

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_integration.h"

/*****
 * This code was written by Ahmed Kebaier using the GSL and was slightly modified
 * by Jérôme Lelong to use the PNL instead.
 *****/

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2009+2) //The "#else"
static int CHK_OPT(RogersDiGraziano)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(RogersDiGraziano)(void *Opt, void *Mod, PricingMethod *Met)
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else

static double A, B, L;

/*****
/* Parameters taken of the Markov chain taken in Di Graziano-Rogers pag.21 */
*****/

/* Initial law of the markov chain */
static double pi[4] = {0.0019, 0.0, 0.9981, 0.0};
```

```

/* Intensity of default depending on the Markov chain */
static double lambda[4] = {0.0545, 0.0134, 0.0000, 0.0007};
/*Infinitesimal generator of the Markov chain */
static double Qdata[16] = { -0.0069, 0.0000, 0.0004, 0.0065, 0.0179, -0.0180, 0.
                           0.0000, 0.0000, 0.0000, -0.4291, 0.4291, 0.0000, 1.2
                           };

/* Weights */
static double wdata[16] = {0., 9.3981, 0.12770, 14.8746, 0.0000, 0., 10.0362, 19
                           9.1688, 7.5897, 0., 0.00000, 6.4959, 0.0009, 0.74070,
                           };

/* Computation of the coefficient beta(t) */
static double betax(double x, double temps)
{
    double p;
    int i, cont;
    double *bxdata;
    PnlMat bx, *ebx;
    PnlVect *W;

    bxdata = malloc(16 * sizeof(double));
    cont = 0;
    for (i = 0; i < 16; i++)
    {
        if (i == 0)
            bxdata[0] = (Qdata[0] - x * lambda[0]) * temps;
        else if ((i % 5 == 0))
        {
            cont++;
            bxdata[i] = (Qdata[i] - x * lambda[cont]) * temps;
        }
        else
            bxdata[i] = Qdata[i] * temps * exp(-wdata[i] * x);
    }

    bx = pnl_mat_wrap_array(bxdata, 4, 4);
    ebx = pnl_mat_create(4, 4);
    pnl_mat_exp(ebx, &bx);

    W = pnl_vect_create(4);

```

```

    pnl_mat_sum_vect(W, ebx, 'c');
    p = 0.;
    for (i = 0; i < 4; i++)
        p += pnl_vect_get(W, i) * pi[i];

    free(bxdata);
    pnl_vect_free(&W);
    pnl_mat_free(&ebx);

    return p;
}

/*Integrand Function for the costant beta */
static double g(double x, void *params)

{
    double kappa = ((double *)params)[0];
    return betax(x, kappa);
}

static double beta(double temps)
{
    double Result, Error;
    int neval;
    PnlFunc G;

    G.F = &g;
    G.params = &temps;

    pnl_integration_GK(&G , 0., 1., 0, 1e-6, &Result, &Error, &neval);
    return Result;
}

/* Laplace transform of the payment and default leg functions */

static double Laplace_transform(double alpha, double t, int Nb_company, double L
{
    double laplace_value;
    double C;

```

```

int i, cont;
PnlVect *VV;
double *Qtildadata;
PnlMat Qtilda, *eQtilda;

Qtildadata = malloc(16 * sizeof(double));

/* Useful constant */
C = (1. - exp(-alpha * L)) * Nb_company * beta(t);

/* Qtildadata contains the data for the Transformation of the infinitesimal
   generator of the markov chain */
cont = 0;
for (i = 0; i < 16; i++)
{
    if (i == 0)
        Qtildadata[0] = (Qdata[0] - R - C * lambda[0]) * t;
    else if ((i % 5 == 0))
    {
        cont++;
        Qtildadata[i] = (Qdata[i] - R - C * lambda[cont]) * t;
    }
    else
        Qtildadata[i] = Qdata[i] * t * exp(-wdata[i] * C);
}

/* Qtilda : the transformation of the infinitesimal generator
   of the markov chain */
Qtilda = pnl_mat_wrap_array(Qtildadata, 4, 4);

/* eQtilda : matrix exponential of Qtilda */
eQtilda = pnl_mat_create(4, 4);
pnl_mat_exp(eQtilda, &Qtilda);

VV = pnl_vect_create(4);

/* Computation of laplace transform of the default and
   payment legs */
pnl_mat_sum_vect(VV, eQtilda, 'c');

```

```

    laplace_value = 0.;
    for (i = 0; i < 4; i++)
        laplace_value += (1.0 / (alpha * alpha)) * pnl_vect_get(VV, i) * pi[i];

    free(Qtildadata);
    pnl_vect_free(&VV);
    pnl_mat_free(&Qtilda);

    return laplace_value;
}

```

```

/* Inverse Laplace Transform for the computation of the
   default and payment legs */
static double InverseTransform(double laplace, double t, int Nb_company,
                               double L, double R)
{
    int k;
    int i;

    int N2 = 14 / 2;
    int NV = 2 * N2;
    double V[14]; //V[NV];
    int sign = 1;
    double ln2t;
    double x = 0.;
    double y = 0.;
    int kmin, kmax;

    if (N2 & 1) sign = -1;
    for (i = 0; i < NV; i++)
    {
        kmin = (i + 2) / 2;
        kmax = i + 1;
        if (kmax > N2)
            kmax = N2;
        V[i] = 0;
        sign = -sign;
        for (k = kmin; k <= kmax; k++)
        {
            V[i] = V[i] + (pow(k, N2) / pnl_fact(k)) *

```

```

        (pnl_fact(2 * k) / pnl_fact(2 * k - i - 1)) / pnl_fact(N2 - k)
        / pnl_fact(k - 1) / pnl_fact(i + 1 - k);
    }
    V[i] = sign * V[i];
}
if (laplace == 0) laplace = 1e-4;
ln2t = M_LN2 / laplace;

for (i = 0; i < NV; i++)
{
    x += ln2t;
    y += V[i] * Laplace_transform(x, t, Nb_company, L, R); /* f(x) */
}

return ln2t * y;
}

/* structure used to pass fixed parameters when integrating functions */
struct par
{
    double alpha;
    double r;
    int n;
};

/* Integrand Function in the default leg formula */
static double f(double x, void *params)
{
    double f, r;
    int Nb_company;

    r = ((struct par *) params)->r;
    Nb_company = ((struct par *) params)->n;

    f = (InverseTransform(B, x, Nb_company, L, r) - InverseTransform(A, x, Nb_comp

    return f;
}

```

```

/* Rogers-Di Graziano Algorithm */
static void rdg(double r, double maturity, int Nb_company, const PnlVect *tranches,
               double recovery, double frequency, PnlVect *prices, PnlVect *deltas)
{
    int Ndate;
    int i, k, neval;
    double Ti;
    double pl, dl;
    double error, result;
    struct par t;
    double alpha = 1.0;
    PnlFunc F;

    L = (1 - recovery) / (double)Nb_company;
    Ndate = (int)(maturity / frequency);

    F.F = &f;
    t.alpha = alpha;
    t.r = r;
    t.n = Nb_company;
    F.params = &t;

    for (k = 0; k < tranches->size - 1; k++)
    {
        /* Interval of tranches */
        A = pnl_vect_get(tranches, k);
        B = pnl_vect_get(tranches, k + 1);

        /*Integration in the default leg formula */

        pnl_integration_GK(&F, 0., maturity, 0, 1e-6, &result, &error, &neval);
        /* Compute Payment Leg */
        Ti = 0;
        pl = 0;
        for (i = 0; i < Ndate; i++)
        {
            Ti += frequency;
            pl += frequency * (InverseTransform(B, Ti, Nb_company, L, r) -
                               InverseTransform(A, Ti, Nb_company, L, r)) / (B - A);
        }
    }
}

```

```

    pnl_vect_set(pleg, k, pl);
    /* Compute Default Leg */
    dl = 1. - (InverseTransform(B, maturity, Nb_company, L, r) -
              InverseTransform(A, maturity, Nb_company, L, r)) / (B - A) - r;
    pnl_vect_set(dleg, k, dl);

    //Compute Price CDO
    pnl_vect_set(prices, k, 10000 * dl / pl);
}

}

int CALC(RogersDiGraziano)(void *Opt, void *Mod, PricingMethod *Met)
{
    TYPEOPT      *ptOpt      = (TYPEOPT *)Opt;
    TYPEMOD      *ptMod      = (TYPEMOD *)Mod;
    int          n_tranch = ptOpt->tranch.Val.V_PNLVECT->size - 1;
    int          n;
    double       recovery, r, frequency, maturity;

    /* initialize Results. Have been allocated in Init method */
    pnl_vect_resize(Met->Res[0].Val.V_PNLVECT, n_tranch);
    pnl_vect_resize(Met->Res[1].Val.V_PNLVECT, n_tranch);
    pnl_vect_resize(Met->Res[2].Val.V_PNLVECT, n_tranch);

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

    maturity = ptOpt->maturity.Val.V_DATE;
    recovery = ptMod->Recovery.Val.V_PDOUBLE;
    frequency = 1. / ptOpt->NbPayment.Val.V_INT;

    rdg(r, maturity, n, ptOpt->tranch.Val.V_PNLVECT, recovery, frequency,
        Met->Res[0].Val.V_PNLVECT, Met->Res[1].Val.V_PNLVECT, Met->Res[2].Val.V_PNLVECT);
    return OK;
}

static int CHK_OPT(RogersDiGraziano)(void *Opt, void *Mod)
{
    Option *ptOpt = (Option *)Opt;
    TYPEOPT *TypeOpt = (TYPEOPT *)ptOpt->TypeOpt;
    int      status = 0;

```



```

if (strcmp(ptOpt->Name, "CDO") != 0) return WRONG;
if (TypeOpt->t_nominal.Val.V_ENUM.value != 1)
{
    printf("Only homogeneous nominals are accepted\ n");
    status ++;
}

if (status) 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;
        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(RogersDiGraziano) =
{
    "RogersDiGraziano",
    {" ", PREMIA_NULLTYPE, {0}, FORBID}},
    CALC(RogersDiGraziano),
    { {"Price(bp)", PNLVECT, {100}, FORBID},
      {"D_leg", PNLVECT, {100}, FORBID},
      {"P_leg", PNLVECT, {100}, FORBID},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CHK_OPT(RogersDiGraziano),
    CHK_ok,

```

```
    MET(Init)  
};
```