

Help

```
#include "acdpid_std.h"
#include "enums.h"
#include "error_msg.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2014+2) //The "#els
static int CHK_OPT(MC_MultiLevel_Scott)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(MC_Path_ACDP)(void *Opt, void *Mod, PricingMethod *Met)
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else

#if 0
//Function used in the calcuclus of the price of the call with no jumps
static double integranda(double x, void *v)
{
    double y;
    double *p = (double *)v;

    y = SQR(p[0]) * (pow((p[1] + x), 2 * p[2]) - pow(x, 2 * p[2]));
    return pnl_bs_call(p[4], p[5], p[1], p[6], 0, sqrt(y / p[1])) * p[3] * exp(-p[
}
#endif

//Function used in order to find the third jump
static double terzoSalto(double x, void *v)
{
    double *vi = (double *)v;
    double p, q;
    p = 1 - (exp(-vi[0] * x) * (1 + vi[0] * x + 0.5 * x * x * vi[0] * vi[0]));
    q = 1 - (exp(-vi[0] * vi[1]) * (1 + vi[0] * vi[1] + 0.5 * vi[1] * vi[1] * vi[0]
    return ((p / q) - vi[2]);
}

//creates a vector with the jumps times (last is T)
```

```

static void jumpGenerator(double lambda, double sigma, double D, double T, int k
{
    int i = 0;
    PnlVect *Valori;
    double fattcom;
    double prodpar;

    Valori = pnl_vect_create(k + 2);
    for (i = 0; i <= k + 1; i++)
    {
        pnl_vect_set(Valori, i, pnl_rand_uni(generator));
    }
    pnl_vect_set(Tau, 0, log(pnl_vect_get(Valori, 0)) / lambda);
    if (k > 0)
    {
        fattcom = 1.0;
        prodpar = 1.0;
        for (i = 1; i <= k + 1; i++)
        {
            fattcom = fattcom * pnl_vect_get(Valori, i);
        }
        for (i = 1; i <= k; i++)
        {
            prodpar = prodpar * pnl_vect_get(Valori, i);
            pnl_vect_set(Tau, i, T * log(prodpar) / log(fattcom));
        }
    }
    pnl_vect_set(Tau, k + 1, T);
    pnl_vect_free(&Valori);
}

//creates a vector with more than 3 jumps(last is T)
static void jumpGenerator3(double lambda, double sigma, double D, double T, PnlV
{
    int i = 0;
    double x;
    double *tmp;
    double u;
    double t;
    double tol;

```

```

double fattcom;
double prodpar;
PnlVect *Valori;
PnlFunc func;

tmp = malloc(3 * sizeof(double));
Valori = pnl_vect_create(3);
tol = T / 1000.;
pnl_vect_resize(Tau, 1);
x = pnl_rand_uni(generator);
pnl_vect_set(Tau, 0, log(x) / lambda);
u = pnl_rand_uni(generator);
tmp[0] = lambda;
tmp[1] = T;
tmp[2] = u;
func.F = terzoSalto;
func.params = (void *) tmp;
t = pnl_root_brent(&func, 0.0, T , &tol);
pnl_vect_resize(Tau, 4);
pnl_vect_set(Tau, 3, t);

for (i = 0; i <= 2; i++)
{
    pnl_vect_set(Valori, i, pnl_rand_uni(generator));
}
fattcom = 1.0;
prodpar = 1.0;
for (i = 0; i <= 2; i++)
{
    fattcom = fattcom * pnl_vect_get(Valori, i);
}
for (i = 0; i < 2; i++)
{
    prodpar = prodpar * pnl_vect_get(Valori, i);
    pnl_vect_set(Tau, i + 1, t * (log(prodpar) / log(fattcom)));
}
i = 4;
do
{
    x = pnl_rand_uni(generator);
    t = t - log(x) / lambda;

```

```

        if (t < T)
        {
            pnl_vect_resize(Tau, i + 1);
            pnl_vect_set(Tau, i, t);
            i = i + 1;
        }
    }
    while (t < T);
    pnl_vect_resize(Tau, i + 1);
    pnl_vect_set(Tau, i, T);
    pnl_vect_free(&Valori);

    free(tmp);
}

//gives the final time with the jumps as a entry
static double changedTimeGeneratorVec(double sigma, double D, int k, PnlVect *Jumps)
{
    int i;
    double finale;

    i = 0;
    finale = -pow(- pnl_vect_get(Jumps, 0), 2 * D);
    if (k > 0)
    {
        for (i = 1; i <= k; i++)
        {
            finale = finale + pow((pnl_vect_get(Jumps, i) - pnl_vect_get(Jumps, i-1)), 2 * D);
        }
    }
    finale = SQR(sigma) * (finale + pow((pnl_vect_get(Jumps, k + 1) - pnl_vect_get(Jumps, k)), 2 * D));
    return finale;
}

//gives the final time with the number of jumps as entry
static double changedTimeGenerator(double lambda, double sigma, double D, double lambda0)
{
    int i = 0;
    double finale;
    PnlVect *Jumps;

```

```

Jumps = pnl_vect_create(k + 2);

jumpGenerator(lambda, sigma, D, T, k, Jumps, generator);
finale = -pow(-pnl_vect_get(Jumps, 0), 2 * D);
if (k > 0)
{
    for (i = 1; i <= k; i++)
    {
        finale = finale + pow((pnl_vect_get(Jumps, i) - pnl_vect_get(Jumps, i-1)), 2 * D);
    }
}
finale = SQR(sigma) * (finale + pow((pnl_vect_get(Jumps, k + 1) - pnl_vect_get(Jumps, k)), 2 * D));

pnl_vect_free(&Jumps);

return finale;
}

```

```

static void generaIstantiTraiettorie(double lambda, double sigma, double D, int n)
{
    double temp = 0.0;
    double delta;
    double delta1;
    int size = 1;
    double temp1;
    double temp2;
    double temp_value;
    int j = 1;
    int salti, minori;
    double expinv;

    delta = T / Minimal_lenght;
    delta1 = r * delta;
    expinv = 1 / (2 * D);
    salti = 0;
    minori = 0;
    pnl_vect_resize(trj_temp, 1);
    pnl_vect_set(trj_temp, 0, temp);
    pnl_vect_resize(value, 1);
    pnl_vect_set(value, 0, 0.0);
}

```

```

while (temp < T)
{
    //at this moment temp is the last inserted instant in the discretization
    temp2 = temp;
    if (temp + delta > pnl_vect_get(Tau, j)) //in this case we choose as discr
    {
        temp = pnl_vect_get(Tau, j);
        temp_value = (pow((temp - pnl_vect_get(Tau, j - 1)), 2 * D) - pow((tem
        j = j + 1;
        salti = salti + 1;
    }
    else
    {
        temp1 = pnl_vect_get(Tau, j - 1) + pow(((2 * delta1 / (sigma * sigma))
        //temp1 is the instant such that I(t)-I(temp)=r*delta;
        //I choose the minimum between temp1 and temp+delta
        if (temp1 < temp + delta)
        {
            minori = minori + 1;
            temp = temp1;
        }
        else
        {
            temp = temp + delta;
        }
        temp_value = sigma * sigma * (pow(temp - pnl_vect_get(Tau, j - 1), 2 *
    }
    pnl_vect_resize(trj_temp, size + 1);
    pnl_vect_resize(value, size + 1);
    pnl_vect_set(trj_temp, size, temp);
    pnl_vect_set(value, size, temp_value);
    size = size + 1;
}

}

//Generate trajectory
static void genTraiettorie(double lambda, double sigma, double D, int Minimal_le
{
    int lungh;
    int i;

```

```

double x;
PnlVect *esponente;
PnlVect *valori_temp;

valori_temp = pnl_vect_new();
generalIstantiTraiettorie(lambda, sigma, D, Minimal_lenght, T, r, Tau, valori_t,
lungh = tempi->size;
esponente = pnl_vect_create(lungh);
pnl_vect_set(esponente, 0, 0);
for (i = 1; i < lungh; i++)
{
    x = sqrt(pnl_vect_get(valori_temp, i)) * pnl_rand_normal(generator) - (pnl_vect_get(esponente, i));
    pnl_vect_set(esponente, i, x);
}
pnl_vect_resize(traiettoria, lungh);
for (i = 0; i < lungh; i++)
{
    pnl_vect_set(traiettoria, i, S0 * exp(pnl_vect_get(esponente, i)));
}
pnl_vect_free(&esponente);
pnl_vect_free(&valori_temp);
}

int MCPPathACDP(double s0, NumFunc_1 *p, double T, double r, double v, double D,
{
    double strike;
    double *ptdelta = NULL;

    strike = p->Par[0].Val.V_PDDOUBLE;
    pnl_rand_init(generator, 1, N);

    if (D == 0.5)
    {
        pnl_cf_call_bs(s0, strike, T, r, 0, v, ptprice, ptdelta);
    }
    else
    {
        int kMax = 3;
        PnlVect *Taun;
        PnlVect *Traiettorian;
        PnlVect *prezzin;

```

```
int j;
int x;
int k;
double *tmp;
double ciccio;
int primo;
PnlVect *Tenta;
PnlVect *p;
double time;
PnlVect *temp;
PnlVect *meancond;
PnlVect *volcond;
PnlVect *prezzi;
double prezzoPar;
double prezzo;
double sigma1;
double fatcom;
int tentatot;
double sigma;

//Compute sigma
sigma = v * sqrt(pow(lambda, 2 * D - 1) / pnl_tgamma(2 * D + 1));

tmp = malloc(7 * sizeof(double));
tmp[0] = sigma;
tmp[1] = T;
tmp[2] = D;
tmp[3] = lambda;
tmp[4] = s0;
tmp[5] = strike;
tmp[6] = r;

ciccio = N;
primo = ciccio / (40);
fatcom = 0.0;
prezzo = 0.0;
prezzoPar = 0.0;
tentatot = 0;

Tenta = pnl_vect_create_from_zero(4);
p = pnl_vect_create_from_zero(4);
```



```

temp = pnl_vect_create_from_zero(primo);
meancond = pnl_vect_create_from_zero(4);
volcond = pnl_vect_create_from_zero(4);
prezzi = pnl_vect_create_from_zero(4);

//beginning of the stratification

for (k = 0; k < kMax; k++)
{
    x = pnl_sf_fact(k);
    pnl_vect_set(p, k, exp(-lambda * T) * (pow(lambda * T, k)) / x);
    pnl_vect_set(meancond, k, 0.0);
    for (j = 0; j < primo; j++)
    {
        time = changedTimeGenerator(lambda, sigma, D, T, k, generator);
        sigma1 = sqrt(time / T);
        pnl_vect_set(temp, j, pnl_bs_call(s0, strike, T, r, 0, sigma1));
        pnl_vect_set(meancond, k, pnl_vect_get(meancond, k) + pnl_vect_get(temp, j));
    }
    pnl_vect_set(meancond, k, pnl_vect_get(meancond, k) / primo);
    for (j = 0; j < primo; j++)
    {
        pnl_vect_set(volcond, k, pnl_vect_get(volcond, k) + pow(pnl_vect_get(temp, j), 2));
    }
    pnl_vect_set(volcond, k, pnl_vect_get(volcond, k) / primo);
    tentatot = tentatot + primo;
}

k = 3;
x = pnl_sf_fact(k);
pnl_vect_set(p, k, 1 - pnl_vect_get(p, 0) - pnl_vect_get(p, 1) - pnl_vect_get(p, 2));
pnl_vect_set(meancond, k, 0.0);
for (j = 0; j < primo; j++)
{
    Taun = pnl_vect_new();
    jumpGenerator3(lambda, sigma, D, T, Taun, generator);
    time = changedTimeGeneratorVec(sigma, D, Taun->size - 2, Taun, generator);
    sigma1 = sqrt(time / T);
    pnl_vect_set(temp, j, pnl_bs_call(s0, strike, T, r, 0, sigma1));
    pnl_vect_set(meancond, k, pnl_vect_get(meancond, k) + pnl_vect_get(temp, j));
    pnl_vect_free(&Taun);
}

```

```

pnl_vect_set(meancond, k, pnl_vect_get(meancond, k) / primo);
for (j = 0; j < primo; j++)
{
    pnl_vect_set(volcond, k, pnl_vect_get(volcond, k) + pow(pnl_vect_get(tentatot, k), primo));
}
pnl_vect_set(volcond, k, pnl_vect_get(volcond, k) / primo);

tentatot = tentatot + primo;

for (k = 0; k <= kMax; k++)
{
    fatcom = fatcom + ((pnl_vect_get(volcond, k)) * pnl_vect_get(p, k));
}
for (k = 0; k <= kMax; k++)
{
    pnl_vect_set(Tenta, k, (int)MAX(N * 0.9 * (pnl_vect_get(volcond, k)) * fatcom));
    tentatot = tentatot + pnl_vect_get(Tenta, k);
}
//end of stratification

//first second and third layer

for (k = 0; k < kMax ; k++)
{
    if (pnl_vect_get(Tenta, k) != 0)
    {
        for (j = 0; j < pnl_vect_get(Tenta, k); j++)
        {
            Taun = pnl_vect_create(k + 2);
            Traiettorian = pnl_vect_new();
            prezzin = pnl_vect_new();
            jumpGenerator(lambda, sigma, D, T, k, Taun, generator);
            genTraiettoria(lambda, sigma, D, Minimal_lenght, s0, r, T, prezzin);
            prezzoPar = exp(-r * T) * MAX(pnl_vect_get(prezzin, prezzin->size - 1), 0);
            pnl_vect_set(prezzi, k, pnl_vect_get(prezzi, k) + prezzoPar);
            pnl_vect_free(&Taun);
            pnl_vect_free(&Traiettorian);
            pnl_vect_free(&prezzin);
        }
    }
}

```

```

    }
    //fourth layer
    k = 3;
    if (pnl_vect_get(Tenta, k) != 0)
    {
        for (j = 0; j < pnl_vect_get(Tenta, k); j++)
        {
            Taun = pnl_vect_new();
            Traiettorian = pnl_vect_new();
            prezzin = pnl_vect_new();
            jumpGenerator3(lambda, sigma, D, T, Taun, generator);
            genTraiettoria(lambda, sigma, D, Minimal_lenght, s0, r, T, prezzin);
            prezzoPar = exp(-r * T) * MAX(pnl_vect_get(prezzin, prezzin->size
            pnl_vect_set(prezzi, k, pnl_vect_get(prezzi, k) + prezzoPar);
            pnl_vect_free(&Taun);
            pnl_vect_free(&Traiettorian);
            pnl_vect_free(&prezzin);
        }
    }
    prezzo = pnl_vect_get(meancond, 0) * primo + pnl_vect_get(prezzi, 0);
    for (k = 1; k <= kMax; k++)
    {
        prezzo = pnl_vect_get(meancond, k) * primo + pnl_vect_get(prezzi, k) +
    }

    //Call Price
    *ptprice = prezzo / (tentatot);

    pnl_vect_free(&Tenta);
    pnl_vect_free(&p);
    pnl_vect_free(&temp);
    pnl_vect_free(&meancond);
    pnl_vect_free(&volcond);
    pnl_vect_free(&prezzi);
    free(tmp);

}

//Put Case
if ((p->Compute) == &Put)
{

```

```

        *ptprice = *ptprice - s0 + strike * exp(-r * T);
    }

    return OK;
}

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

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

    return MCPPathACDP(ptMod->S0.Val.V_PDOUBLE,
                        ptOpt->PayOff.Val.V_NUMFUNC_1,
                        ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
                        r,
                        ptMod->v.Val.V_PDOUBLE,
                        ptMod->D.Val.V_RGDOUBLE005,
                        ptMod->Lambda.Val.V_PDOUBLE,
                        Met->Par[0].Val.V_ENUM.value,
                        Met->Par[1].Val.V_LONG,
                        Met->Par[2].Val.V_PINT,
                        &(Met->Res[0].Val.V_DOUBLE)
    );
}

static int CHK_OPT(MC_Path_ACDP)(void *Opt, void *Mod)
{
    if ((strcmp(((Option *)Opt)->Name, "CallEuro") == 0) || (strcmp(((Option *)Opt)

        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_LONG = 10000;
        Met->Par[2].Val.V_PINT = 100;
    }

    return OK;
}

PricingMethod MET(MC_Path_ACDP) =
{
    "MC_Path_ACDP",
    { {"RandomGenerator", ENUM, {100}, ALLOW},
      {"N iterations", LONG, {100}, ALLOW},
      {"Minimal Number of Discretizations Steps", PINT, {100}, ALLOW},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CALC(MC_Path_ACDP),
    { {"Price", DOUBLE, {100}, FORBID},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CHK_OPT(MC_Path_ACDP),
    CHK_mc,
    MET(Init)
};

```