

## Help

```

extern "C" {
#include "kould_stda.h"
#include "enums.h"
#include "pnl/pnl_finance.h"
}
#include "math/levy.h"
#include "math/fft.h"
#include "pnl/pnl_random.h"
extern "C" {

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2015+2) //The "#els
    static int CHK_OPT(MC_Icppi_KOU)(void *Opt, void *Mod)
    {
        return NONACTIVE;
    }
    int CALC(MC_Icppi_KOU)(void *Opt, void *Mod, PricingMethod *Met)
    {
        return AVAILABLE_IN_FULL_PREMIA;
    }
#else

    static int GapApproximateKou(double S0, double T, double r, double divid,
                                double sigma, double lambda, double lambdap, double
                                double strike, double betagap, int numberperiod,
    {
        double etam = 1. / lambdam;
        double price = lambda * (1. - proba) * etam / (1 + etam) * std::pow(strike /
        double cdf = lambda * (1. - proba) * exp(betagap / etam);

        *ptprice = price * (1. - exp(-T * (r + cdf))) / (r + cdf);

        return 0;
    }

    static int Value_Kou_Mc(double S0, double T, double r, double divid, double si
    {

        int j, np, n0 = n_points / 2;

```

```

double s0, s, y, nu, pas = T / n_points, u;
double *W, *g;
W = new double[n_points + 1];
g = new double[2];
nu = (r - divid) - sigma * sigma / 2 - lambda * (p * lambdap / (lambdap - 1))

// Generate Kou model
pnl_rand_init(generator, 1, 1);

W[0] = 0;
for (j = 1; j < 2 * n0; j += 2)
{
    g[0] = pnl_rand_normal(generator);
    g[1] = pnl_rand_normal(generator);

    W[j] = sigma * g[0] * sqrt(pas) + nu * pas + W[j - 1];
    W[j + 1] = sigma * g[1] * sqrt(pas) + nu * pas + W[j];
}
W[n_points] = sigma * pnl_rand_normal(generator) * sqrt(pas) + nu * pas + W[n_points - 1];
np = pnl_rand_poisson(lambda * T, generator);

s0 = 0;
for (j = 1; j <= np; j++)
{
    u = pnl_rand_uni(generator);

    if (1 - p <= u)
        s0 += -log(1 - (u - 1 + p) / p) / lambdap;
    else
        s0 += log(u / (1 - p)) / lambdam;
}
y = W[n_points] + s0;
s = S0 * exp(y);

*ptvalue = exp(-r * T) * s;

delete [] W;
delete [] g;
return OK;
}

```

```

static int Kou_Mc(double K, double S0, double T, double r, double divid, double lambda, double p, double lambda_dap)
{
    int j, n, np, n0 = n_points / 2;
    double s0, s, y, nu, pas = T / n_points, u;
    double *W, *g;
    W = new double[n_points + 1];
    g = new double[2];
    nu = (r - divid) - sigma * sigma / 2 - lambda * (p * lambda_dap / (lambda_dap - 1));
    double k = log(K / S0);

    pnl_rand_init(generator, 1, n_paths);

    //Put options case
    s = 0;
    n = 0;
    for (int i = 0; i < n_paths; i++)
    {
        W[0] = 0;
        for (j = 1; j < 2 * n0; j += 2)
        {
            g[0] = pnl_rand_normal(generator);
            g[1] = pnl_rand_normal(generator);

            W[j] = sigma * g[0] * sqrt(pas) + nu * pas + W[j - 1];
            W[j + 1] = sigma * g[1] * sqrt(pas) + nu * pas + W[j];
        }
        W[n_points] = sigma * pnl_rand_normal(generator) * sqrt(pas) + nu * pas;
        np = pnl_rand_poisson(lambda * T, generator);

        s0 = 0;
        for (j = 1; j <= np; j++)
        {
            u = pnl_rand_uni(generator);

            if (1 - p <= u)
                s0 += -log(1 - (u - 1 + p) / p) / lambda_dap;
            else
                s0 += log(u / (1 - p)) / lambda_dam;
        }
        y = W[n_points] + s0;
        if (y <= k)
    }
}

```

```

        {
            s += K - S0 * exp(y);
            n++;
        }
    }
    //Put options
    *ptprice = exp(-r * T) * s / n_paths;

    delete [] W;
    delete [] g;
    return OK;
}

int CPPI(double S0, double G, double V0, double T, int n_rebalancing, double m
{
    int i;
    double F = G * exp(-r * T); // Floor at time 0
    double S = S0; // Value of risky asset
    double V = V0; // Initiale value of portfolio
    double C = V - F; // Cushion
    double delta_t = T / n_rebalancing;
    double value = 0.;
    double strike;
    double number_of_puts;
    double cost_of_hedging_with_puts = 0.;
    double cost_of_hedging_with_gap_option = 0.;
    double priceKou = 0.;
    double priceGap = 0.;

    for (i = 0; i < n_rebalancing; i++)
    {
        // Simulate risky asset
        Value_Kou_Mc(S, delta_t, r, divid, sigma, lambda, lambdap, lambdam, p,
                    generator, n_points, &value);

        // Update the floor
        F = F * exp(r * delta_t);
        // Here we can add the ratchet features

        if (C <= 0) // If cushion was already negative or null -> risk-free inve
            C = C * exp(r * delta_t);
    }
}

```

```

else // else the cushion has evolved with the risky asset dynamics.
{
    C = C * (m * value / S - (m - 1.) * exp(r * delta_t));
}

V = C + F;
S = value; // Update

// We can hedge this risk by buying put options.
// We need
number_of_puts = m * C / S;
// puts with strike
strike = (1 - 1. / m) * exp(r * delta_t) * S;
// and maturity the next rebalancing date = (i+1)*delta_t

Kou_Mc(strike,
        S, delta_t, r, divid, sigma,
        lambda, lambdap, lambdam, p,
        1, 100, 100000,
        //int generator, int n_points, long n_paths,
        &priceKou);
cost_of_hedging_with_puts += number_of_puts * priceKou;

// Or we can buy a gap option whose notional is the risky asset
// with strike 1/m

GapApproximateKou(S, delta_t, r, divid,
                  sigma, lambda, lambdap, lambdam, p,
                  1. / m, log(1. / m), 1,
                  &priceGap);

cost_of_hedging_with_gap_option += priceGap;
}

*ptvalue = V;

return OK;
}

int CALC(MC_Icppi_KOU)(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 CPPI(ptMod->S0.Val.V_PDOUBLE, ptOpt->MinimumGuaranteed.Val.V_PDOUBLE,
}

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

#endif //PremiaCurrentVersion

static int MET(Init)(PricingMethod *Met, Option *Opt)
{
    if (Met->init == 0)
    {
        Met->init = 1;
        Met->Par[0].Val.V_ENUM.value = 0;
        Met->Par[0].Val.V_ENUM.members = &PremiaEnumMCRNGs;
        Met->Par[1].Val.V_PINT = 100;
    }
    return OK;
}

PricingMethod MET(MC_Icppi_KOU) =
{
    "MC_iCPPI_KOU",
    { {"RandomGenerator", ENUM, {100}, ALLOW},
      {"Number of discretization steps", LONG, {100}, ALLOW}, {" ", PREMIA_NULLT
    },
    CALC(MC_Icppi_KOU),
    {"Scenario Icppi", DOUBLE, {100}, FORBID}, {" ", PREMIA_NULLTYPE, {0}, FORB
    CHK_OPT(MC_Icppi_KOU),

```

```
        CHK_split,  
        MET(Init)  
    };  
  
}
```