

[Help](#)

```
#include "
href../../mod/sli/sli_stdndc/sli_stdndc_h_src.pdfslsli_stdndc.h"
#include "pnl/pnl_vector.h"
#include "pnl/pnl_random.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (20014+2) //The "#el
static int CHK_OPT(MC_AlfonsiLabartLelong)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(MC_AlfonsiLabartLelong)(void *Opt, void *Mod, PricingMethod *Met)
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else

/**
 * Structure for describing one particle
 */
typedef struct _particle Particule;
struct _particle
{
    int index; /*!< index of the particle */
    int n; /* number of jumps sofar */
    PnlVect *Xgrid; /*!< discretization grid of X: its jumping times */
};

/**
 * Structure for storing all the parameters describing the particle system
 */
typedef struct _sys_params Sys_params;
struct _sys_params
{
    double (*f)(double x, double min_f, double max_f); /*!< function f*/
    double min_f; /*!< lower bound of f */
    double max_f; /*!< upper bound of f */
    double max_lambda; /*!< upper bound of function lambda */
    int M; /* Number of companies */
}
```

```

int N; /*!< Number of particles */
double Nsteps; /*!< number of steps for discretizing Y */
double h; /*!< discretization timestep for Y on [0,T] */
double T; /* maturity */
double X0; /*!< initial value of X */
double Y0; /*!< initial value of Y */
double Yparam[3]; /*!< parameters of Y : drift (alpha), vol (sigma), jump (gamma)
PnlRng *rng;
int n_loop_time; /*!< number of times we enter cond_expect */
int n_compute_expect; /*!< number of times we actually compute E(|) */
PnlVect *Numerator; /*!< vector of \ sum_{i=1}^N f(Y_t^i) 1_{X_t^i = k} for all k
PnlVect *Denominator; /*!< vector of \ sum_{i=1}^N 1_{X_t^i = k} for all possible k
};

/*
 * Local default intensity
 */
static double lambda(double t, double x, int M, double lambda_max)
{
    return (1 - x / M) * lambda_max;
    /* return sqrt( 1 - x / M) * lambda_max; */
    /* return M * ( 0.01 + 0. * (MIN(x, 10) * 0.001)) * (1 - x/M); */
}

/**
 * Computes the value of Y at the next time step.
 *
 *  $dY_t = -a Y_t \log(Y_t) + \sigma Y_t dW_t + \gamma Y_t dX_t$ 
 *
 * Remember that X is a pure jump process, with jump times included in the
 * grid points.
 *
 * The discretization is implemented as follows
 *
 * 1. Run the Euler scheme on the  $\log(Y_t)$  (continuous part of  $Y_t$ ),
 *    which is actually an OU process
 * 2. Integrate the jump at  $t+h$  if  $\Delta X > 0$ 
 *
 * @param t last time step
 * @param Yold value of Y at t
 * @param Xold value of X at t

```

```

* @param DeltaX jump of X at t+h
* @param h timestep
* @param rng random number generator
* @param P parameters of the system [a, sigma, gamma]
*
* @return the value of Y at t+h
*/
static double next_step_Y(double t, double Yold, double Xold,
                        double DeltaX, double h, PnlRng *rng,
                        const Sys_params *P)
{
    double Yt;
    double a, sigma, gamma;
    a = P->Yparam[0];
    sigma = P->Yparam[1];
    gamma = P->Yparam[2];
    Yt = log(Yold);
    Yt += (-sigma * sigma / 2 - a * Yt) * h + sigma * sqrt(h) * pnl_rng_normal(rng);
    Yt = exp(Yt);
    Yt += gamma * Yt * DeltaX;
    return Yt;
}

static double f(double x, double min_f, double max_f)
{
    return MIN(MAX(x, min_f), max_f);
}

/**
* Creates a system of particules (ie. an array of Particule object)
*
* @param param a Sys_param object used to initialize the particule system
* @return an array of Particule
*/
static Particule *create_system(const Sys_params *param)
{
    Particule *p;
    int i;
    int Njump;

```

```

p = malloc(param->N * sizeof(Particule));
Njump = (int) MAX(ceil(param->max_lambda * param->max_f / param->min_f * param

for (i = 0 ; i < param->N ; i++)
{
    p[i].index = i;
    p[i].n = 0;
    p[i].Xgrid = pnl_vect_create(Njump);
    PNL_LET(p[i].Xgrid, 0) = 0.;
}

return p;
}

static void delete_system(Particule **P, int N)
{
    int i;
    for (i = 0 ; i < N ; i++)
    {
        Particule *p = &((*P)[i]);
        pnl_vect_free(&(p->Xgrid));
    }
    free(*P);
    *P = NULL;
}

/**
 *
 * Initializes the parameters of the system
 *
 * @param p system structure
 * @param min_f lower bound of f
 * @param max_f upper bound of f
 * @param max_lambda upper bound of lambda
 * @param M number of names
 * @param N number of particles
 * @param Nsteps number of discretization timesteps
 * @param T maturity time
 * @param Y0 initial value of Y
 * @param alpha drift of Y
 * @param sigma volatility of Y

```

```

* @param gamma jump intensity of Y
* @param rng random number generator
*/
static void init_sys_params(Sys_params *p, double min_f, double max_f, double
                           max_lambda, int M, int N, double Nsteps, double
                           T, double Y0, double alpha, double sigma, double
                           gamma, PnlRng *rng)
{
    int Njump;
    p->f = f;
    p->min_f = min_f;
    p->max_f = max_f;
    p->max_lambda = max_lambda;
    p->N = N;
    p->M = M;
    p->Nsteps = Nsteps;
    p->h = T / Nsteps;
    p->T = T;
    p->Y0 = Y0;
    p->Yparam[0] = alpha;
    p->Yparam[1] = sigma;
    p->Yparam[2] = gamma;
    p->rng = rng;
    p->n_loop_time = 0;
    p->n_compute_expect = 0;

    Njump = (int) MAX(ceil(max_lambda * max_f / min_f * T), 1);
    p->Numerator = pnl_vect_create_from_zero(Njump);
    p->Denominator = pnl_vect_create_from_zero(Njump);
    LET(p->Numerator, 0) = p->f(Y0, min_f, max_f) * N;
    LET(p->Denominator, 0) = N;
}

/**
* Free the dynamic members of a Sys_params.
*
* @param p
*/
static void free_sys_params(Sys_params *p)
{
    pnl_vect_free(&(p->Numerator));

```

```

    pnl_vect_free(&(p->Denominator));
}

/**
 * Computes  $E(f(Y_t)|X_{t^-})$  using the particle system
 *
 * 
$$\frac{\sum_{i=1}^N f(Y_t^i) 1_{\{X_t^i = X_t^I\}}}{\sum_{i=1}^N 1_{\{X_t^i = X_t^I\}}}$$

 *
 * @param I index of the particle considered
 * @param P array of particles
 * @param param parameters describing the system
 * @param Y value of Y
 * @param num (input/output) numerator of the conditionnal expectation
 * @param denom (input/output) denominator of the conditionnal expectation
 * @param X_old last point at which the conditionnal expectation was
 * @param Y_old value of Y(jump_index) at the previous step
 * @param jump_index If -1 last jump was rejected, if non-negative
 * holds the index of the particle which has jumped last
 * @param is_next_step 0 if the last computation was carried out within the
 * same discretization time block and 1 otherwise
 */
static void cond_expect_new(int I, const Particule *P, Sys_params *param,
                           PnlVect *Y, double *num, double *denom, int X_old,
                           double Y_old, int jump_index, int is_next_step)
{
    int j;
    int X_I;

    X_I = P[I].n;

    /**
     * Resize Numerator and Denominator if needed.
     * Both have the same size.
     */
    if (X_old >= param->Numerator->size)
    {
        pnl_vect_resize_from_double(param->Numerator, 2 * param->Numerator->size,
        pnl_vect_resize_from_double(param->Denominator, 2 * param->Numerator->size
    }
}

```

```

param->n_loop_time++;

/*
 * Update Denominator
 */
if (jump_index >= 0)
{
    LET(param->Denominator, X_old) -= 1;
    LET(param->Denominator, X_old + 1) += 1;
}
*denom = GET(param->Denominator, X_I);

/*
 * Update Numerator
 */
if (is_next_step)
{
    /*
     * We cannot benefit from the previous computations and re-compute
     * everything again
     */
    int k;
    param->n_compute_expect++;
    for (k = 0 ; k < param->Numerator->size ; k++)
    {
        double Nk = 0.;
        for (j = 0 ; j < param->N ; j++)
        {
            if (P[j].n == k)
            {
                Nk += (*(param->f))(GET(Y, j), param->min_f, param->max_f);
            }
        }
        LET(param->Numerator, k) = Nk;
    }
}
else
{
    if (jump_index >= 0)
    {
        LET(param->Numerator, X_old) -= f(Y_old, param->min_f, param->max_f);
    }
}

```

```

        LET(param->Numerator, X_old + 1) += f(GET(Y, jump_index), param->min_f
    }
}
*num = GET(param->Numerator, X_I);
}

/**
 * Simulates the particule system ( $X^i$ ,  $Y^i$ )
 *
 * The discretisation grid of  $Y^i$  consists of all the jump times of only the
 *  $X^i$  particule and we ensure that the discretisation step is at most h
 *
 * @param param parameters of the system
 * @param M number of nouns in the CDO
 *
 * @return an array of particules
 */
static Particule *simul_system2(Sys_params *param)
{
    Particule *P;
    PnlRng *rng;
    PnlVect *Y, *grid_t;
    int i, has_jumped, tick, X_old;
    double t, Y_old;
    const double E_lambda = param->max_lambda * param->max_f / param->min_f * para

    P = create_system(param);
    rng = param->rng;
    grid_t = pnl_vect_create_from_zero(param->N); /* component i is the last disc
    Y = pnl_vect_create(param->N); /* Value of  $Y^i$  at grid_t(i) */
    pnl_vect_set_double(Y, param->Y0);

    t = 0; /* current time */
    has_jumped = -1; /* flag to determine if the last jump was accepted */
    X_old = -1;
    Y_old = -1;
    tick = 0; /* current index on the regular time grid with step h, we
                always have tick x param->h <= t */

    /*

```



```

* Evolution loop
*/
for (; ;)
{
    int is_next_step; /* flag to know if we have crossed a new discretisation
    int I; /* index of the selected particule */
    double E; /* Exponential variable */
    double ratio; /* acceptance ratio */
    int Xt; /* value of X before the jump at time t */
    double num, denom;

    I = (int) floor(pnl_rng_uni(rng) * param->N);
    E = pnl_rng_exp(E_lambda, rng);

    Xt = P[I].n;
    t += E;
    if (t > param->T) break;
    is_next_step = FALSE;
    /*
    * Have we jumped over a new tick on the grid?
    */
    while (t > (tick + 1) * param->h)
    {
        tick++;
        is_next_step = TRUE;
        for (i = 0 ; i < param->N ; i++)
        {
            /*
            * Discretize Y up to the greatest time step not larger than t
            * Note the discretization grid of  $Y^i$  is NOT regular: it contains
            * the regular time grid and the jump times of the particule  $X^i$ .
            *
            * With this choice, all  $Y^i$  are at least discretized on the
            * regular time grid.
            */
            int t_i = GET(grid_t, i);
            if (t_i < tick * param->h)
            {
                /*
                * If the last discretization date was a grid tick, dt = h
                * If not, dt is the length to reach the next grid tick

```

```

        */
        double dt = MIN(param->h, tick * param->h - t_i);
        LET(Y, i) = next_step_Y(t_i, GET(Y, i), P[i].n, 0., dt, rng, p);
        t_i += dt;
        LET(grid_t, i) = t_i;
    }
}

ratio = param->min_f / (param->max_lambda * param->max_f)
        * lambda(t, Xt, param->M, param->max_lambda) *
        (*param->f)(GET(Y, I), param->min_f, param->max_f);
cond_expect_new(I, P, param, Y, &num, &denom, X_old, Y_old, has_jumped, is
ratio /= (num / denom);

/* Save Xt for next iteration */
X_old = Xt;
Y_old = GET(Y, I);
/*
 * We accept the jump
 */
if (pnl_rng_uni(rng) <= ratio)
{
    /*
     * Discretize  $Y^I$  up to time t and integrate the jump to  $Y^I$ .
     * Note that  $X_{t^-}$  is needed to update the continuous part of
     * Y, so we must update Y before X
     */
    LET(Y, I) = next_step_Y(GET(grid_t, I), GET(Y, I), P[I].n, 1.,
                            t - GET(grid_t, I), rng, param);
    LET(grid_t, I) = t; /* advance discretization grid of  $Y^I$  */
    /*
     * Do we need to enlarge the particule attributes?
     */
    if (P[I].Xgrid->size <= P[I].n + 1)
    {
        pnl_vect_resize(P[I].Xgrid, 2 * P[I].n);
    }
    /*
     * Save the jump
     */
}

```

```

        (P[I].n) ++;
        PNL_LET(P[I].Xgrid, P[I].n) = t;
        has_jumped = I;
    }
    else /* ( pnl_rng_uni (rng) > ratio ) */
    {
        has_jumped = -1;
    }
}

/*
 * Resize all the arrays to match the number of jumps of each particule
 */
for (i = 0 ; i < param->N ; i++)
{
    pnl_vect_resize(P[i].Xgrid, P[i].n + 1);
}
pnl_vect_free(&Y);
pnl_vect_free(&grid_t);
return P;
}

#define Hab(x) ( x < A ? 0 : ( x < B ? x : B) )

/**
 * Compute the Premium and Default Legs
 *
 * @param P the particle system
 * @param params parameters of the problems
 * @param r Interest rate
 * @param Recovery Recovery rate
 * @param tranches vector of tranches
 * @param[out] PL the vector of values of the premium leg for each tranch
 * @param[out] DL the vector of values of the default leg for each tranch
 */
static void price_legs_particle(const Particule *P, const Sys_params *params,
                                double r, double Recovery, int Fcoupons,
                                const PnlVect *tranches, PnlVect *PL, PnlVect *DL)
{
    int i, j, k, t;
    pnl_vect_set_zero(DL);

```

```

pnl_vect_set_zero(PL);

for (i = 0 ; i < params->N ; i++)
{
    for (j = 1 ; j < P[i].n ; j++)
    {
        double tau_j = GET(P[i].Xgrid, j);
        double Tj = floor(tau_j * Fcoupons) / Fcoupons;
        double discount_tau_j = exp(-r * tau_j);
        double Lj = j * (1 - Recovery) / params->M;
        double Lj_minus = (j - 1) * (1 - Recovery) / params->M;

        /* Only for the regular payments */
        double tau_j_minus = GET(P[i].Xgrid, j - 1);
        int start = ceil(tau_j_minus * Fcoupons); /* first payment tick after
        int end = floor(tau_j * Fcoupons); /* last payment tick before tau_j
        double discount = 0.; /* Add discounts between T_start and T_end */
        for (t = start ; t <= end ; t++)
        {
            discount += exp(-r * t / (double) Fcoupons);
        }

        for (k = 0 ; k < tranches->size - 1 ; k++)
        {
            double A = GET(tranches, k);
            double B = GET(tranches, k + 1);
            double Hj = Hab(Lj);
            double Hj_minus = Hab(Lj_minus);

            LET(DL, k) += discount_tau_j * (Hj - Hj_minus);
            /* Accrued margin of the premium leg */
            LET(PL, k) += discount_tau_j * (Hj - Hj_minus) * (tau_j - Tj);
            /* Regular paiements of the premium leg */
            LET(PL, k) += discount * (B - A - Hj_minus);
        }
    }
}

pnl_vect_div_scalar(DL, params->N);
pnl_vect_div_scalar(PL, params->N);
}

```

```

int CALC(MC_AlfonsiLabartLelong)(void *Opt, void *Mod, PricingMethod *Met)
{
    Sys_params params;
    Particule *P;
    PnlRng *rng;
    PnlVect *tranches, *DL, *PL, *Price;
    double T, r, Recovery, Y0, alpha, sigma, gamma, max_lambda, min_f, max_f;
    int Ncomp, Nparticles, Nsteps, Fcoupons;
    TYPEOPT *ptOpt = (TYPEOPT *)Opt;
    TYPEMOD *ptMod = (TYPEMOD *)Mod;

    T = ptOpt->maturity.Val.V_PDOUBLE;
    Fcoupons = ptOpt->NbPayment.Val.V_PINT;
    tranches = ptOpt->tranch.Val.V_PNLVECT;

    Ncomp = ptMod->Ncomp.Val.V_INT;
    r = ptMod->r.Val.V_PDOUBLE;
    Recovery = ptMod->Recovery.Val.V_PDOUBLE;
    max_lambda = ptMod->Intensity.Val.V_PDOUBLE;
    Y0 = ptMod->Y0.Val.V_DOUBLE;
    sigma = ptMod->sigma.Val.V_DOUBLE;
    gamma = ptMod->gamma.Val.V_DOUBLE;
    alpha = ptMod->alpha.Val.V_DOUBLE;
    min_f = 1. / 3.;
    max_f = 3.;

    Nparticles = Met->Par[0].Val.V_PINT;
    Nsteps = Met->Par[1].Val.V_PINT;
    rng = PnlRngArray[Met->Par[2].Val.V_ENUM.value];
    pnl_rng_sseed(rng, 0);

    Price = Met->Res[0].Val.V_PNLVECT;
    DL = Met->Res[1].Val.V_PNLVECT;
    PL = Met->Res[2].Val.V_PNLVECT;

    init_sys_params(&params, min_f, max_f, max_lambda, Ncomp, Nparticles,
                    Nsteps, T, Y0, alpha, sigma, gamma, rng);

    P = simul_system2(&params);
    price_legs_particle(P, &params, r, Recovery, Fcoupons, tranches, PL, DL);
}

```

```

    pnl_vect_clone(Price, DL);
    pnl_vect_div_vect_term(Price, PL);
    pnl_vect_mult_scalar(Price, 1000);

    delete_system(&P, Nparticles);
    free_sys_params(&params);

    return OK;
}

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

#endif

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->Par[0].Val.V_INT = 5000;
        Met->Par[1].Val.V_INT = 100;
        Met->Par[2].Val.V_ENUM.value = 0;
        Met->Par[2].Val.V_ENUM.members = &PremiaEnumRNGs;

        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(MC_AlfonsiLabartLelong) =

```

```

{
  "MC_AlfonsiLabartLelong",
  { {"N particles", PINT, {4}, ALLOW},
    {"N time steps", PINT, {4}, ALLOW},
    {"RandomGenerator", ENUM, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_AlfonsiLabartLelong),
  { {"Price(bp)", PNLVECT, {100}, FORBID},
    {"D_leg", PNLVECT, {100}, FORBID},
    {"P_leg", PNLVECT, {100}, FORBID},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(MC_AlfonsiLabartLelong),
  CHK_ok,
  MET(Init)
};

```