

[Help](#)

```
#include <stdlib.h>
#include <
href../../common/math/cdo/cdo_math_h_src.pdfmath.h>
#include "
href../../mod/copula/copula_stdndc/price_cdo_h_src.pdfcdo.h"

/**
 * positive part
 *
 * @param x : double
 * @param y : double
 * @return max(x-y, 0)
 */

static double      pp(double      x, double      y)
{
    return ((x > y) ? (x - y) : 0.);
}

CDO      *init_CDO(const int      n_comp,
                  company      **C,
                  const int      n_dates,
                  const double   *t,
                  const int      n_tranches,
                  const double   *tr)
{
    CDO      *cdo = malloc(sizeof(CDO));
    int      jt;

    cdo->n_comp = n_comp;
    cdo->C = C;
    cdo->dates = init_grid_cdo(n_dates, t);
    cdo->n_tranches = n_tranches;
    cdo->tr = malloc(n_tranches * sizeof(double));
    for (jt = 0; jt < n_tranches; jt++)
        cdo->tr[jt] = tr[jt];

    return (cdo);
}
```

```

}

CDO      *homogenize_CDO(const CDO      *cdo)
{
    CDO      *hcdo = malloc(sizeof(CDO));
    company  **hCo;
    double    nominal;
    double    delta;
    int       jc, jt;

    hcdo->n_comp = cdo->n_comp;
    hcdo->dates = init_fine_grid(cdo->dates, 1);
    hcdo->n_tranches = cdo->n_tranches;
    hcdo->tr = malloc(cdo->n_tranches * sizeof(double));
    for (jt = 0; jt < cdo->n_tranches; jt++)
        hcdo->tr[jt] = cdo->tr[jt];
    hCo = malloc(cdo->n_comp * sizeof(company *));
    nominal = 0;
    delta = 0;
    for (jc = 0; jc < cdo->n_comp; jc++)
    {
        nominal += cdo->C[jc]->nominal;
        delta += cdo->C[jc]->mean_delta;
    }
    nominal /= (double) cdo->n_comp;
    delta /= (double) cdo->n_comp;
    for (jc = 0; jc < cdo->n_comp; jc++)
    {
        hCo[jc] = homogenize_company(cdo->C[jc], nominal, delta);
    }
    hcdo->C = hCo;

    return (hcdo);
}

void      free_cdo(CDO      **cdo)
{
    int     jn;

    free_grid((*cdo)->dates);

```

```

for (jn = 0; jn < (*cdo)->n_comp; jn++)
{
    free_company(((cdo)->C)[jn]);
}
free((cdo)->C);
(cdo)->C = NULL;
free((cdo)->tr);
(cdo)->tr = NULL;
free(cdo);
cdo = NULL;

return;
}

cond_prob      *init_cond_prob(const CDO      *cdo,
                                const copula    *cop,
                                const grid      *t)
{
    cond_prob      *cp;
    double          f_jt_jn;
    int             jn;
    int             jt;

    cp = malloc(sizeof(cond_prob));
    cp->p = malloc(cdo->n_comp * sizeof(double **));
    for (jn = 0; jn < cdo->n_comp; jn++)
    {
        cp->p[jn] = malloc(t->size * sizeof(double *));
        for (jt = 0; jt < t->size; jt++)
        {
            f_jt_jn = 1.0 - exp(- compute_sf(cdo->C[jn]->H, t->data[jt]));
            cp->p[jn][jt] = cop->compute_cond_prob(cop, f_jt_jn);
        }
    }
    cp->n_comp = cdo->n_comp;
    cp->n_t = t->size;

    return (cp);
}

void            free_cond_prob(cond_prob      *cp)

```

```

{
    int          jn;
    int          jt;

    for (jn = 0; jn < cp->n_comp; jn++)
    {
        for (jt = 0; jt < cp->n_t; jt++)
            free(cp->p[jn][jt]);
        free(cp->p[jn]);
    }
    free(cp->p);
    free(cp);

    return;
}

grid          **mean_losses(const CDO      *cdo,
                           const grid      *t,
                           const grid      *x,
                           double *const *losses)
{
    grid          **ml;
    int           jt;
    int           jx;
    int           jtr;
    double        A;
    double        B;
    double        ml_previous;

    ml = malloc((cdo->n_tranches - 1) * sizeof(grid *));
    for (jtr = 0; jtr < cdo->n_tranches - 1; jtr++)
    {
        ml_previous = 0.;
        A = cdo->tr[jtr];
        B = cdo->tr[jtr + 1];
        ml[jtr] = create_grid(t->size);
        for (jt = 0; jt < t->size; jt++)
        {
            ml[jtr]->data[jt] = 0;
            for (jx = 0; jx < x->size; jx++)
                ml[jtr]->data[jt] += (pp(x->data[jx], A) - pp(x->data[jx], B)) * losses[jtr][jt];
        }
    }
}

```

```

        ml[jtr]->delta[jt] = ml[jtr]->data[jt] - ml_previous;
        ml_previous = ml[jtr]->data[jt];
    }
}

return (ml);
}

grid          **mean_losses_from_numdef(const CDO      *cdo,
    const grid      *t,
    double *const    *numdef)
{
    grid          **ml;
    int            jt;
    int            jr;
    int            jtr;
    double         A;
    double         B;
    double         m;
    double         ml_previous;

    ml = malloc((cdo->n_tranches - 1) * sizeof(grid *));
    for (jtr = 0; jtr < cdo->n_tranches - 1; jtr++)
    {
        ml_previous = 0.;
        A = cdo->tr[jtr];
        B = cdo->tr[jtr + 1];
        m = cdo->C[0]->nominal * (1. - cdo->C[0]->mean_delta);
        ml[jtr] = create_grid(t->size);
        for (jt = 0; jt < t->size; jt++)
        {
            ml[jtr]->data[jt] = 0;
            for (jr = 0; jr < (cdo->n_comp + 1); jr++)
                ml[jtr]->data[jt] += (pp(m * jr, A) - pp(m * jr, B)) * numdef[jt][jr];
            ml[jtr]->delta[jt] = ml[jtr]->data[jt] - ml_previous;
            ml_previous = ml[jtr]->data[jt];
        }
    }

    return (ml);
}

```

```

double          *payment_leg(const CDO      *cdo,
                             const step_fun *rates,
                             const grid     *t,
                             grid *const    *mean_losses)
{
    int          jt;
    int          jt_payment;
    double       t_jt;
    double       t_previous;
    double       *pl;
    int          jpl;
    double       tau;
    pl = malloc((cdo->n_tranches - 1) * sizeof(double));
    for (jpl = 0; jpl < cdo->n_tranches - 1; jpl++)
    {
        pl[jpl] = 0;
    }
    jt_payment = 0;
    t_previous = 0;
    for (jt = 0; jt < t->size; jt++)
    {
        t_jt = t->data[jt];

        tau = exp(- compute_sf(rates, t_jt - t->delta[jt] * 0.5));
        for (jpl = 0; jpl < cdo->n_tranches - 1; jpl++)
        {
            pl[jpl] += tau * mean_losses[jpl]->delta[jt] * (t_jt - t->delta[jt] *
        }
        tau = exp(- compute_sf(rates, t_jt));
        if (t_jt == cdo->dates->data[jt_payment])
        {
            for (jpl = 0; jpl < cdo->n_tranches - 1; jpl++)
            {
                pl[jpl] += tau * (cdo->tr[jpl + 1] - cdo->tr[jpl] - mean_losses[jp
            }
            t_previous = cdo->dates->data[jt_payment];
            jt_payment++;
        }
    }
}

```

```

    }

    return (p1);
}

double      *default_leg(const CDO      *cdo,
                        const step_fun *rates,
                        const grid      *t,
                        grid *const     *mean_losses)
{
    int      jt;
    double   *dl;
    int      jdl;
    double   tau;

    dl = malloc((cdo->n_tranches - 1) * sizeof(double));
    for (jdl = 0; jdl < cdo->n_tranches - 1; jdl++)
    {
        dl[jdl] = 0;
    }
    for (jt = 0; jt < t->size; jt++)
    {
        tau = exp(- compute_sf(rates, t->data[jt] - t->delta[jt] * 0.5));
        for (jdl = 0; jdl < cdo->n_tranches - 1; jdl++)
        {
            dl[jdl] += tau * mean_losses[jdl]->delta[jt];
        }
    }

    return (dl);
}

```