

[Help](#)

```
#include "
href../../../../mod/kould/kould_std/kould_std_h_src.pdfkould_std.h"
#include "
href../../../../common/enums_h_src.pdfenums.h"
#include <stdlib.h>
#include "pnl/pnl_cdf.h"
#include "pnl/pnl_root.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2011+2) //The "#els

static int CHK_OPT(MC_Kou_Digital_LRM)(void *Opt, void *Mod)
{
    return NONACTIVE;
}

int CALC(MC_Kou_Digital_LRM)(void *Opt, void *Mod, PricingMethod *Met)
{
    return AVAILABLE_IN_FULL_PREMIA;
}

#else

//Algorithme de tri croissant
static void tri_up(double *x, int size)
{
    double sup, temp;
    int i, j, k = 0;
    for (i = 0; i < size - 1; i++)
    {
        sup = x[0];
        for (j = 0; j < size - i; j++)
        {
            if (x[j] > sup)
            {
                sup = x[j];
                k = j;
            }
        }
        if (k != size - i - 1)
```

```

        {
            temp = x[size - i - 1];
            x[size - i - 1] = x[k];
            x[k] = temp;
        }
    }

// the CGF of the Kou's model and its derivatives
static double kou_CGF(double u, double t, double sigma, double drift, double lambda,
double lambda_dam, double p)
{
    return t * (drift * u + sigma * sigma * u * u / 2 + lambda * (p * lambda_dap / (1 - p) *
    POW(lambda_dap - u, 2) + (1 - p) * lambda_dam * u));
}

static double kou_CGF_2diff(double u, double t, double sigma, double drift, double lambda,
double lambda_dam, double p)
{
    return t * (sigma * sigma + 2 * lambda * (p * lambda_dap / POW(lambda_dap - u, 3) +
    (1 - p) * lambda_dam));
}

static double kou_CGF_3diff(double u, double t, double sigma, double drift, double lambda,
double lambda_dam, double p)
{
    return 6 * t * lambda * (p * lambda_dap / POW(lambda_dap - u, 4) - (1 - p) * lambda_dam);
}

static double kou_CGF_4diff(double u, double t, double sigma, double drift, double lambda,
double lambda_dam, double p)
{
    return 24 * t * lambda * (p * lambda_dap / POW(lambda_dap - u, 5) + (1 - p) * lambda_dam);
}

//function used in pnl_root_newton_bisection
void func(double x, double *fx, double *dfx, void *temp)
{
    double *param = (double *) temp;
    *fx = param[0] * (param[2] + param[1] * param[1] * x + param[3] * (param[6] *
    POW(x, 2) + (1 - p) * lambda_dam * x));
    *dfx = param[0] * (param[1] * param[1] + 2 * param[3] * (param[6] * param[4] /
    POW(x, 3) + (1 - p) * lambda_dam));
}

//Estimate of the transition pdf
static double kou_pdf(double x, double t, double sigma, double drift, double lambda,
double lambda_dap, double lambda_dam, double p)
{

```

```

double s, lambda4, lambda3, *param, x_min, x_max, tol, temp1, temp2;
PnlFuncDFunc fdf_func;
int N_max;
param = (double *)malloc(8 * sizeof(double));
param[0] = t;
param[1] = sigma;
param[2] = drift;
param[3] = lambda;
param[4] = lambdap;
param[5] = lambdam;
param[6] = p;
param[7] = x;
fdf_func.params = param;
fdf_func.F = func;
x_min = -lambdam + 1e-1;
x_max = lambdap - 1e-1;
tol = 1e-6;
N_max = 1000;
pnl_root_newton_bisection(&fdf_func, x_min, x_max, tol, N_max, &s);
temp1 = kou_CGF(s, t, sigma, drift, lambda, lambdap, lambdam, p);
temp2 = kou_CGF_2diff(s, t, sigma, drift, lambda, lambdap, lambdam, p);
lambda3 = kou_CGF_3diff(s, t, sigma, drift, lambda, lambdap, lambdam, p) / POW
lambda4 = kou_CGF_4diff(s, t, sigma, drift, lambda, lambdap, lambdam, p) / POW
free(param);
return exp(temp1 - s * x) * (1 + (lambda4 / 8 - 5 * lambda3 * lambda3 / 24)) /
}

int Kou_Mc_Digital_saddlepoint(double S0, NumFunc_1 *P, double T, double r,
                                double divid, double sigma, double lambda,
                                double lambdap, double lambdam,
                                double p, int generator, int n_paths,
                                double *ptprice, double *priceerror,
                                double *ptDelta)
{
double payoff, *jump_time_vect, *X, *W, sum_payoff,
    sum_square_payoff, nu, u0, *jump_size_vect, var_payoff, K;
int i, j, k, jump_number, n_vect;

K = P->Par[0].Val.V_DOUBLE;
nu = ((r - divid) - sigma * sigma / 2 - lambda * (p * lambdap / (lambdap - 1)
sum_payoff = 0;

```

```

sum_square_payoff = 0;
n_vect = pnl_iround(1000 * lambda * T);
jump_size_vect = (double *)malloc(n_vect * sizeof(double));
jump_time_vect = (double *)malloc(n_vect * sizeof(double));
X = (double *)malloc(n_vect * sizeof(double));
W = (double *)malloc(n_vect * sizeof(double));
W[0] = 0;
X[0] = 0;
pnl_rand_init(generator, 1, 1);
for (i = 0; i < n_paths; i++)
{
    jump_number = pnl_rand_poisson(lambda * T, generator);
    jump_time_vect[0] = 0;
    // simulation of the jump's times and the size of the jumps
    for (j = 1; j <= jump_number; j++)
    {
        jump_time_vect[j] = pnl_rand_uni_ab(0., T, generator);
        u0 = pnl_rand_uni(generator);
        if (1 - p <= u0)
            jump_size_vect[j] = -log(1 - (u0 - 1 + p) / p) / lambdap;
        else
            jump_size_vect[j] = log(u0 / (1 - p)) / lambdam;
    }
    jump_time_vect[jump_number + 1] = T;
    jump_size_vect[jump_number + 1] = 0;
    tri_up(jump_time_vect, jump_number + 1); //rearranging jump's times in asc
    // simulation of the Brownian motion part at jump's times
    for (j = 1; j <= jump_number + 1; j++)
    {
        W[j] = sigma * pnl_rand_normal(generator) * sqrt(jump_time_vect[j] - j);
    }
    // simulation of one Levy process X at jump's times
    for (k = 1; k <= jump_number + 1; k++)
    {
        X[k] = X[k - 1] + (W[k] - W[k - 1]) + jump_size_vect[k];
    }
    payoff = (S0 * exp(X[jump_number + 1]) > K);
    sum_payoff += payoff;
    sum_square_payoff += payoff * payoff;
}
var_payoff = (sum_square_payoff - sum_payoff * sum_payoff / ((double)n_paths))

```

```

*ptprice = sum_payoff / n_paths;
*priceerror = 1.96 * sqrt(var_payoff) / sqrt((double)n_paths);
*ptDelta = kou_pdf(log(K / S0), T, sigma, nu, lambda, lambdap, lambdam, p) / S

free(jump_time_vect);
free(jump_size_vect);
free(X);
free(W);

return OK;
}

int CALC(MC_Kou_Digital_LRM)(void *Opt, void *Mod, PricingMethod *Met)
{
    TYPEOPT *ptOpt = (TYPEOPT *)Opt;
    TYPEMOD *ptMod = (TYPEMOD *)Mod;
    double r, divid;

    r = log(1. + ptMod->R.Val.V_DOUBLE / 100.);
    divid = log(1. + ptMod->Divid.Val.V_DOUBLE / 100.);

    return Kou_Mc_Digital_saddlepoint(
        ptMod->S0.Val.V_PDOUBLE,
        ptOpt->PayOff.Val.V_NUMFUNC_1,
        ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
        r, divid,
        ptMod->Sigma.Val.V_PDOUBLE,
        ptMod->Lambda.Val.V_PDOUBLE,
        ptMod->LambdaPlus.Val.V_PDOUBLE,
        ptMod->LambdaMinus.Val.V_PDOUBLE,
        ptMod->P.Val.V_PDOUBLE,
        Met->Par[0].Val.V_ENUM.value,
        Met->Par[1].Val.V_LONG,
        &(Met->Res[0].Val.V_DOUBLE),
        &(Met->Res[1].Val.V_DOUBLE),
        &(Met->Res[2].Val.V_DOUBLE));
}

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

```

```

    return WRONG;
}

#endif //PremiaCurrentVersion
static int MET(Init)(PricingMethod *Met, Option *Opt)
{
    if (Met->init == 0)
    {
        Met->init = 1;
        Met->HelpFilenameHint = "MC_Kou_Digital_lrm";
        Met->Par[0].Val.V_ENUM.value = 0;
        Met->Par[0].Val.V_ENUM.members = &PremiaEnumMCRNGs;
        Met->Par[1].Val.V_LONG = 100000;
    }
    return OK;
}
PricingMethod MET(MC_Kou_Digital_LRM) =
{
    "MC_Kou_Digital_LRM",
    {
        {"RandomGenerator", ENUM, {100}, ALLOW},
        {"N iterations", LONG, {100}, ALLOW},
        {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CALC(MC_Kou_Digital_LRM),
    {
        {"Price", DOUBLE, {100}, FORBID},
        {"Price Error", DOUBLE, {100}, FORBID},
        {"Delta", DOUBLE, {100}, FORBID},
        {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CHK_OPT(MC_Kou_Digital_LRM),
    CHK_ok,
    MET(Init)
};

```