Help

```
/*******************************************************/
//Functions for Broadie-Kaya and Smith Exact simulation in
//the Heston model
/*******************************************************/

#include <
href../../common/math/cdo/cdo_math_h_src.pdfmath.h>
#include <stdio.h>

#include "pnl/pnl_root.h"
#include "
href../../common/math/ESM_func_h_src.pdfESM_func.h"
#include "pnl/pnl_complex.h"
#include "pnl/pnl_specfun.h"
#include "pnl/pnl_mathtools.h"

struct cumulative_function_params
{
  double h;
  int N;
  double *val;
  double u;
};

static double f, g_noV, derivlnf, deriv2lnf, derivg_noV , deriv2g_noV;
static double nu, SK1, SK2, SK3;

// the models constants required everytime we need th compute the characteristic

/*
** in this function we will compute f(K)=0.5*K/sinh(0.5*K*delta), and g(K)= K/ta
** and the following quantities derivlnf=f'(K)/f(K), deriv2lnf = f"(K)/f(K), der
** deriv2g_noV=g"(K). we don't bother to compute the approximation of these quan
** K*delta <<< 1, because for our values of K and delta, we'll never get singula
*/

void ESM_update_const_char(double Kappa, double sigma, double delta, double d)
{
```

```
  double tanhk;
  double sinh2k;

  tanhk = 1. / tanh(0.5 * Kappa * delta);
  sinh2k = 1. / pow(sinh(0.5 * Kappa * delta), 2.);
  f = 0.5 * Kappa / sinh(0.5 * Kappa * delta);
  derivlnf = 1. / Kappa - 0.5 * delta * tanhk;
  deriv2lnf = -1. / (Kappa * Kappa) + pow(0.5 * delta, 2.) * sinh2k + pow(derivl

  g_noV = Kappa * tanhk;
  derivg_noV = tanhk - 0.5 * delta * Kappa * sinh2k;
  deriv2g_noV = - 1.*delta * sinh2k + 2.*Kappa * pow(0.5 * delta, 2.) * sinh2k *
  nu = 0.5 * d - 1.;
  SK1 = sigma * sigma / Kappa;
  SK2 = SK1 * SK1;
  SK3 = SK2 / Kappa;
}


// |x| <= |v|, CF1_ik converges rapidly
// |x| > |v|, CF1_ik needs O(|x|) iterations to converge

//Gautschi (Euler) equivalent series was proposed by gerard
//Jungman "GSL", and I've implemented another methof
// modified Lentz's method, see
// Lentz, Applied Optics, vol 15, 668 (1976)
/*
 * using Gautschi (Euler) equivalent series.
 */

static int
bessel_I_CF1_ser_real(const double x, double *ratio)
{
  const int maxk = 20000;
  double tk   = 1.0;
  double sum  = 1.0;
  double rhok = 0.0;
  int k;
  double ak;
```

```
  for (k = 1; k < maxk; k++)
    {
      ak = 0.25 * (x / (nu + k)) * x / (nu + k + 1.0);
      rhok = -ak * (1.0 + rhok) / (1.0 + ak * (1.0 + rhok));
      tk  *= rhok;
      sum += tk;
      if (fabs(tk / sum) < DBL_EPSILON) break;
    }

  *ratio = x / (2.0 * (nu + 1.0)) * sum;

  return 0;
}

/*
 * this function compute the ratio Inu'(x)/Inu(x) & Inu"(x)/Inu(x)
 * x!=0;
 */

static int
bessel_Inu_real_ratios(double x, double *ratio1, double *ratio2)
{
  double ratio;
  int stat_I;

  stat_I =  bessel_I_CF1_ser_real(x, &ratio);
  *ratio1 = nu / x + ratio;
  *ratio2 = 1. + (nu / x) * (nu / x) - nu / (x * x) - ratio / x;
  return stat_I;
}

/*
 * compute the mean and the variance by derivating the characteristic functions!
 */

void Moments_ESM(double Vs, double Vt, double Kappa, double sigma, double delta,
                 double d, double *mean, double *variance)
{
  double b1, b2;
  double V_V;
  double VV;
```

```
  double derivg;
  double deriv2g;
  double phiR;
  double deriv_phiR, derivh, deriv2h, mean2;

  V_V = (Vs + Vt) / (sigma * sigma);
  VV = sqrt(Vs * Vt) / (sigma * sigma);
  derivg = - V_V * derivg_noV;
  deriv2g = - V_V * deriv2g_noV + derivg * derivg;
  phiR = 4.*VV * f;
  bessel_Inu_real_ratios(phiR, &b1, &b2);
  deriv_phiR = phiR * derivlnf;
  derivh = deriv_phiR * b1;
  deriv2h = phiR * deriv2lnf * b1 + pow(deriv_phiR, 2.) * b2;

  mean2 = - SK3 * (derivlnf + derivg + derivh) + SK2 *
          (deriv2lnf + deriv2g + deriv2h + 2.*(derivlnf * derivg + derivlnf * de

  *mean = -SK1 * (derivlnf + derivg + derivh);
  *variance =  mean2 - pow(*mean, 2.);
}

/*
 * for a given Vt and Vs, we compute all the values required in order to compute
 */

void values_all_ESM(int M, double Vs, double Vt, double Kappa, double sigma, dou
                    double d, double epsilon, double  h, int *N, double *values)
{

  double a;
  double module;
  int sign_arg;
  dcomplex y, g, gd;

  double V_V;
  double VV;
  double phiR;
  double bessel_0;
  int j, m, signe;
  dcomplex Phi, Phi1, Phi2, phiC, besselC, Phi3, char_func;
```

```
V_V = (Vt + Vs) / (sigma * sigma);
VV = sqrt(Vt * Vs) / (sigma * sigma);
phiR = 4.*VV * f;
j = 0;
m = 0;
signe = 1;
bessel_0 = pnl_bessel_i_scaled(nu, phiR);
do
  {

    a = h * (j + 1);
    y = Complex(Kappa * Kappa, -2.*sigma * sigma * a);
    g = Csqrt(y);
    gd = RCmul(0.5 * delta, g);

    Phi = Cdiv(g, Csinh(gd));
    Phi = RCmul(0.5, Phi);
    Phi1 = CRdiv(Phi, f);


    Phi2 = Cdiv(g, Ctanh(gd));
    Phi2 = Cminus(Phi2);
    Phi2 = CRadd(Phi2, g_noV);
    Phi2 = RCmul(V_V, Phi2);
    Phi2 = Cexp(Phi2);

    phiC = RCmul(4.*VV, Phi);

    /*
     * continuite de la fonction de bessel, on determine l'argument non princi
     * Arg(phiC) is increasing from 0 when a=0 to +infinity when a is infinite
     * we need to keep track on arg(phiC) and change the branch every time we
     * pass from a positive argument to a negative argument by substracting 2*
     */
    sign_arg = (Carg(phiC) > 0) ? 1 : -1;
    if (sign_arg - signe == -2) m++; // change the branch
    signe = sign_arg;

    besselC = pnl_complex_bessel_i_scaled(nu, phiC);
```

```
      besselC = Cmul(besselC, Complex_polar(1., -2 * m * M_PI * nu)); // analyti

      Phi3 = CRdiv(besselC, bessel_0);
      Phi3 = RCmul(exp(Creal(phiC) - phiR), Phi3); // the non scaled versions

      char_func = Cmul(Cmul(Phi1, Phi2) , Phi3);

      values[j] = Creal(char_func) / (j + 1);
      module = Cabs(char_func) / (j + 1);
      j++;
    }
  while (module > M_PI * epsilon / 2 && j < M);
  *N = j - 1;
}



/*
 * we gonna find all the values required to compute all the cumulative
 * functions and their invrerses, in the AESM model(Smith algorithm). z is of th
 * z_i=i*Vmax/N_z_grid  i=1...NS
 */
void values_all_AESM(int M, double Vmax, int NS , double Kappa, double sigma, do
                     double *mean, double *variance, double *h, int *N, double *
{
  int i;
  double z_i;

  for (i = 0; i < NS ; i++)
    {

      //NS for number of slices
      z_i = Vmax * (i + 1) / NS; //so z_i goes from Vmax/NS to Vmax;
      Moments_ESM(z_i, z_i, Kappa, sigma, delta, d, &mean[i], &variance[i]);
      h[i] = M_PI / (mean[i] + 5.*sqrt(variance[i]));
      values_all_ESM(M, z_i, z_i, Kappa, sigma, delta, d, epsilon, h[i], &N[i],

    }
}
```

```
static double cumulative_function_ESM(double x, double h, int N, double *val)
{
  int j;
  double sum;

  sum = h * x / 2.;
  for (j = 0; j < N + 1 ; j++)
    {
      sum += sin(h * (j + 1) * x) * val[j];
    }
  return 2.*sum / M_PI;
}


static double cumulative_function_ESM2(double x, void *params)
{
  struct cumulative_function_params *p = (struct cumulative_function_params *) p
  double h;
  int N;
  double u;

  double *val = p->val;

  h = p->h;
  N = p->N;
  u = p->u;

  return cumulative_function_ESM(x, h, N, val) - u;
}

double inverse_ESM(double u, double h, int N, double *val)
{

  PnlFunc F;
  double x_lo, x_hi; // x_hi=Ueps  ~~ the upper bond for the sampling by \ int_u
  double tol;

  struct cumulative_function_params params = { h, N, val, u};
  x_lo = 0.0;
  x_hi = M_PI / h;
```

```
  tol = 1.e-6;
  F.F = &cumulative_function_ESM2;
  F.params = &params;

  return pnl_root_brent(&F, x_lo, x_hi, &tol);
}
```