

[Help](#)

```
#include "jump1d_stdg.h"
#include "pnl/pnl_cdf.h"
#include "enums.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2008+2) //The "#els
static int CHK_OPT(MC_Mnif)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(MC_Mnif)(void *Opt, void *Mod, PricingMethod *Met)
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else

static double *tabbrownien, *tabpoisson;

static int ele(int i, int j, int n)
{
    return (i - 1) * (n) + (j);
}

static double put_BS(double S, double r_p, double time, double sigma, double K)
{
    double time_sqrt;
    double d1s;
    double d2s;
    double ds;

    time_sqrt = sqrt(time);

    d1s = (log(S / K) + r_p * time) / (sigma * time_sqrt) + 0.5 * sigma * time_sqr
    d2s = d1s - (sigma * time_sqrt);
    ds = -S * cdf_nor(-d1s) + K * exp(-r_p * time) * cdf_nor(-d2s);

    return ds;
}
```

```

static double put_jump(double S, double time, double ampli, double sigma, double
{
    int MAXN;
    int i;
    double r_i;
    double tau, phi, val;
    double p;
    double log_i;

    MAXN = 50;

    phi = exp(ampli) - 1.;
    tau = time;
    val = r - (lambda * phi);
    p = exp(-lambda * tau) * put_BS(S, val, tau, sigma, K);
    phi = exp(ampli) - 1.;

    log_i = 0;
    for (i = 1; i <= MAXN; i++)
    {
        log_i = log_i + log((double)i);
        r_i = r - lambda * phi + i * ampli / ((double)tau);
        p = p + put_BS(S, r_i, tau, sigma, K) * exp(-lambda * tau + (double)i * lo
    }
    return p;
}

static double PhiBP(double t, double b, double p, double s, double drift, double
{
    double y;
    double x;

    x = s * exp(drift * t + ampli * p + sigma * b);
    if (x < K) y = K - x;
    else y = 0;
    return y;
}

static double S_time(double t, double b, double p, double s, double drift, doubl

```

```

{
    double x;

    x = s * exp(drift * t + ampli * p + sigma * b);
    return (x);
}

static double densite(int i, int j, double lambda, double sigma, double pas, double t)
{
    double log_k, d, val;
    int NMAX = 50;
    int k;

    log_k = 0;
    val = (sigma * tabbrownien[ele(i, j, n)] + ampli * (tabpoisson[ele(i, j, n)])) *
    d = exp(-lambda * (pas * j)) *
        exp(-0.5 * SQR(val))
        * 1.0 / (((double)sqrt(((double)2 * M_PI * j * pas)));
    for (k = 1; k <= NMAX; k++)
    {
        log_k = log_k + log((double)k);
        val = (sigma * tabbrownien[ele(i, j, n)] + ampli * (tabpoisson[ele(i, j, n)])) *
        d = d + exp(-lambda * (pas * j) + k * log(lambda * (pas * j)) - log_k) *
            exp(-0.5 * SQR(val))
            * 1.0 / (((double)sqrt(((double)2 * M_PI * j * pas)));
    }

    return d;
}

static double esp_conditionelle(int droit, int i, int j, int k, double *suivant,
                                int N, int n, double pas, double ampli,
                                double sigma, double lambda, double K)
{
    int iter;
    double numerateur, denominateur1, esp, corrige;
    double alpha, beta, nu;
    double *a, *ap;

    a = malloc((N + 1) * sizeof(double));
    ap = malloc((N + 1) * sizeof(double));

```

```

numérateur = 0;
alpha = 0;
beta = 0;

for (iter = 1; iter <= N; iter++)
{
    a[iter] = -(tabbrownien[ele(iter, j, n)] / (j * pas)
               - (tabbrownien[ele(iter, k, n)] - tabbrownien[ele(iter, j, n)]

    alpha = alpha + suivant[iter] * suivant[iter] * a[iter] * a[iter];

    beta = beta + suivant[iter] * suivant[iter];

}

nu = sqrt(alpha / beta);

for (iter = 1; iter <= N; iter++)
{
    if (tabbrownien[ele(iter, j, n)] <=
        (tabbrownien[ele(i, j, n)] + ampli * (tabpoisson[ele(i, j, n)] - tabpo

    {
        corrige = exp(+nu * (tabbrownien[ele(iter, j, n)] -
                           (tabbrownien[ele(i, j, n)] + ampli *
                           (tabpoisson[ele(i, j, n)] - tabpoisson[ele(iter,
                           * (a[iter] + nu);

        numérateur = numérateur + (suivant[iter] * corrige);
    }
}

denominateur1 = densite(i, j, lambda, sigma, pas, ampli, n);

esp = (numérateur / ((double)N * denominateur1));

if (esp > droit * K)
{
    esp = droit * K;
}

```

```

    if (esp < 0)
    {
        esp = 0;
    }

    free(a);
    free(ap);

    return (esp);
}

```

```

static double *payoff_k(int right, double *option_precedente, int n, int N, double delta)
{
    int it, j, m;
    double *t, *tab3;
    double *ptab, *psuivant;
    double mr, payoff;

    /*Memory allocation*/
    t = malloc((n * N + 1) * sizeof(double));
    tab3 = malloc((N + 1) * sizeof(double));

    ptab = t;
    psuivant = tab3;
    mr = delta * 1.0 / pas;
    m = (int)mr;

    for (j = 1; j <= n - m; j++)
    {
        if (right == 1)
        {
            for (it = 1; it <= N; it++)
                t[ele(it, j, n)] = PhiBP(j * pas, tabbrownien[ele(it, j, n)], tabpoisson[it, j, n]);
        }
        else
        {
            for (it = 1; it <= N; it++)
                tab3[it] = option_precedente[ele(it, j + m, n)] - (right - 1) *
                    put_jump(S_time(
                        (j + m) * pas,

```

```

        tabbrownien[ele(it, j + m, n)],
        tabpoisson[ele(it, j + m, n)],
        s, drift, ampli, sigma
    ),
    T - (pas * (j + m)),
    ampli, sigma, r, lambda, K);

for (it = 1; it <= N; it++)
{
    payoff = PhiBP(j * pas, tabbrownien[ele(it, j, n)], tabpoisson[ele
        + exp(-r * delta) * esp_conditionelle(right, it, j, j + m)
        + exp(-r * delta) *
        (right - 1) *
        put_jump(S_time(
            (j * pas), tabbrownien[ele(it, j, n)],
            tabpoisson[ele(it, j, n)],
            s, drift, ampli, sigma),
            T - (pas * j),
            ampli, sigma, r, lambda, K);

    if (payoff > right * K) payoff = right * K;
    t[ele(it, j, n)] = payoff;
}
}

for (j = 1 + n - m; j <= n; j++)
{
    for (it = 1; it <= N; it++)
        t[ele(it, j, n)] = PhiBP(j * pas, tabbrownien[ele(it, j, n)],
            tabpoisson[ele(it, j, n)], s, drift, ampli, sig

}

/*free(t);*/
free(tab3);

return ptab;
}

```

```

static double *ppd(int right, double *option_precedente, int n, int N, double de

```

```

{
    int it, j;
    double *t, *tab3, *tab4;
    double *ptab;
    double *psuivant, *ptabpayoff;
    double a1, a2, a, a3;
    double simul;

    /*Memory allocation*/
    t = malloc((n * N + 1) * sizeof(double));
    tab3 = malloc((N + 1) * sizeof(double));
    tab4 = malloc((N + 1) * sizeof(double));

    ptab = t;
    psuivant = tab3;

    ptabpayoff = payoff_k(right, option_precedente, n, N, delta, pas, s, drift, am

    for (it = 1; it <= N; it++)
    {
        t[ele(it, n, n)] = ptabpayoff[ele(it, n, n)];

        tab3[it] = t[ele(it, n, n)]
                    - right * PhiBP(n * pas, tabbrownien[ele(it, n, n)], tabpoisson

    }

    for (j = n - 1; j >= 1; j--)
    {

        for (it = 1; it <= N; it++)
        {
            a1 = ptabpayoff[ele(it, j, n)];

            if (j == n - 1)
                a2 = exp(-r * pas) * put_jump(S_time((j * pas), tabbrownien[ele(it,
                    tabpoisson[ele(it, j, n)],
                    s, drift, ampli, sigma),
                    T - (pas * j), ampli, sigma, r, lambda

            else
                a2 = exp(-r * pas) * esp_conditionnelle(right, it, j, j + 1, psuivant
                    + exp(-r * pas) * right * put_jump(S_time(

```

```

        (j * pas),
        tabbrownien[ele(it, j, n)],
        tabpoisson[ele(it, j, n)],
        s, drift, ampli, sigma),

        T - (pas * j),
        ampli, sigma, r, lambda, K
    );

a3 = a1 - a2;

simul = s * exp(drift * (pas * j) + ampli * tabpoisson[ele(it, j, n)])

if (a3 > 0 && a3 < 0.1 && simul > 100)
{
    a = a1;
    a1 = a2;
    a2 = a;
}

if (a3 < 0 && simul < 75)
{
    a2 = a1;
}
if (a3 > 0 && simul > 130)
{
    a1 = a2;
}

if (a1 >= a2) tab4[it] = a1;
else tab4[it] = a2;

t[ele(it, j, n)] = tab4[it];

}
for (it = 1; it <= N; it++)
{
    tab3[it] = tab4[it] - right *
        put_jump(S_time(
            (j * pas),
            tabbrownien[ele(it, j, n)],
            tabpoisson[ele(it, j, n)],

```



```

        s, drift, ampli, sigma),
        T - (pas * j),
        ampli, sigma, r, lambda, K
    );
}

}

/*free(t);*/
free(tab3);
free(tab4);
free(ptabpayoff);

return ptab;
}

static double Phi(double x, double K)
{
    double y;

    if (x < K) y = K - x;
    else y = 0;
    return (y);
}

static double price_swing(double initial, double *pay, int N, int n, double r, d
{
    int i;
    double value, esp, actu;

    esp = 0;
    for (i = 1; i <= N; i++)
        esp = esp + pay[ele(i, 1, n)];

    esp = exp(-r * pas) * esp / N;
    actu = Phi(initial, K);
    if (actu >= esp) value = actu;
    else value = esp;

    return value;
}

```

```

static int MCMnif(double s, NumFunc_1 *p, double T, int dr, double delta, double
                double divid, double sigma, double lambda, double phi, long N,
                int generator, int n, double inc, double confidence, double *p
{
    int i, j;
    double pas, ampli, K, new_lambda, g;
    int simulation_dim = 1;
    int init_mc;
    double *taboption_precedente, *poption_precedente, *value = NULL, price_value
    double drift;

    ampli = log(1 + phi);
    drift = (r - divid - lambda * phi - (sigma * sigma * 0.5));
    pas = T / n;
    K = p->Par[0].Val.V_PDDOUBLE;

    /*Memory allocation*/
    tabbrownien = malloc((n * N + 1) * sizeof(double));
    if (tabbrownien == NULL)
        return MEMORY_ALLOCATION_FAILURE;

    tabpoisson = malloc((n * N + 1) * sizeof(double));
    if (tabpoisson == NULL)
        return MEMORY_ALLOCATION_FAILURE;

    taboption_precedente = malloc((n * N + 1) * sizeof(double));
    if (taboption_precedente == NULL)
        return MEMORY_ALLOCATION_FAILURE;

    /*Simulation of Brownian Motion and Poisson Process*/
    init_mc = pnl_rand_init(generator, simulation_dim, N);
    if (init_mc == OK)
    {

        /*Simulation of the whole trajectory of Brownian Motion and Poisson Proces
        for (i = 1; i <= N; i++)
        {
            g = pnl_rand_normal(generator);

```

```

    tabbrownien[ele(i, 1, n)] = sqrt(pas) * g;
    new_lambda = lambda * pas;
    tabpoisson[ele(i, 1, n)] = pnl_rand_poisson(new_lambda, generator);

    for (j = 2; j <= n; j++)
    {
        g = pnl_rand_normal(generator);
        tabbrownien[ele(i, j, n)] = tabbrownien[ele(i, j - 1, n)] + sqrt(p
        new_lambda = lambda * pas;
        tabpoisson[ele(i, j, n)] = tabpoisson[ele(i, j - 1, n)] + pnl_rand

    }
}

/*Init*/
for (i = 1; i <= N; i++)
    for (j = 1; j <= n; j++)
        tabpoption_precedente[ele(i, j, n)] = 0;

poption_precedente = tabpoption_precedente;

for (i = 1; i <= dr; i++)
{
    value = ppd(i, poption_precedente, n, N, delta, pas, s, drift, ampli,
    free(poption_precedente);
    price_value = price_swing(s, value, N, n, r, pas, K);
    poption_precedente = value;
}

*ptprice = price_value;
free(value);
}
free(tabbrownien);
free(tabpoisson);

return init_mc;
}

int CALC(MC_Mnif)(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 MCMnif(ptMod->S0.Val.V_PDOUBLE,
              ptOpt->PayOff.Val.V_NUMFUNC_1,
              ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
              ptOpt->NbExerciseDate.Val.V_PINT, ptOpt->RefractingPeriod.Val.V_
              r,
              divid,
              ptMod->Sigma.Val.V_PDOUBLE,
              ptMod->Lambda.Val.V_PDOUBLE,
              ptMod->Mean.Val.V_PDOUBLE,
              Met->Par[0].Val.V_LONG,
              Met->Par[1].Val.V_ENUM.value,
              Met->Par[2].Val.V_INT,
              Met->Par[3].Val.V_PDOUBLE,
              Met->Par[4].Val.V_DOUBLE,
              &(Met->Res[0].Val.V_DOUBLE)
              /*,&(Met->Res[1].Val.V_DOUBLE),
              &(Met->Res[2].Val.V_DOUBLE),
              &(Met->Res[3].Val.V_DOUBLE),
              &(Met->Res[4].Val.V_DOUBLE),
              &(Met->Res[5].Val.V_DOUBLE),
              &(Met->Res[6].Val.V_DOUBLE),
              &(Met->Res[7].Val.V_DOUBLE)*/);

}

static int CHK_OPT(MC_Mnif)(void *Opt, void *Mod)
{
    Option *ptOpt = (Option *)Opt;
    TYPEOPT *opt = (TYPEOPT *) (ptOpt->TypeOpt);

    if ((opt->EuOrAm).Val.V_BOOL == AMER)
        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_LONG = 1000;
        Met->Par[1].Val.V_ENUM.value = 0;
        Met->Par[1].Val.V_ENUM.members = &PremiaEnumMCRNGs;
        Met->Par[2].Val.V_INT = 20;
        Met->Par[3].Val.V_PDOUBLE = 0.01;
        Met->Par[4].Val.V_DOUBLE = 0.95;
    }
    return OK;
}

PricingMethod MET(MC_Mnif) =
{
    "MC_Swing_Jump",
    { {"N iterations", LONG, {100}, ALLOW},
      {"RandomGenerator", ENUM, {100}, ALLOW},
      {"TimeStepNumber", INT2, {100}, ALLOW},
      {"Delta Increment Rel (Digit)", PDOUBLE, {100}, ALLOW},
      {"Confidence Value", DOUBLE, {100}, ALLOW},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CALC(MC_Mnif),
    { {"Price", DOUBLE, {100}, FORBID},
      /*{"Delta",DOUBLE,{100},FORBID} ,
        {"Error Price",DOUBLE,{100},FORBID},
        {"Error Delta",DOUBLE,{100},FORBID} ,
        {"Inf Price",DOUBLE,{100},FORBID},
        {"Sup Price",DOUBLE,{100},FORBID} ,
        {"Inf Delta",DOUBLE,{100},FORBID},
        {"Sup Delta",DOUBLE,{100},FORBID} ,*/
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CHK_OPT(MC_Mnif),
    CHK_mc,

```

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