn = 0; jn < cdo->n_comp; jn++)
        {
            p_default = cp->p[jn][jt][jv];
            sum = 0;
            for (jx = 1; jx < x->size; jx++)
            {
                delta[jx] = p_default * (cond_losses[jx - 1][jv]
                                          - cond_losses[jx][jv]);
                sum += delta[jx];
            }
            cond_losses[0][jv] -= sum;
            for (jx = 1; jx < x->size; jx++)
            {
                cond_losses[jx][jv] += delta[jx];
            }
        }
    }
    losses[jt] = malloc(x->size * sizeof(double));
    for (jx = 0; jx < x->size; jx++)
    {
        losses[jt][jx] = 0;
        for (jv = 0; jv < cop->size; jv++)
        {
            losses[jt][jx] += cond_losses[jx][jv] * cop->weights[jv];
        }
    }
}
for (jx = 0; jx < x->size; jx++)
    free(cond_losses[jx]);

```

```

    free(cond_losses);
    free(delta);

    return (losses);
}

```

```

double          **hw_losses_h1(const CDO      *cdo,
                                const copula   *cop,
                                const grid     *t,
                                const grid     *x,
                                const cond_prob *cp)
{
    double      ***cond_losses;
    double      **losses;
    double      ***losses1;
    double      *delta;
    double      p_default;
    double      sum;
    int         jt;
    int         jw;
    int         jx;
    int         jv;
    int         jn;

    cond_losses = malloc(x->size * sizeof(double **));
    for (jx = 0; jx < x->size; jx++)
        cond_losses[jx] = malloc(cop->size * sizeof(double *));
    delta = malloc(x->size * sizeof(double));
    losses = malloc(t->size * sizeof(double *));
    losses1 = malloc(t->size * sizeof(double **));

    for (jx = 0; jx < x->size; jx++)
    {
        for (jv = 0; jv < cop->size; jv++)
        {
            cond_losses[jx][jv] = malloc(cop->size * sizeof(double));
        }
    }
    for (jt = 0; jt < t->size; jt++)
    {

```

```

    losses1[jt] = malloc(x->size * sizeof(double *));
    losses[jt] = malloc(x->size * sizeof(double));
    for (jx = 0; jx < x->size; jx++)
    {
        losses1[jt][jx] = malloc(cop->size * sizeof(double));
        losses[jt][jx] = 0;
    }
}

for (jt = 0; jt < t->size; jt++)
{
    for (jx = 0; jx < x->size; jx++)
    {
        for (jv = 0; jv < cop->size; jv++)
        {
            losses1[jt][jx][jv] = 0.0;
        }
    }
}

for (jt = 0; jt < t->size; jt++)
{
    for (jv = 0; jv < cop->size; jv++)
    {
        for (jw = 0; jw < cop->size; jw++)
        {
            cond_losses[0][jv][jw] = 1.;
        }
        for (jx = 1; jx < x->size; jx++)
        {
            for (jw = 0; jw < cop->size; jw++)
            {
                cond_losses[jx][jv][jw] = 0.;
            }
        }

        for (jw = 0; jw < cop->size; jw++)
        {

```



```

    for (jn = 0; jn < cdo->n_comp; jn++)
    {
        p_default = cp->p[jn][jt][jv + jw * cop->size];
        sum = 0;

        for (jx = 1; jx < x->size; jx++)
        {
            delta[jx] = p_default * (cond_losses[jx - 1][jv][jw] - cond_losses[jx][jv][jw]);
            sum = sum + delta[jx];
        }

        cond_losses[0][jv][jw] = cond_losses[0][jv][jw] - sum;

        for (jx = 1; jx < x->size; jx++)
        {
            cond_losses[jx][jv][jw] = cond_losses[jx][jv][jw] + delta[jx];
        }
    }

    for (jx = 0; jx < x->size; jx++)
    {
        for (jw = 0; jw < cop->size; jw++)
        {
            losses1[jt][jx][jv] = losses1[jt][jx][jv] + cond_losses[jx][jv][jw];
        }
        losses[jt][jx] = losses[jt][jx] + losses1[jt][jx][jv] * cop->weight[jt][jx];
    }
}

/** Free **/
for (jx = 0; jx < x->size; jx++)
{
    for (jv = 0; jv < cop->size; jv++)
    {
        free(cond_losses[jx][jv]);
    }
    free(cond_losses[jx]);
}

```

```

for (jt = 0; jt < t->size; jt++)
{
    for (jx = 0; jx < x->size; jx++)
    {
        free(losses1[jt][jx]);
    }
    free(losses1[jt]);
}
free(cond_losses);
free(losses1);
free(delta);
return (losses);
}

```

```

double          **hw_losses_nh(const CDO      *cdo,
                                const copula   *cop,
                                const grid     *t,
                                const grid     *x,
                                const cond_prob *cp)
{
    double      **cond_losses;
    double      **mean_cond_losses;
    double      **losses;
    double      *add_cond;
    double      *add_mean;
    double      L_j;
    double      A_kpL_j;
    double      p_default;
    int          jt;
    int          jx;
    int          ujx;
    int          jv;
    int          jn;

    cond_losses = malloc(x->size * sizeof(double *));
    mean_cond_losses = malloc(x->size * sizeof(double *));
    for (jx = 0; jx < x->size; jx++)

```

```

{
    cond_losses[jx] = malloc(cop->size * sizeof(double));
    mean_cond_losses[jx] = malloc(cop->size * sizeof(double));
}
losses = malloc(t->size * sizeof(double *));
add_cond = malloc(x->size * sizeof(double));
add_mean = malloc(x->size * sizeof(double));
for (jt = 0; jt < t->size; jt++)
{
    for (jv = 0; jv < cop->size; jv++)
    {
        cond_losses[0][jv] = 1.;
        mean_cond_losses[0][jv] = 0.;
    }
    for (jx = 1; jx < x->size; jx++)
    {
        for (jv = 0; jv < cop->size; jv++)
        {
            cond_losses[jx][jv] = 0.;
            mean_cond_losses[jx][jv] = 0.;
        }
    }
    for (jn = 0; jn < cdo->n_comp; jn++)
    {
        L_j = cdo->C[jn]->nominal * (1 - RECOVERY(jn));
        for (jv = 0; jv < cop->size; jv++)
        {
            p_default = cp->p[jn][jt][jv];
            for (jx = 0; jx < x->size; jx++)
            {
                add_cond[jx] = 0.;
                add_mean[jx] = 0.;
            }
            for (jx = 0; jx < x->size; jx++)
            {
                A_kpL_j = mean_cond_losses[jx][jv] + L_j;
                ujx = jx;
                while ((ujx + 1 < x->size) && (A_kpL_j >= x->data[ujx + 1])) u
                if (ujx > jx)
                {
                    add_cond[jx] -= cond_losses[jx][jv] * p_default;

```

```

        add_cond[ujx] += cond_losses[jx][jv] * p_default;
        if (cond_losses[ujx][jv] + cond_losses[jx][jv] * p_default
            add_mean[ujx] = 0;
        else
            add_mean[ujx] += (cond_losses[jx][jv] * p_default * (A_k
        }
    else
    {
        add_mean[jx] += p_default * L_j;
    }
}
for (jx = 0; jx < x->size; jx++)
{
    cond_losses[jx][jv] += add_cond[jx];
    mean_cond_losses[jx][jv] += add_mean[jx];
}
}
}
losses[jt] = malloc(x->size * sizeof(double));
for (jx = 0; jx < x->size; jx++)
{
    losses[jt][jx] = 0;
    for (jv = 0; jv < cop->size; jv++)
    {
        losses[jt][jx] += cond_losses[jx][jv] * cop->weights[jv];
    }
}
}
free(add_cond);
free(add_mean);

return (losses);
}

```

```

double      **hw_losses_nh1(const CDO      *cdo,
                           const copula    *cop,
                           const grid      *t,
                           const grid      *x,
                           const cond_prob *cp)
{

```

```

double      ***cond_losses;
double      ***mean_cond_losses;
double      **losses;
double      *add_cond;
double      ***losses1;
double      *add_mean;
double      L_j;
double      A_kpL_j;
double      p_default;
int          jt;
int          jx;
int          ujx;
int          jv;
int          jn;
int          jw;
losses1 = malloc(t->size * sizeof(double **));
cond_losses = malloc(x->size * sizeof(double **));
mean_cond_losses = malloc(x->size * sizeof(double **));

for (jt = 0; jt < t->size; jt++)
{
    losses1[jt] = malloc(x->size * sizeof(double **));
}
for (jt = 0; jt < t->size; jt++)
{
    for (jx = 0; jx < x->size; jx++)
    {
        losses1[jt][jx] = malloc(cop->size * sizeof(double));
    }
}

for (jx = 0; jx < x->size; jx++)
{
    cond_losses[jx] = malloc(cop->size * sizeof(double **));
    mean_cond_losses[jx] = malloc(cop->size * sizeof(double **));
}

for (jx = 0; jx < x->size; jx++)
{
    for (jv = 0; jv < cop->size; jv++)

```

```

        {
            cond_losses[jx][jv] = malloc(cop->size * sizeof(double));
            mean_cond_losses[jx][jv] = malloc(cop->size * sizeof(double));
        }
    }
    losses = malloc(t->size * sizeof(double *));
    add_cond = malloc(x->size * sizeof(double));
    add_mean = malloc(x->size * sizeof(double));
    for (jt = 0; jt < t->size; jt++)
    {
        for (jv = 0; jv < cop->size; jv++)
        {
            for (jw = 0; jw < cop->size; jw++)
            {
                cond_losses[0][jv][jw] = 1.;
                mean_cond_losses[0][jv][jw] = 0.;
            }
        }

        for (jx = 1; jx < x->size; jx++)
        {
            for (jv = 0; jv < cop->size; jv++)
            {
                for (jw = 0; jw < cop->size; jw++)
                {
                    cond_losses[jx][jv][jw] = 0.;
                    mean_cond_losses[jx][jv][jw] = 0.;
                }
            }
        }

        for (jn = 0; jn < cdo->n_comp; jn++)
        {
            L_j = cdo->C[jn]->nominal * (1 - RECOVERY(jn));
            for (jv = 0; jv < cop->size; jv++)
            {
                for (jw = 0; jw < cop->size; jw++)
                {
                    p_default = cp->p[jn][jt][jv + jw * cop->size];
                    for (jx = 0; jx < x->size; jx++)
                    {
                        add_cond[jx] = 0.;

```

```

        add_mean[jx] = 0.;
    }
    for (jx = 0; jx < x->size; jx++)
    {
        A_kpL_j = mean_cond_losses[jx][jv][jw] + L_j;
        ujsx = jx;
        while ((ujsx + 1 < x->size) && (A_kpL_j >= x->data[ujsx + 1]))
        {
            if (ujsx > jx)
            {
                add_cond[jx] -= cond_losses[jx][jv][jw] * p_default;
                add_cond[ujsx] += cond_losses[jx][jv][jw] * p_default;
                if (cond_losses[ujsx][jv][jw] + cond_losses[jx][jv][jw] > 0)
                    add_mean[ujsx] = 0;
            }
            else
                add_mean[ujsx] += (cond_losses[jx][jv][jw] * p_default);
        }
        else
        {
            add_mean[jx] += p_default * L_j;
        }
    }
    for (jx = 0; jx < x->size; jx++)
    {
        cond_losses[jx][jv][jw] += add_cond[jx];
        mean_cond_losses[jx][jv][jw] += add_mean[jx];
    }
}

}

losses[jt] = malloc(x->size * sizeof(double));

for (jx = 0; jx < x->size; jx++)
{
    for (jv = 0; jv < cop->size; jv++)
    {
        losses1[jt][jx][jv] = 0;

        for (jw = 0; jw < cop->size; jw++)
        {
            losses1[jt][jx][jv] = losses1[jt][jx][jv] + cond_losses[jx][jv][jw];
        }
    }
}

```

```

    }
}
for (jx = 0; jx < x->size; jx++)
{
    losses[jt][jx] = 0;
    for (jv = 0; jv < cop->size; jv++)
    {
        losses[jt][jx] = losses[jt][jx] + losses1[jt][jx][jv] * cop->weigh
    }
}
}
free(add_cond);
free(add_mean);

return (losses);
}

```

```

double          **hw_losses_nh2(const CDO      *cdo,
                                const copula    *cop,
                                const grid      *t,
                                const grid      *x,
                                const cond_prob *cp)
{
    double        **cond_losses;
    double        **mean_cond_losses;
    double        **losses;
    double        *add_cond;
    double        *add_mean;
    double        L_j;
    double        A_kpL_j;
    double        p_default;
    int           jt;
    int           jx;
    int           ujx;
    int           jv;
    int           jn;

```



```

cond_losses = malloc(x->size * sizeof(double *));
mean_cond_losses = malloc(x->size * sizeof(double *));
for (jx = 0; jx < x->size; jx++)
{
    cond_losses[jx] = malloc(cop->size * sizeof(double));
    mean_cond_losses[jx] = malloc(cop->size * sizeof(double));
}
losses = malloc(t->size * sizeof(double *));
add_cond = malloc(x->size * sizeof(double));
add_mean = malloc(x->size * sizeof(double));
for (jt = 0; jt < t->size; jt++)
{
    for (jv = 0; jv < cop->size; jv++)
    {
        cond_losses[0][jv] = 1.;
        mean_cond_losses[0][jv] = 0.;
    }
    for (jx = 1; jx < x->size; jx++)
    {
        for (jv = 0; jv < cop->size; jv++)
        {
            cond_losses[jx][jv] = 0.;
            mean_cond_losses[jx][jv] = 0.;
        }
    }
    for (jn = 0; jn < cdo->n_comp; jn++)
    {
        L_j = cdo->C[jn]->nominal * (1 - RECOVERY(jn));
        for (jv = 0; jv < cop->size; jv++)
        {
            p_default = cp->p[jn][jt][jv];
            for (jx = 0; jx < x->size; jx++)
            {
                add_cond[jx] = 0.;
                add_mean[jx] = 0.;
            }
            for (jx = 0; jx < x->size; jx++)
            {
                A_kpL_j = mean_cond_losses[jx][jv] + L_j;
                u_jx = jx;
                while ((u_jx + 1 < x->size) && (A_kpL_j >= x->data[u_jx + 1])) u

```

```

        add_cond[jx] -= cond_losses[jx][jv] * p_default;
        add_cond[ujx] += cond_losses[jx][jv] * p_default;
        add_mean[ujx] += add_mean[ujx] + (mean_cond_losses[jx][jv] + L
    }
    for (jx = 0; jx < x->size; jx++)
    {
        cond_losses[jx][jv] += add_cond[jx];
        mean_cond_losses[jx][jv] += add_mean[jx];
    }
}
}
losses[jt] = malloc(x->size * sizeof(double));
for (jx = 0; jx < x->size; jx++)
{
    losses[jt][jx] = 0;
    for (jv = 0; jv < cop->size; jv++)
    {
        losses[jt][jx] += cond_losses[jx][jv] * cop->weights[jv];
    }
}
}
free(add_cond);
free(add_mean);

return (losses);
}

```

```

int CALC(HullWhite)(void *Opt, void *Mod, PricingMethod *Met)
{
    PnlVect          *nominal, *intensity, *dates, *x_rates, *y_rates;
    int              n_dates, n_rates, n_tranches, t_method, is_homo;
    int              t_copula, t_recovery;
    PremiaEnumMember *e;
    double           *p_copula, *p_recovery;

    int *p_method;
    TYPEOPT *ptOpt = (TYPEOPT *)Opt;
    TYPEMOD *ptMod = (TYPEMOD *)Mod;

    premia_interf_price_cdo(ptOpt, ptMod, Met,

```

```

        &nominal, &intensity,
        &n_rates, &x_rates, &y_rates,
        &n_dates, &dates, &n_tranches,
        &p_method, &is_homo);

t_method = (is_homo ? T_METHOD_HULL_WHITE_HOMO : T_METHOD_HULL_WHITE);
/*
 * Clayton copula not treated because the recursive approach of Hull and
 * White bursts out
 */
if (ptMod->t_copula.Val.V_ENUM.value == T_COPULA_CLAYTON)
    return PREMIA_UNTREATED_COPULA;

t_copula = (ptMod->t_copula.Val.V_ENUM.value);
e = lookup_premia_enum(&(ptMod->t_copula), t_copula);
p_copula = e->Par[0].Val.V_PNLVECT->array;
t_recovery = (ptMod->t_recovery.Val.V_ENUM.value);
p_recovery = get_t_recovery_arg(&(ptMod->t_recovery));

price_cdo(&(ptMod->Ncomp.Val.V_PINT),
          nominal->array,
          n_dates,
          dates->array,
          n_tranches + 1, /* size of the next array */
          ptOpt->tranch.Val.V_PNLVECT->array,
          intensity->array,
          n_rates,
          x_rates->array,
          y_rates->array,
          &t_recovery, /*t_recovery*/
          p_recovery,
          &(ptMod->t_copula.Val.V_ENUM.value),
          p_copula,
          &t_method,
          p_method,
          Met->Res[0].Val.V_PNLVECT->array,
          Met->Res[1].Val.V_PNLVECT->array,
          Met->Res[2].Val.V_PNLVECT->array
);

```

```

    pnl_vect_free(&nominal);
    pnl_vect_free(&intensity);
    pnl_vect_free(&dates);
    pnl_vect_free(&x_rates);
    pnl_vect_free(&y_rates);
    free(p_method);
    p_method = NULL;

    return OK;
}

static int CHK_OPT(HullWhite)(void *Opt, void *Mod)
{
    Option *ptOpt = (Option *)Opt;
    if (strcmp(ptOpt->Name, "CDO") == 0) return OK;
    return WRONG;
}

#endif //PremiaCurrentVersion
static int MET(Init)(PricingMethod *Met, Option *Opt)
{
    TYPEOPT *ptOpt = (TYPEOPT *)Opt->TypeOpt;
    int n_tranch;
    if (Met->init == 0)
    {
        Met->init = 1;
        Met->Par[0].Val.V_INT = 4;
        n_tranch = ptOpt->tranch.Val.V_PNLVECT->size - 1;
        Met->Res[0].Val.V_PNLVECT = pnl_vect_create_from_double(n_tranch, 0.);
        Met->Res[1].Val.V_PNLVECT = pnl_vect_create_from_double(n_tranch, 0.);
        Met->Res[2].Val.V_PNLVECT = pnl_vect_create_from_double(n_tranch, 0.);
    }

    return OK;
}

PricingMethod MET(HullWhite) =
{
    "Hull_White",
    { {"N subdivisions", INT, {4}, ALLOW},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    }
}

```

```

    },
    CALC(HullWhite),
    { {"Price(bp)", PNLVECT, {100}, FORBID},
      {"D_leg", PNLVECT, {100}, FORBID},
      {"P_leg", PNLVECT, {100}, FORBID},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CHK_OPT(HullWhite),
    CHK_ok,
    MET(Init)
};

```