

[Help](#)

```
/*
 * Problèmes :
 *   3. imposer une maturité plus petite que 10 ans
 */

#include "black_cox_extended_std.h"
#include "enums.h"

#include "pnl/pnl_complex.h"
#include "pnl/pnl_random.h"
#include "pnl/pnl_mathtools.h"
#include "pnl/pnl_vector.h"
#include "pnl/pnl_fft.h"
#include "pnl/pnl_laplace.h"

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

/*
 * Structure used to store the model parameters with multiple barriers
 */
typedef struct
{
    int n;
    const PnlVect *mu; /* Vector of the n default intensities */
    double m;
    double m_sq; /* m * m */
    const PnlVect *b; /* Vector of the n-1 barriers */
}
```

```

} ParasianMultipleBarrier ;

typedef dcomplex(*CmplxF)(dcomplex, void *);

typedef enum { FFT, EULER } InvMeth;

/* static double ext_parasian_cdf (double t, const PnlVect *b, double m, const P
 * static int ext_parasian_cdf_fft (PnlVect *p, const PnlVect *t, const PnlVect
 *                                     double m, const PnlVect *mu); */
static dcomplex laplace_ext_parasian_cdf(dcomplex l, ParasianMultipleBarrier *pa

/* Laplace inversion */
static void inverse_laplace_fft(PnlVect *res, PnlCmplxFunc *f, double T);

/* interface */
static void get_ext_parasian_params(ParasianMultipleBarrier *params,
                                   const PnlVect *b, double m, const PnlVect *m

/* CDS auxiliary pricing functions */
static void CDS_p(PnlVect *PL, PnlVect *DL, const PnlVect *mat,
                 const PnlVect *proba, double r, double R);

/* CDS interface */
static void price_CDS_ext(double *PL, double *DL, double *price,
                         double T, const ParasianMultipleBarrier *params,
                         double r, double R, InvMeth inv);

/*
 * Takes the parameters b, m, mu and construct the vector of
 * parameters passed to the other functions
 */
static void get_ext_parasian_params(ParasianMultipleBarrier *params,
                                   const PnlVect *b, double m, const PnlVect *m

{
    params->n      = mu->size;
    params->mu     = mu;
    params->m      = m;
    params->m_sq   = m * m;

```

```

    params->b          = b;
}

/*
 * -m \ pm \ sqrt{m^2 + 2 (1 + mu) }
 */
static dcomplex R(dcomplex l, double m, double m_sq, double mu, int plus_or_minus)
{
    dcomplex z;

    z = Csqrt(RCadd(m_sq, RCmul(2., CRadd(1, mu))));
    if (plus_or_minus == 1)
    {
        z = RCadd(-m, z);
    }
    else
    {
        z = RCadd(-m, Cminus(z));
    }
    return z;
}

static void P_mat(PnlMatComplex *P, dcomplex l, double m, double m_sq,
                  double mu1, double mu2, double b)
{
    dcomplex rp2, rm2, rp1, rm1;

    rp2 = R(l, m, m_sq, mu2, 1);
    rm2 = R(l, m, m_sq, mu2, -1);
    rp1 = R(l, m, m_sq, mu1, 1);
    rm1 = R(l, m, m_sq, mu1, -1);

    pnl_mat_complex_set(P, 0, 0, Cmul(Csub(rp2, rm1), Cexp(RCmul(b, Csub(rm1, rm2))));
    pnl_mat_complex_set(P, 0, 1, Cmul(Csub(rp2, rp1), Cexp(RCmul(b, Csub(rp1, rm2))));
    pnl_mat_complex_set(P, 1, 0, Cmul(Csub(rm1, rm2), Cexp(RCmul(b, Csub(rm1, rp2))));
    pnl_mat_complex_set(P, 1, 1, Cmul(Csub(rp1, rm2), Cexp(RCmul(b, Csub(rp1, rp2))));

    pnl_mat_complex_div_dcomplex(P, Csub(rp2, rm2));
}

static void invA_mat(PnlMatComplex *invA, dcomplex l, double m, double m_sq,

```

```

                                double mu, double b)
{
    dcomplex rp, rm;

    rp = R(1, m, m_sq, mu, 1);
    rm = R(1, m, m_sq, mu, -1);

    pnl_mat_complex_set(invA, 0, 0, Cmul(rp, Cexp(RCmul(-b, rm))));
    pnl_mat_complex_set(invA, 0, 1, Cminus(Cexp(RCmul(-b, rm))));
    pnl_mat_complex_set(invA, 1, 0, Cmul(Cminus(rm), Cexp(RCmul(-b, rp))));
    pnl_mat_complex_set(invA, 1, 1, Cexp(RCmul(-b, rp)));

    pnl_mat_complex_div_dcomplex(invA, Csub(rp, rm));
}

/*
 * Laplace transform of the CDF of the parasian time
 */
static dcomplex laplace_ext_parasian_cdf(dcomplex l, ParasianMultipleBarrier *pa
{
    int slice, n, i;
    double m, m_sq, mu_1, mu_2, b;
    dcomplex res;
    PnlMatComplex *P, *Pi_i, *Pi_n, *tmpm, *invA;
    PnlVectComplex *vi, *vn, *x, *beta, *tmpv;

    n = params->n;
    m = params->m;
    m_sq = params->m_sq;
    P = pnl_mat_complex_create(2, 2);
    Pi_i = pnl_mat_complex_create(2, 2);
    Pi_n = pnl_mat_complex_create(2, 2);
    tmpm = pnl_mat_complex_create(2, 2);
    invA = pnl_mat_complex_create(2, 2);
    vi = pnl_vect_complex_create_from_dcomplex(2, CZERO);
    vn = pnl_vect_complex_create_from_dcomplex(2, CZERO);
    x = pnl_vect_complex_create_from_dcomplex(2, CZERO);
    beta = pnl_vect_complex_create_from_dcomplex(2, CZERO);
    tmpv = pnl_vect_complex_create(2);
    pnl_mat_complex_set_id(Pi_n);

```

```

pnl_mat_complex_set_id(Pi_i);

/* search for the index i s.t b_i <= 0 < b_{i-1}
 * this index is stored in slice */
slice = 0;
while (slice < n - 1 && pnl_vect_get(params->b, slice) > 0)
{
    slice ++;
}

/* Compute v_{slice} and v_{n-1} */
/* Compute Pi_{slice} and Pi_{n-1} */
for (i = 0 ; i < n - 1 ; i++)
{
    mu_1 = GET(params->mu, i);
    mu_2 = GET(params->mu, i + 1);
    b = GET(params->b, i);
    invA_mat(invA, l, m, m_sq, mu_2, b);
    pnl_vect_complex_set(x, 1, CZERO);
    pnl_vect_complex_set(x, 0, Csub(Cinv(CRadd(1, mu_1)),
                                   Cinv(CRadd(1, mu_2))));
    P_mat(P, l, m, m_sq, mu_1, mu_2, b);
    pnl_mat_complex_mult_mat_inplace(tmpm, P, Pi_n);
    pnl_mat_complex_clone(Pi_n, tmpm);
    pnl_mat_complex_mult_vect_inplace(tmpv, P, vn);
    pnl_vect_complex_clone(vn, tmpv);
    pnl_mat_complex_lAxpby(CONE, invA, x, CONE, vn);
    if (i == slice - 1)
    {
        pnl_vect_complex_clone(vi, vn);
        pnl_mat_complex_clone(Pi_i, Pi_n);
    }
}

/* Compute beta_1^- */
pnl_vect_complex_set(beta, 0, Cdiv(Cminus(pnl_vect_complex_get(vn, 0)),
                                   pnl_mat_complex_get(Pi_n, 0, 0)));

/* Compute beta_i */
pnl_mat_complex_lAxpby(CONE, Pi_i, beta, CONE, vi); /* vi += Pi * beta */
pnl_vect_complex_clone(beta, vi);

```

```

/* force beta_n^- to be 0 */
if (slice == n - 1)
{
    pnl_vect_complex_set(beta, 0, CZERO);
}

res = Csub(Csub(Cinv(l), Cinv(CRadd(l, GET(params->mu, slice)))),
           pnl_vect_complex_sum(beta));

pnl_mat_complex_free(&P);
pnl_mat_complex_free(&Pi_i);
pnl_mat_complex_free(&Pi_n);
pnl_mat_complex_free(&tmpm);
pnl_mat_complex_free(&invA);
pnl_vect_complex_free(&vi);
pnl_vect_complex_free(&vn);
pnl_vect_complex_free(&x);
pnl_vect_complex_free(&beta);
pnl_vect_complex_free(&tmpv);

return res;
}

/* /\ *
 * * Computes the CDF of the Parasian time using Euler inversion
 * *\ /
 * static double ext_parasian_cdf (double t, const PnlVect *b, double m, const P
 * {
 *     ParasianMultipleBarrier params;
 *     PnlCmplxFunc f;
 *
 *     get_ext_parasian_params (&params, b, m, mu);
 *     f.params = &params;
 *     f.F = (CmplxF) laplace_ext_parasian_cdf;
 *
 *     return pnl_ilap_euler (&f, t, 15, 15);
 * } */

/* /\ *
 * * Computes the CDF of the Parasian time using FFT
 * *\ /

```

```

* static int ext_parasian_cdf_fft (PnlVect *p, const PnlVect *t, const PnlVect
* {
*     int                n_mat, i, j;
*     double             ti, tj, T, h;
*     ParasianMultipleBarrier params;
*     PnlVect            *proba;
*     PnlCmplxFunc       lap;
*
*     n_mat = t->size;
*     T = pnl_vect_get (t, n_mat-1);
*     get_ext_parasian_params (&params, b, m, mu);
*     lap.params = &params;
*     lap.F = (CmplxF) laplace_ext_parasian_cdf;
*
*     proba = pnl_vect_create (0);
*     inverse_laplace_fft (proba, &lap, T);
*
*     /\ * proba contains the cdf on the grid used by the fft inversion algorithm
*     * need to extract the values corresponding to the dates given in t
*     *\ /
*     pnl_vect_resize (p, n_mat);
*     j = 0;
*     h = T / (proba->size - 1);
*     tj = 0.;
*     for ( i=0 ; i<n_mat ; i++)
*     {
*         ti = pnl_vect_get (t, i);
*         while (ti > tj)
*         {
*             tj += h; j++;
*         }
*         if ( ti != tj )
*         {
*             printf ("t contains values not in the FFT grid\ n");
*             return FAIL;
*         }
*         else
*         {
*             pnl_vect_set (p, i, pnl_vect_get (proba, j));
*         }
*     }

```

```

*
*   pnl_vect_free(&proba);
*   return OK;
* } */

/** Inversion functions */

/*
* FFT algorithm to invert a Laplace transform
* res : a real vector containing the result of the inversion. We know that
* the imaginary part of the inversion vanishes.
* f : the Laplace transform to be inverted
* T : the time horizon up to which the function is to be recovered
*/
static void inverse_laplace_fft(PnlVect *res, PnlCmplxFunc *f, double T)
{
    PnlVectComplex *fft;
    int            i, N;
    double         a;
    double         omega;
    double         h, f_a, eps, time_step;
    dcomplex       fac, mul;

    eps = 1E-5;
    h = 5 * M_PI / (8 * T); /* pour ce h, T doit être un multiple de 10, à changer
    a = h * log(1 + 1 / eps) / (2 * M_PI);

    N = MAX(sqrt(exp(a * T) / eps), h / (2 * M_PI * eps)) ;
    N = pow(2, ceil((log(N) / log(2))));
    time_step = M_2PI / (N * h);

    fft = pnl_vect_complex_create(N);
    pnl_vect_resize(res, N);

    fac = CIexp(-M_2PI / N);
    mul = fac;
    f_a = Creal(PNL_EVAL_FUNC(f, Complex(a, 0)));
    omega = h;

    for (i = 0 ; i < N ; i++)

```



```

    {
        pnl_vect_complex_set(fft, i, Cmul(mul, PNL_EVAL_FUNC(f, Complex(a, - omega
        omega += h;
        mul = Cmul(mul, fac);
    }
    pnl_fft_inplace(fft);
    mul = Complex(1., 0.);

    for (i = 0 ; i * time_step <= T ; i++)
    {
        double res_i;
        res_i = Creal(Cmul(pnl_vect_complex_get(fft, i), mul));
        mul = Cmul(mul, fac);
        res_i = (h / M_PI) * exp(a * (i + 1) * time_step) * (res_i + 0.5 * f_a);
        pnl_vect_set(res, i, res_i);
    }
    pnl_vect_resize(res, i);
    pnl_vect_complex_free(&fft);
}

/*
 * PL: payment leg (output parameter)
 * DL: default leg (output parameter)
 * same size as mat (maturities)
 * proba: cdf of the default time computed on the regular
 * time grid with step size max(mat)/(size(proba)-1)
 * r : interest rate
 * R : recovery rate
 */
static void CDS_p(PnlVect *PL, PnlVect *DL, const PnlVect *mat, const PnlVect *p
                double r, double R)
{
    double T, ts, t;
    int ind_mat;
    double int_co; /*  $\int_0^T e^{-rs} P(\tau \leq s) ds$  */
    double int_disc; /*  $\int_0^T e^{-rs} (s - T_{\beta(s)-1}) P(\tau \leq s) ds$  */
    int simpson;
    int i;
    int last_payment;

```

```

double proba_i, update_disc, update_co, pli;

T          = pnl_vect_get(mat, mat->size - 1);
ts         = T / (proba->size - 1);
ind_mat    = 0;
int_co     = 0;
int_disc   = 0;
simpson    = 4;
last_payment = 0;
/* Simpson rule */

for (i = 0, t = ts ; t <= T ; i++, t += ts)
{
    proba_i = pnl_vect_get(proba, i);
    update_co = (exp(-r * t) * proba_i);
    int_co += simpson * update_co;
    update_disc = r * exp(-r * t) * (t - last_payment * 0.25) * (1. - proba_i)
    if (t == (last_payment + 1) * 0.25)
    {
        int_disc += update_disc;
        last_payment++;
    }
    else int_disc += simpson * update_disc;

    if (t == pnl_vect_get(mat, ind_mat))
    {
        /* à une maturité, simpson=2 */
        pli = - (int_co - update_co) * ts / 3 - (int_disc - update_disc) * ts
        pli += (1 - exp(-r * t)) / r;
        pnl_vect_set(PL, ind_mat, pli);
        pnl_vect_set(DL, ind_mat, r * (int_co - update_co) * ts / 3 + exp(-r *
        ind_mat++;
    }

    if (simpson == 4) simpson = 2;
    else simpson = 4;
}
/* mutiplier by LGD (= 1. - R) */
pnl_vect_mult_double(DL, 1. - R);
}

```

```

/*
 * Computes the legs and prices of CDS
 * PL : Premium Leg (output parameter)
 * DL : Default Leg (output parameter)
 * price : CDS price (output parameter)
 * T : maturity
 * tab : parameters of the CDS [mu, m, m^2, b]
 * r: interest rate
 * inv : flag for the inversion (FFT or EULER)
 * muminus : mu_-
 */
static void price_CDS_ext(double *PL, double *DL, double *price,
                        double T, const ParasianMultipleBarrier *tab,
                        double r, double R, InvMeth inv)
{

    int          n_mat, n, i;
    PnlVect      *proba, *mat;
    PnlCmplxFunc lap;
    PnlVect      *price_v, *pl_v, *dl_v;
    double       t;

    lap.F = (CmplxF) laplace_ext_parasian_cdf;
    lap.params = (ParasianMultipleBarrier *) tab; /* to avoid warning because of c

switch (inv)
{
    case FFT:
        mat = pnl_vect_create_from_list(2, T, 10.);
        n_mat = 2;
        proba = pnl_vect_create(0);
        inverse_laplace_fft(proba, &lap, 10);
        break;
    case EULER:
        n_mat = 1;
        n = (int) ceil(T * 32.); /* 32 steps per year */
        mat = pnl_vect_create_from_double(1, T);
        proba = pnl_vect_create_from_double(n + 1, 0.);
        for (i = 0, t = 1. / 32. ; i < n ; i++, t += 1. / 32.)
            {

```

```

        pnl_vect_set(proba, i, pnl_ilap_euler(&lap, t, 15, 15));
    }
    break;
default:
    abort();
}

pl_v = pnl_vect_create(n_mat);
dl_v = pnl_vect_create(n_mat);

CDS_p(pl_v, dl_v, mat, proba, r, R);
price_v = pnl_vect_copy(dl_v);
pnl_vect_div_vect_term(price_v, pl_v);

/* extract results */
*price = pnl_vect_get(price_v, 0);
*PL = pnl_vect_get(pl_v, 0);
*DL = pnl_vect_get(dl_v, 0);

pnl_vect_free(&mat);
pnl_vect_free(&proba);
pnl_vect_free(&pl_v);
pnl_vect_free(&dl_v);
pnl_vect_free(&price_v);
}

static int CALC(AP_Alfonsi_Lelong)(void *Opt, void *Mod, PricingMethod *Met)
{
    TYPEOPT          *ptOpt;
    TYPEMOD          *ptMod;
    ParasianMultipleBarrier params;
    PnlVect          *b;
    double            price, pl, dl;
    double            T, R, r, sigma, l, m;
    int               i;
    InvMeth           inv;

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

```

```

T = ptOpt->Maturity.Val.V_DATE;
r = log(1. + ptMod->R.Val.V_DOUBLE / 100.);
sigma = ptMod->Sigma.Val.V_DOUBLE;
m = (r - ptMod->alpha.Val.V_DOUBLE - sigma * sigma / 2) / sigma;
b = pnl_vect_create(ptMod->L.Val.V_PNLVECT->size);
R = (ptOpt->Recovery).Val.V_PDOUBLE;

if (Met->Par[0].Val.V_ENUM.value == 1 || T > 10)
{
    inv = EULER;
}
else
{
    inv = FFT;
}

for (i = 0 ; i < b->size ; i++)
{
    l = pnl_vect_get(ptMod->L.Val.V_PNLVECT, i);
    pnl_vect_set(b, i, log(1 / ptMod->S0.Val.V_PDOUBLE) / sigma);
}

get_ext_parasian_params(&params, b, m, ptMod->mu.Val.V_PNLVECT);
price_CDS_ext(&pl, &dl, &price, T, &params, r, R, inv);

Met->Res[0].Val.V_DOUBLE = dl;
Met->Res[1].Val.V_DOUBLE = pl;
Met->Res[2].Val.V_DOUBLE = price;

pnl_vect_free(&b);

return OK;
}

static int CHK_OPT(AP_Alfonsi_Lelong)(void *Opt, void *Mod)
{
    return OK;
}

#endif //PremiaCurrentVersion
static PremiaEnumMember InversionMembers[] =

```

```

{
    { "Euler Acceleration", 1 },
    { "FFT", 2 },
    { NULL, NULLINT }
};

static DEFINE_ENUM(Inversion, InversionMembers);

static int MET(Init)(PricingMethod *Met, Option *Opt)
{
    if (Met->init == 0)
    {
        Met->Par[0].Val.V_ENUM.value = 2;
        Met->Par[0].Val.V_ENUM.members = &Inversion;
        Met->init = 1;
    }
    return OK;
}

PricingMethod MET(AP_Alfonsi_Lelong) =
{
    "AP_Alfonsi_Lelong",
    {
        {"Inversion method", ENUM, {100}, ALLOW},
        {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CALC(AP_Alfonsi_Lelong),
    { {"Default Leg", DOUBLE, {100}, FORBID},
      {"Premium Leg", DOUBLE, {100}, FORBID},
      {"Price", DOUBLE, {100}, FORBID},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CHK_OPT(AP_Alfonsi_Lelong),
    CHK_ok,
    MET(Init)
};

```