Help

```
#include  "acdp1d_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_Direct_ACDP)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else


//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[
}

static double integranda1(double x, void *v)
{
  double y, z, q;
  double *p = (double *)v;

  y = SQR(p[0]) * (pow((p[1] + x), 2 * p[2]) - pow(x, 2 * p[2]));
  pnl_cf_call_bs(p[4], p[5], p[1], p[6], 0, sqrt(y / p[1]), &q, &z);
  return z * p[3] * exp(-p[3] * x);
}

//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 *Ju
{
  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
        }
    }
  finale = SQR(sigma) * (finale + pow((pnl_vect_get(Jumps, k + 1) - pnl_vect_get
  return finale;
}
```

```
//gives the final time with the number of jumps as entry
static double changedTimeGenerator(double lambda, double sigma, double D, double
{
  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
        }
    }
  finale = SQR(sigma) * (finale + pow((pnl_vect_get(Jumps, k + 1) - pnl_vect_get

  pnl_vect_free(&Jumps);

  return finale;
}

int MCDirectACDP(double s0, NumFunc_1  *p, double T, double r, double v, double
{
  double strike;

  strike = p->Par[0].Val.V_PDOUBLE;
  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;
```

```
int j;
int x;
int k;
double *tmp;
double ciccio;
int primo;
PnlVect *Tenta;
PnlVect *p;
double time;
PnlVect *temp;
PnlVect *temp2;
double temp3;
double temp4;
PnlVect *meancond;
PnlVect *meancond2;
PnlVect *volcond;
PnlVect *volcond2;
PnlVect *prezzi;
PnlVect *delte;
double prezzoPar;
double deltaPar;
double prezzo;
double delta;
double sigma1;
double fatcom;
int tentatot;
PnlFunc func;
PnlFunc func1;
double sigma;

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

func.F = integranda;
func1.F = integranda1;

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;
func.params = (void *) tmp;
func1.params = (void *) tmp;

ciccio = N;
primo = ciccio / (40);
fatcom = 0.0;
prezzo = 0.0;
prezzoPar = 0.0;
deltaPar = 0.0;
delta = 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);
temp2 = pnl_vect_create_from_zero(primo);
meancond = pnl_vect_create_from_zero(4);
meancond2 = pnl_vect_create_from_zero(4);
volcond = pnl_vect_create_from_zero(4);
volcond2 = pnl_vect_create_from_zero(4);
prezzi = pnl_vect_create_from_zero(4);
delte = 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_cf_call_bs(s0, strike, T, r, 0, sqrt(time / T), &prezzoPar, &d
        pnl_vect_set(temp, j, prezzoPar);
        pnl_vect_set(temp2, j, deltaPar);
```

```
            pnl_vect_set(meancond, k, pnl_vect_get(meancond, k) + prezzoPar);
            pnl_vect_set(meancond2, k, pnl_vect_get(meancond2, k) + deltaPar);
          }
        pnl_vect_set(meancond, k, pnl_vect_get(meancond, k) / primo);
        pnl_vect_set(meancond2, k, pnl_vect_get(meancond2, k) / primo);
        for (j = 0; j < primo; j++)
          {
            pnl_vect_set(volcond, k, pnl_vect_get(volcond, k) + pow(pnl_vect_g
            pnl_vect_set(volcond2, k, pnl_vect_get(volcond2, k) + pow(pnl_vect
          }
        pnl_vect_set(volcond, k, pnl_vect_get(volcond, k) / primo);
        pnl_vect_set(volcond2, k, pnl_vect_get(volcond2, 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_
    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, generat
        sigma1 = sqrt(time / T);
        pnl_cf_call_bs(s0, strike, T, r, 0, sigma1, &temp3, &temp4);
        pnl_vect_set(temp, j, temp3);
        pnl_vect_set(temp2, j, temp4);
        pnl_vect_set(meancond, k, pnl_vect_get(meancond, k) + pnl_vect_get(tem
        pnl_vect_set(meancond2, k, pnl_vect_get(meancond2, k) + pnl_vect_get(t
        pnl_vect_free(&Taun);

      }
    pnl_vect_set(meancond, k, pnl_vect_get(meancond, k) / primo);
    pnl_vect_set(meancond2, k, pnl_vect_get(meancond2, k) / primo);
    for (j = 0; j < primo; j++)
      {
        pnl_vect_set(volcond, k, pnl_vect_get(volcond, k) + pow(pnl_vect_get(t
        pnl_vect_set(volcond2, k, pnl_vect_get(volcond2, k) + pow(pnl_vect_get
      }
    pnl_vect_set(volcond, k, pnl_vect_get(volcond, k) / primo);
    pnl_vect_set(volcond2, k, pnl_vect_get(volcond2, k) / primo);
```

```
tentatot = tentatot + primo;

for (k = 0; k <= kMax; k++)
  {
    fatcom = fatcom + ((pnl_vect_get(volcond, k) + pnl_vect_get(volcond2,
  }
for (k = 0; k <= kMax; k++)
  {
    pnl_vect_set(Tenta, k, (int)MAX(N * 0.9 * (pnl_vect_get(volcond, k) +
    tentatot = tentatot + pnl_vect_get(Tenta, k);
  }
//end of stratification

//first layer
k = 0;
pnl_vect_set(prezzi, 0, pnl_integration(&func, 0.0, 10.0 / lambda, (int)pn
pnl_vect_set(delte, 0, pnl_integration(&func1, 0.0, 10.0 / lambda, (int)pn
//second and third layer
for (k = 1; k < kMax ; k++)
  {
    if (pnl_vect_get(Tenta, k) != 0)
      {
        for (j = 0; j < pnl_vect_get(Tenta, k); j++)
          {
            time = changedTimeGenerator(lambda, sigma, D, T, k, generator)
            pnl_cf_call_bs(s0, strike, T, r, 0, sqrt(time / T), &prezzoPar
            pnl_vect_set(prezzi, k, pnl_vect_get(prezzi, k) + prezzoPar);
            pnl_vect_set(delte, k, pnl_vect_get(delte, k) + deltaPar);
          }
      }
  }
//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();
        jumpGenerator3(lambda, sigma, D, T, Taun, generator);
        time = changedTimeGeneratorVec(sigma, D, (Taun->size - 2), Taun, g
        pnl_cf_call_bs(s0, strike, T, r, 0, sqrt(time / T), &prezzoPar, &d
```

```
              pnl_vect_set(prezzi, k, pnl_vect_get(prezzi, k) + prezzoPar);
              pnl_vect_set(delte, k, pnl_vect_get(delte, k) + deltaPar);
              pnl_vect_free(&Taun);
            }
        }
      prezzo = pnl_vect_get(meancond, 0) * primo + pnl_vect_get(prezzi, 0) * pnl
      delta = pnl_vect_get(meancond2, 0) + pnl_vect_get(delte, 0) * pnl_vect_get
      for (k = 1; k <= kMax; k++)
        {
          prezzo = pnl_vect_get(meancond, k) * primo + pnl_vect_get(prezzi, k) +
          delta = pnl_vect_get(meancond2, k) + pnl_vect_get(delte, k) + delta;
        }
      prezzo = prezzo / (tentatot);
      delta = delta / (tentatot);
      *ptprice = prezzo;
      *ptdelta = delta;

      pnl_vect_free(&Tenta);
      pnl_vect_free(&p);
      pnl_vect_free(&temp);
      pnl_vect_free(&temp2);
      pnl_vect_free(&meancond);
      pnl_vect_free(&meancond2);
      pnl_vect_free(&volcond);
      pnl_vect_free(&volcond2);
      pnl_vect_free(&prezzi);
      pnl_vect_free(&delte);
      free(tmp);

    }

  //Put Case
  if ((p->Compute) == &Put)
    {
      *ptprice = *ptprice - s0 + strike * exp(-r * T);
    }

  return OK;
}

int CALC(MC_Direct_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 MCDirectACDP(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->Res[0].Val.V_DOUBLE),
                      &(Met->Res[1].Val.V_DOUBLE)
                     );
}

static int CHK_OPT(MC_Direct_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;
    }

  return OK;
```

```
}

PricingMethod MET(MC_Direct_ACDP) =
{
  "MC_Direct_ACDP",
  { {"RandomGenerator", ENUM, {100}, ALLOW},
    {"N iterations", LONG, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_Direct_ACDP),
  { {"Price", DOUBLE, {100}, FORBID}, {"Delta", DOUBLE, {100}, FORBID},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(MC_Direct_ACDP),
  CHK_mc,
  MET(Init)
};
```