Help

```
#include  "
href../../../mod/dup1d/dup1d_std/dup1d_std_h_src.pdfdup1d_std.h"
#include "pnl/pnl_cdf.h"
#include "
href../../../common/enums_h_src.pdfenums.h"

/********************** Code of the core (solver) functions ******************
/*****************************************************************************

static double Payoff(int payoff_type, double payoff_param1, double payoff_param2
{
if (payoff_type==0) //vanilla call when the input is X_t=e^(-mu*T)*S_T
{
// In that case, payoff_param1 = strike, payoff_param2 = mu*T.
return( fmax(0,exp(payoff_param2)*x-payoff_param1) );
}
else //vanilla put when the input is X_t=e^(-mu*T)*S_T
{
// In that case, payoff_param1 = strike, payoff_param2 = mu*T.
return( fmax(0,payoff_param1-exp(payoff_param2)*x) );
};

}

double DiffCoeff(int diffcoeff_type, double diffcoeff_param, double t, double x)
{
    if (diffcoeff_type==0)  // sigma(t,s) = s*lv(t,s) and lv(t,s)=15/s ... so in
{
  return(15.0*exp(-diffcoeff_param*t));
}
else  // sigma(t,s) = s*lv(t,s) and lv(t,s) = 0.01 + 0.01*t + 0.1*e^(-s/100). Th
{
return(  x * (0.01 + 0.01*t + 0.1*exp(-exp(diffcoeff_param*t)*x/100.0)) );
}
}

static double DerivativeOfDiffCoeff(int diffcoeff_type, double diffcoeff_param,
{
 if (diffcoeff_type==0)  // sigma(t,s) = s*lv(t,s) and lv(t,s)=15/s ... so in th
```

```
  {
return(0.0);
}
else  // sigma(t,s) = s*lv(t,s) and lv(t,s) = 0.01 + 0.01*t + 0.1*e^(-s/100). Th
{
return(  1 * (0.01 + 0.01*t + 0.1*exp(-exp(diffcoeff_param*t)*x/100.0)) +  x * (
}
}


static double MCSampler_Driftless(double beta, double T, double x0, double diffc
{
/* Declarations of the variables used */
double Psi, Tim1, Ti, Tip1, X_Tim1, X_Ti, X_Tip1;
double tau, DeltaTip1, DeltaWip1, MW, Weight, WeightedPayoff;
double Sigma_Ti, Sigma_x_Ti,c1,c2,antiX_Tip1,antiMW,antiWeight,antiWeightedPayof


/* Initialization of the variables (for safety in some cases, for the algorithm

Psi = 1;      // to be updated as time advances from 0 to T


Tim1 = 0;   //previous time
Ti   = 0;   //current time
Tip1 = 0;   //next time

X_Tim1 = x0;//location of X at time Tim1
X_Ti   = x0;//location of X at time Ti
X_Tip1 = x0;//location of X at time Tip1

tau               = 0;
DeltaTip1         = Tip1-Ti ;
DeltaWip1         = 0; //increments of the BM
MW                = 1; //Mallivin weight in the weight to update Psi
Weight            = 1; //weight to update Psi
WeightedPayoff    = 1; //weight to finalize Psi

Sigma_Ti          = 0;
Sigma_x_Ti        = 0;
c1                = 0;
```

```
c2                = 0;
antiX_Tip1        = 0;
antiMW            = 0;
antiWeight        = 0;
antiWeightedPayoff= 0;
Sigma_Tim1        = 0;
Sigma_x_Tim1      = 0;
TildeSigma        = 0;
TildeA            = 0;
A                 = 0;


/* * * * * Handling the first interval * * * * */

// Computing the next time Tip1
tau = pnl_rand_exp(beta,generator);
    Tip1 = fmin(Ti+tau,T);    // next switching ... or the time horizon.
    DeltaTip1 = Tip1-Ti ;

    // Compute X_Tip1 (finer scheme)
    DeltaWip1 = sqrt(DeltaTip1)*pnl_rand_normal(generator);
    Sigma_Ti  = DiffCoeff(diffcoeff_type,diffcoeff_param,Ti,X_Ti);
    Sigma_x_Ti= DerivativeOfDiffCoeff(diffcoeff_type,diffcoeff_param,Ti,X_Ti);
    c1 = Sigma_Ti - Sigma_x_Ti * X_Ti;
    if (c2==0)
    {
        X_Tip1 = X_Ti + Sigma_Ti * DeltaWip1;
    }
    else
    {
        X_Tip1 = -c1/c2 + (c1/c2 + X_Ti) * exp( c2*DeltaWip1 - pow(c2,2)/2 * Del
    }

    // Finalize or just update Psi // Note : there's no weight to compute, as we
    MW = 1;
    Weight = 1*MW;
    if (Tip1<T) // something new happened, update Psi with the weight
    {
        Psi = Weight * Psi ; //completely formal, since Weight=1 and Psi=1 at th
    }
    else // Tip1==T, compute final value of Psi
```

```
    {
        WeightedPayoff = exp(beta*T) * (Payoff(payoff_type,payoff_param1,payoff_

        //computing the antithetic quantities for the last interval [Ti,Tip1=T]
        if (c2==0)
        {
            antiX_Tip1 = X_Ti + Sigma_Ti * (-1)*DeltaWip1;
     }
        else
        {
            antiX_Tip1 = -c1/c2 + (c1/c2 + X_Ti) * exp( c2*(-1)*DeltaWip1 - pow(
        }
        antiMW = 1;
        antiWeight = 1*antiMW ;
        antiWeightedPayoff = exp(beta*T) * (Payoff(payoff_type,payoff_param1,pay

        //computing the final Psi
        Psi = (WeightedPayoff*Weight + antiWeightedPayoff*antiWeight)/2 * Psi; /
}

    // Updating the indexing for the next step // Tim1 and X_Tim1 are left at th
    Tim1   = Ti; // has no effect since they were equal initially
    X_Tim1 = X_Ti; // has no effect since they were equal initially
    Ti = Tip1;
    X_Ti = X_Tip1;


/* * * * * Handling the subsequent intervals * * * * */
while(Ti<T)
{
// Computing the next time Tip1
  tau = pnl_rand_exp(beta,generator);
Tip1 = fmin(Ti+tau,T);    // next switching ... or the time horizon.
DeltaTip1 = Tip1-Ti ;

// Compute X_Tip1 (finer scheme)
DeltaWip1 = sqrt(DeltaTip1)*pnl_rand_normal(generator); //scalar Brownian increm
Sigma_Ti  = DiffCoeff(diffcoeff_type,diffcoeff_param,Ti,X_Ti);
Sigma_x_Ti= DerivativeOfDiffCoeff(diffcoeff_type,diffcoeff_param,Ti,X_Ti);
c1 = Sigma_Ti - Sigma_x_Ti * X_Ti;
c2 = Sigma_x_Ti;
```

```
if (c2==0)
{
X_Tip1 = X_Ti + Sigma_Ti * DeltaWip1;
}
else
{
X_Tip1 = -c1/c2 + (c1/c2 + X_Ti) * exp( c2*DeltaWip1 - pow(c2,2)/2 * DeltaTip1 )
}

// Finalize or just update Psi
MW = -Sigma_x_Ti * DeltaWip1/DeltaTip1 + (pow(DeltaWip1,2)-DeltaTip1)/pow(DeltaT
    Sigma_Tim1 = DiffCoeff(diffcoeff_type,diffcoeff_param,Tim1,X_Tim1);
    Sigma_x_Tim1 = DerivativeOfDiffCoeff(diffcoeff_type,diffcoeff_param,Tim1,X_T
    TildeSigma = Sigma_Tim1 + Sigma_x_Tim1 * (X_Ti-X_Tim1);
    TildeA = pow(TildeSigma,2);
    A = pow(Sigma_Ti,2) ;
    Weight = (1/beta) * (A-TildeA)/(2*A) * MW ;
if (Tip1<T) // something new happened, update Psi with the weight
{
Psi = Weight * Psi ;
}
else // Tip1==T, compute final value of Psi
{
WeightedPayoff = exp(beta*T) * (Payoff(payoff_type,payoff_param1,payoff_param2,X

//computing the antithetic quantities for the last interval [Ti,Tip1=T]
if (c2==0)
{
antiX_Tip1 = X_Ti + Sigma_Ti * (-1)*DeltaWip1;
}
else
{
antiX_Tip1 = -c1/c2 + (c1/c2 + X_Ti) * exp( c2*(-1)*DeltaWip1 - pow(c2,2)/2 * De
}
antiMW = -Sigma_x_Ti * (-1)*DeltaWip1/DeltaTip1 + (pow(-DeltaWip1,2)-DeltaTip1)/
antiWeight = (1/beta) * (A-TildeA)/(2*A) * antiMW ;
antiWeightedPayoff = exp(beta*T) * (Payoff(payoff_type,payoff_param1,payoff_para

//computing the final Psi
        Psi = (WeightedPayoff*Weight + antiWeightedPayoff*antiWeight)/2 * Psi;
}
```

```
// Updating the indexing for the next step
Tim1   = Ti;
X_Tim1 = X_Ti;
Ti = Tip1;
X_Ti = X_Tip1;
}

return(Psi);
}

static int MCUnbiasedDupire(double x0, NumFunc_1  *p, double T, double r, double
{
  double discount_factor, diffcoeff_param;
  int i;
double Psi;
double SumOfPsis;
double SumOfPsiSquares;
double EstimatedMean;
double EstimatedVariance;
double Error;
double K;
double payoff_param1,payoff_param2;
int flag;

K=p->Par[0].Val.V_PDOUBLE;
 if ((p->Compute) == &Call)
   flag=0;
 else flag=1;

  discount_factor = exp(-r*T);
  diffcoeff_param = r-divid;


payoff_param1 =K;
payoff_param2 = diffcoeff_param*T;

  SumOfPsis=0;
SumOfPsiSquares=0;

for (i=1;i<=M;i=i+1)
```

```
{
  Psi = MCSampler_Driftless(beta,T,x0,sigma_type,diffcoeff_param,flag,payoff_par
SumOfPsis = SumOfPsis + Psi;
SumOfPsiSquares = SumOfPsiSquares + Psi*Psi;
}

EstimatedMean = discount_factor * 1.0/(double)M * SumOfPsis;
EstimatedVariance = pow(discount_factor,2.) * ( (1.0/(double)(M-1)) * SumOfPsiSq
Error = sqrt(EstimatedVariance/(double)M);

*ptprice = EstimatedMean;
*pterror_price = Error;

  return OK;
}


int CALC(MC_UnbiasedDupire)(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 MCUnbiasedDupire(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_PDOUBLE,
                  Met->Par[2].Val.V_ENUM.value,
                  &(Met->Res[0].Val.V_DOUBLE),
                  &(Met->Res[1].Val.V_DOUBLE));
}

static int CHK_OPT(MC_UnbiasedDupire)(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 = 1000000;
  Met->Par[1].Val.V_DOUBLE = 0.25;
      Met->Par[2].Val.V_ENUM.value = 0;
      Met->Par[2].Val.V_ENUM.members = &PremiaEnumMCRNGs;
    }

  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_UnbiasedDupire) =
{
  "MC_UnbiasedDupire",
  { {"N iterations", LONG, {100}, ALLOW}, {"Beta", PDOUBLE, {100}, ALLOW},
    {"RandomGenerator (Quasi Random not supported)", ENUM, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_UnbiasedDupire),
  { {"Price", DOUBLE, {100}, FORBID},
    {"Error Price", DOUBLE, {100}, FORBID},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(MC_UnbiasedDupire),
  CHK_mc,
  MET(Init)
};
```