Help

```
#include  "dup1d_std.h"
#include "pnl/pnl_cdf.h"
#include "enums.h"

static double sigma3(double t, double S0, int sigma_type)
{
  double a1, a2, b1, b2, h1, h2, sum, sum2, sigma;
  int n, m, i, j;

  n = 1000;
  m = 1000;
  a1 = 0;
  b1 = t;
  a2 = S0 / 10.;
  b2 = 10.*S0;
  h1 = (b1 - a1) / (double)n;
  h2 = (b2 - a2) / (double)m;
  sum = 0;
  sum2 = 0;
  for (i = 1; i <= n - 1; i++)
    {

      sum = sum + premia_local_vol(a1 + (double)i * h1, a2, sigma_type);
      sum = sum + premia_local_vol(a1 + (double)i * h1, b2, sigma_type);
    }
  for (i = 1; i <= m - 1; i++)
    {
      sum = sum + premia_local_vol(a1, a2 + (double)i * h2, sigma_type);
      sum = sum + premia_local_vol(b1, a2 + (double)i * h2, sigma_type);
    }
  for (i = 1; i <= n - 1; i++)
    {
      for (j = 1; j <= m - 1; j++)
        {
          sum2 = sum2 + premia_local_vol(a1 + (double)i * h1, a2 + (double)j * h
        }
    }
  sigma = 1. / ((b1 - a1) * (b2 - a2)) * h1 * h2 * (1. / 4.*(premia_local_vol(a1
  return (sigma);
```

```
}

static int MCDupire(double s, NumFunc_1  *p, double t, double r, double divid, i
{
  int flag;
  long i;
  double mean_price, mean_delta, var_price, var_delta, price_sample_plus, price_
  double sigma;
  int init_mc;
  int simulation_dim = 1;
  double alpha, z_alpha;

  /* double eps=1.0;*/
  double S, W, y, d, Sh, h, b;


  int j, a;
  double price, delta, K;

  /* Increment for Delta*/
  h = 0.001;

  /* Value to construct the confidence interval */
  alpha = (1. - confidence) / 2.;
  z_alpha = pnl_inv_cdfnor(1. - alpha);

  /*Initialisation*/
  flag = 0;
  mean_price = 0.0;
  mean_delta = 0.0;
  var_price = 0.0;
  var_delta = 0.0;
  K = p->Par[0].Val.V_DOUBLE;

  /* Change a Call into a Put to apply the Call-Put parity */
  if ((p->Compute) == &Call)
    {
      (p->Compute) = &Put;
      flag = 1;
    }
```

```
sigma = sigma3(t, s, sigma_type);

/*MC sampling*/
init_mc = pnl_rand_init(generator, simulation_dim, N);

/* Test after initialization for the generator */
if (init_mc == OK)
  {

    d = t / (double)M;

    /* Begin N iterations */
    for (i = 1 ; i <= N ; i++)
      {
        S = log(s);
        Sh = log(s + h);
        a = 1;
        b = 0;
        for (j = 0; j < M; j++)
          {
            /* Simulation of a gaussian variable according to the generator ty
             that is Monte Carlo or Quasi Monte Carlo. */
            y = pnl_rand_normal(generator);

            W = (sqrt(d)) * y;
            S = S + premia_local_vol(a * d, exp(S), sigma_type) * W + (r - div


            Sh = Sh + premia_local_vol(a * d, exp(Sh), sigma_type) * W + (r -
            a = a + 1;
            b = b + W;
          }

        price_sample = (p->Compute)(p->Par, exp(S)) - (p->Compute)(p->Par, exp

        /*Delta*/
        price_sample_plus = (p->Compute)(p->Par, exp(Sh)) -
                            (p->Compute)(p->Par, exp(log(s + h) + sigma * b +

        delta_sample = (price_sample_plus - price_sample) / (h);
        /*Sum*/
```

```
        mean_price += price_sample;

        mean_delta += delta_sample;

        /*Sum of squares*/
        var_price += SQR(price_sample);
        var_delta += SQR(delta_sample);
      }
    /* End N iterations */


    /* Price */
    pnl_cf_put_bs(s, K, t, r, divid, sigma, &price, &delta);
    /* reduction variance method  */
    *ptprice = exp(-r * t) * (mean_price / (double) N) + price;
    *pterror_price = sqrt(fabs(exp(-2.0 * r * t) * var_price / (double)N - SQR

    /*Delta*/
    *ptdelta = (exp(-r * t) * mean_delta / (double) N + delta);
    *pterror_delta = sqrt(fabs(exp(-2.0 * r * t) * (var_delta / (double)N - SQ

    /* Call Price and Delta with the Call Put Parity */
    if (flag == 1)
      {
        *ptprice += s * exp(-divid * t) - p->Par[0].Val.V_DOUBLE * exp(-r * t)
        *ptdelta += exp(-divid * t);
        (p->Compute) = &Call;
        flag = 0;
      }
    /* Price Confidence Interval */
    *inf_price = *ptprice - z_alpha * (*pterror_price);
    *sup_price = *ptprice + z_alpha * (*pterror_price);

    /* Delta Confidence Interval */
    *inf_delta = *ptdelta - z_alpha * (*pterror_delta);
    *sup_delta = *ptdelta + z_alpha * (*pterror_delta);
  }
  return init_mc;
}
```

```
int CALC(MC_Dupire)(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 MCDupire(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_INT,
                  Met->Par[0].Val.V_LONG, Met->Par[1].Val.V_INT,
                  Met->Par[2].Val.V_ENUM.value,
                  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_Dupire)(void *Opt, void *Mod)
{
  if ((strcmp(((Option *)Opt)->Name, "CallEuro") == 0) || (strcmp(((Option *)Opt
    return OK;

  return WRONG;
}


static int MET(Init)(PricingMethod *Met, Option *Opt)
{
  int type_generator;
```

```
  if (Met->init == 0)
    {
      Met->init = 1;

      Met->Par[0].Val.V_LONG = 10000;
      Met->Par[1].Val.V_INT = 1000;
      Met->Par[2].Val.V_ENUM.value = 0;
      Met->Par[2].Val.V_ENUM.members = &PremiaEnumMCRNGs;
      Met->Par[3].Val.V_PDOUBLE = 0.01;
      Met->Par[4].Val.V_DOUBLE = 0.95;


    }

  type_generator = Met->Par[2].Val.V_ENUM.value;


  if (pnl_rand_or_quasi(type_generator) == PNL_QMC)
    {
      Met->Res[2].Viter = IRRELEVANT;
      Met->Res[3].Viter = IRRELEVANT;
      Met->Res[4].Viter = IRRELEVANT;
      Met->Res[5].Viter = IRRELEVANT;
      Met->Res[6].Viter = IRRELEVANT;
      Met->Res[7].Viter = IRRELEVANT;

    }
  else
    {
      Met->Res[2].Viter = ALLOW;
      Met->Res[3].Viter = ALLOW;
      Met->Res[4].Viter = ALLOW;
      Met->Res[5].Viter = ALLOW;
      Met->Res[6].Viter = ALLOW;
      Met->Res[7].Viter = ALLOW;
    }
  return OK;
}


PricingMethod MET(MC_Dupire) =
```

```
{
  "MC_Dupire",
  { {"N iterations", LONG, {100}, ALLOW}, {"TimeStepNumber", LONG, {100}, ALLOW}
    {"RandomGenerator (Quasi Random not supported)", ENUM, {100}, ALLOW},
    {"Delta Increment  Rel (Digit)", PDOUBLE, {100}, ALLOW},
    {"Confidence Value", DOUBLE, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_Dupire),
  { {"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_Dupire),
  CHK_mc,
  MET(Init)
};
```