Help

```
extern "C" {
#include  "mer1d_stda.h"
#include "enums.h"
#include "pnl/pnl_finance.h"
}
#include "math/levy.h"
#include "math/fft.h"

extern "C" {

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

  static int Ap_Gap_Merton(double S0, NumFunc_1  *p, int numberperiod, double al
  {
    /* Declarations */
    // Integration parameters
    int Nlimit = 2048; // Number of integral discretization steps
    double logstrikestep;
    double k0; // log-moneyness strike
    double h; // Integral discretization step
    double A; // Integration domain is (-A/2,A/2)
    double vn;
    double weight; // Simpson's rule weights

    // Black-Scholes formula
    double sigmaBS;
    double d1;
    double d2;
    double CallBS;
    double DeltaBS;
```

```
// Array and variables for integration
std::vector<double> x(Nlimit), x_img(Nlimit), y(Nlimit), y_img(Nlimit), z(Nl
complex<double> dphi;
complex<double> dzeta;

double price;
double delta;
double cdf;
int n;
double strike;
double betagap, deltagap;

strike = p->Par[0].Val.V_DOUBLE;
if ((S0 < strike))
  {
    return GAP_SHOULD_BE_NEGATIVE;
  }
deltagap = T / (double)numberperiod;
betagap = log(alpha_gap);


/* Compute simultaneously the put part and the CDF. */

S0 *= exp(-divid * deltagap); // Taking account of dividends

Nlimit =  2048;  // Number of integral discretization steps
logstrikestep = 0.01;
k0 = log(strike / S0);
h = 2 * M_PI / Nlimit / logstrikestep; // Integral discretization step
A = (Nlimit - 1) * h; // Integration domain is (-A/2,A/2)

// Black-Scholes formula
sigmaBS = 0.2;
d1 = (log(S0 / strike) + (r + sigmaBS * sigmaBS / 2) * deltagap) / sigmaBS /
d2 = d1 - sigmaBS * sqrt(deltagap);
CallBS = S0 * normCDF(d1) - strike * exp(-r * deltagap) * normCDF(d2);
DeltaBS = normCDF(d1);

// Construction of the model
Merton_measure measure(mu, deltaVol, lambda, sigma, 0.1);
```

```
vn = -A / 2;
weight = 1. / 3; //Simpson's rule weights

dphi = -exp(-I * vn * betagap) * (measure.cf(deltagap, vn)
                                  - exp(-sigmaBS * sigmaBS / 2.*vn * vn)) /
dzeta = exp(I * vn * (r * deltagap - k0)) * (measure.cf(deltagap, vn - I)
        - exp(-deltagap * sigmaBS * sigmaBS / 2.*(vn - I) * vn)) / (I * vn);
// cdf
x[0] = weight * real(dphi);
x_img[0] = weight * imag(dphi);
// price
y[0] = weight * real(dzeta / (1. + I * vn));
y_img[0] = weight * imag(dzeta / (1. + I * vn));
// delta
z[0] = weight * real(dzeta);
z_img[0] = weight * imag(dzeta);

for (n = 1; n < Nlimit - 1; n++)
  {
    vn += h;
    weight = (weight < 1) ? 4. / 3 : 2. / 3; //Simpson's rule weights

    dphi = -exp(-I * vn * betagap) * (measure.cf(deltagap, vn)
                                      - exp(-sigmaBS * sigmaBS / 2.*vn * vn)
    dzeta = exp(I * vn * (r * deltagap - k0)) * (measure.cf(deltagap, vn - I
            - exp(-deltagap * sigmaBS * sigmaBS / 2.*(vn - I) * vn)) / (I *
    // cdf
    x[n] = weight * real(dphi);
    x_img[n] = weight * imag(dphi);
    // price
    y[n] = weight * real(dzeta / (1. + I * vn));
    y_img[n] = weight * imag(dzeta / (1. + I * vn));
    // delta
    z[n] = weight * real(dzeta);
    z_img[n] = weight * imag(dzeta);
  }
vn += h;
weight = 2. / 3; //Simpson's rule weights

dphi = -exp(-I * vn * betagap) * (measure.cf(deltagap, vn)
                                  - exp(-sigmaBS * sigmaBS / 2.*vn * vn)) /
```

```
  dzeta = exp(I * vn * (r * deltagap - k0)) * (measure.cf(deltagap, vn - I)
          - exp(-deltagap * sigmaBS * sigmaBS / 2.*(vn - I) * vn)) / (I * vn);
  // cdf
  x[Nlimit - 1] = weight * real(dphi);
  x_img[Nlimit - 1] = weight * imag(dphi);
  // price
  y[Nlimit - 1] = weight * real(dzeta / (1. + I * vn));
  y_img[Nlimit - 1] = weight * imag(dzeta / (1. + I * vn));
  // delta
  z[Nlimit - 1] = weight * real(dzeta);
  z_img[Nlimit - 1] = weight * imag(dzeta);

  fft1d(&x[0], &x_img[0], Nlimit, -1);
  fft1d(&y[0], &y_img[0], Nlimit, -1);
  fft1d(&z[0], &z_img[0], Nlimit, -1);

  // Compute call
  price = CallBS + S0 * A / 2. / M_PI / (Nlimit - 1) * y[0];
  delta = exp(-divid * deltagap) * (DeltaBS + A / 2. / M_PI / (Nlimit - 1) * z
  // Compute Put via parity
  price = price - S0 + strike * exp(-r * deltagap);
  delta = delta - exp(-divid * deltagap);
  // Compute CDF
  cdf = normCDF(betagap) + A / 2. / M_PI / (Nlimit - 1) * x[0];

  /* Compute the gap option price. */
  *ptprice = exp(-r * deltagap) * price * (1. - exp(-r * T) *
             std::pow(1. - cdf, numberperiod)) / (1. - exp(-r * deltagap) * (1
  *ptdelta = exp(-r * deltagap) * delta * (1. - exp(-r * T) *
             std::pow(1. - cdf, numberperiod)) / (1. - exp(-r * deltagap) * (1

  return OK;
}

int CALC(AP_GAP_MERTON)(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 Ap_Gap_Merton(ptMod->S0.Val.V_PDOUBLE, ptOpt->PayOff.Val.V_NUMFUNC_1,
  }

  static int CHK_OPT(AP_GAP_MERTON)(void *Opt, void *Mod)
  {
    if ((strcmp(((Option *)Opt)->Name, "Gap") == 0))
      return OK;
    return WRONG;
  }

#endif //PremiaCurrentVersion

  static int MET(Init)(PricingMethod *Met, Option *Opt)
  {
    if (Met->init == 0)
      {
        Met->init = 1;
      }
    return OK;
  }

  PricingMethod MET(AP_GAP_MERTON) =
  {
    "AP_GAP_MERTON",
    {{" ", PREMIA_NULLTYPE, {0}, FORBID}},
    CALC(AP_GAP_MERTON),
    {{"Price", DOUBLE, {100}, FORBID}, {"Delta", DOUBLE, {100}, FORBID}, {" ", P
    CHK_OPT(AP_GAP_MERTON),
    CHK_split,
    MET(Init)
  };

}
```