

[Help](#)

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

#include "
href../../../../mod/dynamic/dynamic_stdndc/dynamic_stdndc_h_src.pdfdynamic_stdndc.h
#include "pnl/pnl_matrix.h"
#include "pnl/pnl_cdf.h"
#include "pnl/pnl_random.h"
#include "pnl/pnl_mathtools.h"
#include "pnl/pnl_integration.h"
#include "pnl/pnl_root.h"
#include "pnl/pnl_specfun.h"

/*
 * July 2009.
 * Cédric Allali who worked under the supervision of Céline Labart.
 * Modified by Jérôme Lelong to use Pnl
 */

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2010+2) //The "#els
static int CHK_OPT(Hedging_CousinFermanianLaurent)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(Hedging_CousinFermanianLaurent)(void *Opt, void *Mod, PricingMethod *Me
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else

/*
 *
 *      Calcul des probabilités de défaut
 *      suivant le modèle à 1 facteur gaussien
 *
 */
```

```

typedef struct
{
    double rho;
    double T;
    double s;
    double R;
    long n;
    long k;
} Funct_Param;

typedef struct
{
    int k;
    int n;
    double T;
    PnlVect *DP; /* probabilités de défauts p(T,i) 0<=i<=n */
    PnlVect *lambda; /* intensités de pertes*/
    PnlVect *a; /* coefficients a_ki */
} cdo_params;

/*
 * creates a new cdo_params structure
 * The parameters n and T are fixed at the creation of the structure
 */
static cdo_params *new_cdo_params(int n, double T)
{
    cdo_params *p;

    p = malloc(sizeof(cdo_params));
    p->n = n;
    p->T = T;
    p->DP = pnl_vect_create(n + 1);
    p->lambda = pnl_vect_create(n + 1);
    p->a = pnl_vect_create(n + 1);
    return p;
}

/*
 * Deletes a cdo_params structure
 */

```

```

static void free_cdo_params(cdo_params **p)
{
    pnl_vect_free(&((*p)->DP));
    pnl_vect_free(&((*p)->lambd));
    pnl_vect_free(&((*p)->a));
    free(*p);
    *p = NULL;
}

static double nbdef(double p, double T, long n, long k, double s, double R, double b)
{
    /* On commence par calculer  $C_n^k \tilde{p}(x)^k (1 - \tilde{p}(x))^{n-k}$  puis on
       intégrera suivant x*/
    int which, status, i;
    double bound, x, mean, sd, rrho, roprho, a, b;
    double p_T = 1 - exp(-s / (1 - R) * T); /*p_T*/
    double q_T = 1 - p_T;
    double q = 1 - p;
    double res = pnl_sf_choose(n, k);
    rrho = sqrt(rho);
    roprho = sqrt(1 - rho);
    a = roprho / rrho;
    which = 2;
    mean = 0.;
    sd = 1.;
    /*calcul de  $b = \phi^{-1}(1 - \exp(-s/(1-R)T))$ */
    pnl_cdf_nor(&which, &p_T, &q_T, &b, &mean, &sd, &status, &bound);
    if (status != 0)
    {
        printf("Error in pnl_cdf_nor : \n");
        printf("\ tstatus=%d \n", status);
        abort();
    }
    pnl_cdf_nor(&which, &p, &q, &x, &mean, &sd, &status, &bound); /*calcul de  $x = \phi^{-1}(1 - \exp(-s/(1-R)T))$ */
    if (status != 0)
    {
        printf("Error in pnl_cdf_nor : \n");
        printf("\ tstatus=%d \ t bound=%f \n", status, bound);
        abort();
    }
    for (i = 1 ; i <= k ; i++) res *= p;
}

```

```

    for (i = k + 1 ; i <= n ; i++) res *= q;
    res *= a * exp(x * x / 2 * (1 - a * a) - b * b / (2 * rho) + x * b * a / rrho)
    return res;
}

static double nbdef_Apply(double x, void *D)
{
    double rho = ((Funct_Param *) D)->rho;
    double T   = ((Funct_Param *) D)->T;
    double s    = ((Funct_Param *) D)->s;
    double R    = ((Funct_Param *) D)->R;
    long n      = ((Funct_Param *) D)->n;
    long k      = ((Funct_Param *) D)->k;
    return nbdef(x, T, n, k, s, R, rho);
}

PnlVect *proba_nb_def(double T, long n, double s, double R, double rho)
{
    PnlVect      *proba;
    Funct_Param  P;
    PnlFunc      func;
    double       proba_k;
    double       epsres;
    int          k = 0;
    int          N = 10000000;
    proba = pnl_vect_create(n + 1);

    P.T = T;
    P.n = n;
    P.s = s;
    P.R = R;
    P.rho = rho;
    for (k = 0; k <= n; k++)
    {
        P.k = k;
        func.F = nbdef_Apply;
        func.params = &P;
        pnl_integration_GK(&func, 0.0, 1.0, 0.000000001, 0.0000000001, &proba_k, &
        pnl_vect_set(proba, k, proba_k);
    }
}

```

```

    return proba;
}

/*
 *
 * Calibration des intensités de perte
 *
 */

/* Définition de la fonction f_k(x) : on passe les lambda_k, a_ki et p(T,k) en
 * paramètre sous forme de tableau */
static void fdf_func(double x, double *f, double *df, int k,
                    double T, PnlVect *DP, PnlVect *lambda, PnlVect *a)
{
    int i = 0;
    double coeff = 0;
    double f_x = 0;
    double denof_x = 0;
    double exp_x = 0;
    double sum1 = 0;
    double sum2 = 0;
    for (i = 0; i <= k - 1; i++)
    {
        exp_x = exp(-(x - GET(lambda, i)) * T);
        coeff = GET(a, i) * exp(-GET(lambda, i) * T);
        denof_x = 1 / (x - GET(lambda, i));
        f_x = (1 - exp_x) * denof_x;
        sum1 += coeff * f_x;
        sum2 += coeff * (exp_x * (T * (x - GET(lambda, i)) + 1) - 1) * denof_x * d
    }
    *f = sum1 - GET(DP, k) / GET(lambda, k - 1);
    *df = sum2;
}

/*Redéfition de f_k(x) pour pouvoir exploiter les solveurs de la pnl */
static void Param_Func_Apply(double x, double *f, double *df, void *p)
{

```

```

int      k      = ((cdo_params *) p) -> k;
double   T      = ((cdo_params *) p) -> T;
PnlVect *DP     = ((cdo_params *) p) -> DP;
PnlVect *lambda = ((cdo_params *) p)-> lambda;
PnlVect *a      = ((cdo_params *) p) -> a;
fdf_func(x, f, df, k, T, DP, lambda, a);
}

```

```

static PnlVect *calib(double T, long n, PnlVect *prob, double epsabs, double eps
{
    double      x0, r, temp;
    cdo_params  *D;
    PnlFuncDFunc func;
    PnlVect     *lambda;
    int         i, j;

    D = new_cdo_params(n, T);
    LET(D->a, 0) = 1;
    pnl_vect_clone(D->DP, prob);

    LET(D->lambda, 0) = -log(GET(prob, 0)) / T;
    for (i = 1; i <= 35; i++)
    {
        temp = 0;
        D->k = i;
        x0 = GET(D->lambda, i - 1) + 0.1;
        func.F = Param_Func_Apply;
        func.params = D;
        pnl_root_newton(&func, x0, epsrel, epsabs, N_max, &r);
        LET(D->lambda, i) = r;
        /*mise à jour des coefficients a_ki*/
        for (j = 0; j <= i - 1; j++)
        {
            LET(D->a, j) *= GET(D->lambda, i - 1) / (GET(D->lambda, i) - GET(D->la
            temp += GET(D->a, j);
        }
        LET(D->a, i) = -temp; /* a_ii = -sum_j a_ij */
    }
    for (i = 36; i <= n; i++)
    {

```

```

        LET(D->lambda, i) = 2 * GET(D->lambda, i - 1) - GET(D->lambda, i - 2);
    }
    lambda = pnl_vect_copy(D->lambda);
    free_cdo_params(&D);
    return lambda;
}

/*
 *
 *   Calcul des deltas par une méthode
 *   d'arbre recombinaant
 *
 */

/* Calcul du nominal en cas de k défaut pour la tranche [a,b] d'un CDO */
static double outnom(int k, double R, int n, double a, double b)
{
    double L = (1 - R) * k / n;
    if (L < a) return (b - a);
    else if (L >= b) return 0;
    else return (b - L);
}

/* Calcul de la jambe de défaut */

static PnlMat *default_leg(double a, double b, PnlVect *lambda,
                           double r, double R, int n, double delta, double T)
{
    int      i, k, min;
    double   exp_k, curr, diff;
    long int Ns = (long int) floor(T / delta);
    PnlMat *dl = pnl_mat_create_from_double(Ns + 1, n + 1, 0.0);
    for (i = Ns - 1; i >= 0; i--)
    {
        min = (i >= (n - 1)) ? (n - 1) : i;
        for (k = min; k >= 0; k--)
        {
            exp_k = exp(-GET(lambda, k) * delta);
            diff = outnom(k, R, n, a, b) - outnom(k + 1, R, n, a, b);
            curr = exp(-r * delta) * ((1 - exp_k) * (pnl_mat_get(dl, i + 1, k + 1)

```

```

        pnl_mat_set(dl, i, k, curr);
    }
}

return dl;
}

/*
 * Fonction qui retourne l'entier l tel que tab[l] < x <= tab[l+1]
 */
static int is_out(double x, PnlVect *tab, int *out)
{
    int l = 0;
    int length = tab->size;
    while ((x > GET(tab, l)) && (l < length))
        l++;
    if (l <= length)
    {
        if (LET(tab, l) == x) *out = 0;
        else *out = 1;
        return l - 1;
    }
    else
    {
        if (LET(tab, l - 1) == x) *out = 0;
        else *out = 1;
        return l - 1;
    }
}

static PnlMat *premium_leg(double a, double b, PnlVect *lambda,
                           double r, double R, int n, double delta,
                           double T, PnlVect *PT)
{
    int i, k, min, l;
    double exp_k, curr, diff;
    long int Ns = (long int) floor(T / delta);
    int out = 1;
    PnlMat *pl = pnl_mat_create_from_double(Ns + 1, n + 1, 0.0);
    curr = 0;
    diff = 0;

```

```

for (i = Ns - 1; i >= 0; i--)
{
    min = (i >= (n - 1)) ? (n - 1) : i;
    for (k = min; k >= 0; k--)
    {
        exp_k = exp(-GET(lambda, k) * delta);
        l = is_out((i + 1) * delta, PT, &out);
        if (out)
        {
            diff = outnom(k, R, n, a, b) - outnom(k + 1, R, n, a, b);
            curr = exp(-r * delta) * ((1 - exp_k) * (pnl_mat_get(pl, i + 1, k +
                diff * ((i + 1) * delta - GET(PT, l)))
                + exp_k * pnl_mat_get(pl, i + 1, k));
            pnl_mat_set(pl, i, k, curr);
        }
        else
        {
            curr = exp(-r * delta) * (outnom(k, R, n, a, b) * (GET(PT, l + 1)
                (1 - exp_k) * pnl_mat_get(pl, i + 1, k +
                + exp_k * pnl_mat_get(pl, i + 1, k));
            pnl_mat_set(pl, i, k, curr);
        }
    }
}
return pl;
}

```

```

/* Calcul de la matrice V_CDO(i,k)=d(i,k)-sr(i,k)*/
/* On récupère le spread par passage par référence*/
static PnlMat *V_CDO(double a, double b, PnlVect *lambda, double r,
    double R, int n, double delta, double T,
    PnlVect *PT, double *s)
{
    int i, k;
    long int Ns = (long int) floor(T / delta);
    PnlMat *V = pnl_mat_create(Ns + 1, n + 1);
    PnlMat *dl = default_leg(a, b, lambda, r, R, n, delta, T);
    PnlMat *pl = premium_leg(a, b, lambda, r, R, n, delta, T, PT);
    *s = pnl_mat_get(dl, 0, 0) / pnl_mat_get(pl, 0, 0);
    if ((a == 0) && (b == 0.03))

```

```

    {
        *s = 0.05; /*spread à 500bps pour la tranche equity*/
    }
    for (i = 0; i <= Ns; i++)
    {
        for (k = 0; k <= n; k++)
        {
            pnl_mat_set(V, i, k, pnl_mat_get(dl, i, k) - (*s) * pnl_mat_get(pl, i, k));
        }
    }
    pnl_mat_free(&dl);
    pnl_mat_free(&pl);
    return V;
}

/* Calcul de la matrice V-IS */

static PnlMat *premium_leg_IS(PnlVect *lambda, double r, double R, int n,
                              double delta, double T, PnlVect *PT)
{
    /* Même relation de récurrence que pour un CDO mais le nominal est différent */
    int i, k, min, l;
    int out;
    double exp_k, curr;
    long int Ns = (long int) floor(T / delta);
    PnlMat *pl = pnl_mat_create_from_double(Ns + 1, n + 1, 0);
    for (i = Ns - 1; i >= 0; i--)
    {
        min = (i >= (n - 1)) ? (n - 1) : i;
        for (k = min; k >= 0; k--)
        {
            exp_k = exp(-GET(lambda, k) * delta);
            l = is_out((i + 1) * delta, PT, &out);
            if (out)
            {
                curr = exp(-r * delta) * ((1 - exp_k) * (pnl_mat_get(pl, i + 1, k)
                                                            + 1. / n * ((i + 1) * delta - GET(PT, l)
                                                            + exp_k * pnl_mat_get(pl, i + 1, k)));
                pnl_mat_set(pl, i, k, curr);
            }
            else

```

```

        {
            curr = exp(-r * delta) * ((1 - k / ((float) n)) * (GET(PT, l + 1)
                + (1 - exp_k) * pnl_mat_get(pl, i + 1, k)
                + exp_k * pnl_mat_get(pl, i + 1, k)));
            pnl_mat_set(pl, i, k, curr);
        }
    }
    return pl;
}

```

```

/* On récupère le spread par passage par référence*/
static PnlMat *V_IS(PnlVect *lambda, double r, double R, int n,
    double delta, double T, PnlVect *PT, double *s)
{
    int i, k, min;
    double exp_k, curr, diff;
    long int Ns = (long int) floor(T / delta);
    PnlMat *V;
    PnlMat *dl;
    PnlMat *pl;
    V = pnl_mat_create(Ns + 1, n + 1);
    dl = pnl_mat_create_from_double(Ns + 1, n + 1, 0.0);
    for (i = Ns - 1; i >= 0; i--)
    {
        min = (i >= (n - 1)) ? (n - 1) : i;
        for (k = min; k >= 0; k--)
        {
            exp_k = exp(-GET(lambda, k) * delta);
            diff = (1 - R) / n;
            curr = exp(-r * delta) * ((1 - exp_k) * (pnl_mat_get(dl, i + 1, k + 1)
                + exp_k * pnl_mat_get(dl, i + 1, k)));
            pnl_mat_set(dl, i, k, curr);
        }
    }
    pl = premium_leg_IS(lambda, r, R, n, delta, T, PT);
    *s = pnl_mat_get(dl, 0, 0) / pnl_mat_get(pl, 0, 0);
    for (i = 0; i <= Ns; i++)
    {
        for (k = 0; k <= n; k++)

```

```

        {
            pnl_mat_set(V, i, k, pnl_mat_get(dl, i, k) - (*s)*pnl_mat_get(pl, i, k)
        }
    }
    pnl_mat_free(&dl);
    pnl_mat_free(&pl);
    return V;
}

```

```

static PnlMat *DeltaCDO(double a, double b, double r, double R,
                        int n, double delta, double T, PnlVect *PT, PnlVect *lam)
{
    int i, k, l, out;
    PnlMat *Vcdo, *Vis, *mat_delta;
    double s, s_IS, num, denum, diff;
    int Ns = (int) floor(T / delta);
    mat_delta = pnl_mat_create(Ns, n);
    Vcdo = V_CDO(a, b, lambda, r, R, n, delta, T, PT, &s);
    Vis = V_IS(lambda, r, R, n, delta, T, PT, &s_IS);
    for (i = 0; i < Ns; i++)
        for (k = 0; k < n; k++)
        {
            l = is_out((i + 1) * delta, PT, &out);
            diff = outnom(k, R, n, a, b) - outnom(k + 1, R, n, a, b);
            num = pnl_mat_get(Vcdo, i + 1, k + 1) - pnl_mat_get(Vcdo, i + 1, k) + diff;
            denum = pnl_mat_get(Vis, i + 1, k + 1) - pnl_mat_get(Vis, i + 1, k) + (1 - s_IS);
            pnl_mat_set(mat_delta, i, k, num / denum);
        }
    pnl_mat_free(&Vcdo);
    pnl_mat_free(&Vis);
    return mat_delta;
}

```

```

static void Hedging_CFL(PnlVect *Delta, const PnlVect *tranch, double R, int n,
                        double rho, int n_coupons, double T, double r,
                        double t, int n_defaults)
{
    double s;
    PnlVect *PT;
    PnlVect *prob;
}

```

```

PnlVect *lambda;
PnlMat *CH;
int      i;
long     k = 0;
s = 0.002; /* initial value of the spread, see the article */

prob = proba_nb_def(T, n, s, R, rho);
lambda = calib(T, n, prob, 0.0001, 0.00001, 1000);
PT = pnl_vect_create((int)(T * n_coupons) + 1);
for (k = 0 ; k < (int)(T * n_coupons) + 1 ; k++)
{
    LET(PT, k) = ((double) k) / n_coupons;
}
for (i = 0 ; i < tranch->size - 1 ; i++)
{
    CH = DeltaCDO(pnl_vect_get(tranch, i), pnl_vect_get(tranch, i + 1),
                  r, R, n, 1. / 365, T, PT, lambda);
    pnl_vect_set(Delta, i, 10000 * pnl_mat_get(CH, (int)(t * 365), n_defaults)
    pnl_mat_free(&CH);
}

pnl_vect_free(&PT);
pnl_vect_free(&prob);
pnl_vect_free(&lambda);
}

int CALC(Hedging_CousinFermanianLaurent)(void *Opt, void *Mod, PricingMethod *Me
{
    int      n, n_coupons;
    double   R, T, r, t, n_defaults;
    double   rho;
    PnlVect *tranch;
    TYPEOPT *ptOpt = (TYPEOPT *)Opt;
    TYPEMOD *ptMod = (TYPEMOD *)Mod;

    tranch = ptOpt->tranch.Val.V_PNLVECT;
    n = ptMod->Ncomp.Val.V_PINT;
    r = ptMod->r.Val.V_DOUBLE;

```

```

    T = ptOpt->maturity.Val.V_DATE;
    t = ptOpt->date.Val.V_DATE;
    n_defaults = ptOpt->n_defaults.Val.V_INT;
    R = ptMod->Recovery.Val.V_PDOUBLE;
    n_coupons = ptOpt->NbPayment.Val.V_INT;

    rho = Met->Par[0].Val.V_DOUBLE;
    Hedging_CFL(Met->Res[0].Val.V_PNLVECT, tranch, R, n, rho, n_coupons, T, r, t,
    return OK;
}

```

```

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

```

```

#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->HelpFilenameHint = "LCF_cdo_hedging";
        Met->Par[0].Val.V_DOUBLE = 0.3;
        n_tranch = ptOpt->tranch.Val.V_PNLVECT->size - 1;

        Met->Res[0].Val.V_PNLVECT = pnl_vect_create_from_double(n_tranch, 0.);
    }
    return OK;
}

```

```

PricingMethod MET(Hedging_CousinFermanianLaurent) =
{
    "Hedging_CousinFermanianLaurent",
    { {"Rho", DOUBLE, {0}, FORBID},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    }
}

```

```

    },
    CALC(Hedging_CousinFermanianLaurent),
    { {"Delta(bp)", PNLVECT, {100}, FORBID},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CHK_OPT(Hedging_CousinFermanianLaurent),
    CHK_ok,
    MET(Init)
};

```