Help

```
#include "bs1d_std.h"
#define INC 1.0e-5 /*Relative Increment for Delta-Hedging*/

/*assign_var_temp*/
static void assign_var_temp(double *sst,
                            double *alpha,
                            double *phi,
                            double *f,
                            double *b,
                            const double K,
                            const double T,
                            const double sigma,
                            const double delta,
                            const double r)
{
  *sst = sigma * sqrt(T);
  *b = delta - r + sigma * sigma / 2.;
  *f = sqrt((*b) * (*b) + 2.*r * sigma * sigma);
  *phi = ((*b) - (*f)) / 2.;
  *alpha = ((*b) + (*f)) / 2.;
}

/*assign_var_temp_L*/
static void assign_var_temp_L(const double sst,
                              const double alpha,
                              const double phi,
                              const double f,
                              const double b,
                              double *lambda,
                              double *d0,
                              double *d1pL,
                              double *d1pK,
                              double *d1mL,
                              double *d1mK,
                              const double K,
                              const double T,
                              const double sigma,
                              const double delta,
                              const double r,
```

```
                              const double x,
                              const double L)
{
  *lambda = x / L;
  *d0 = (log(*lambda) - f * T) / sst;
  *d1pL = (log(*lambda) + b * T) / sst;
  *d1pK = (log(*lambda * K / L) + b * T) / sst;
  *d1mL = (-log(*lambda) + b * T) / sst;
  *d1mK = (log(K / x) + b * T) / sst;
}

/*call_up_out*/
static double call_up_out(const double sst,
                          const double lambda,
                          const double alpha,
                          const double phi,
                          const double f,
                          const double b,
                          const double d0,
                          const double d1pL,
                          const double d1pK,
                          const double d1mL,
                          const double d1mK,
                          const double K,
                          const double T,
                          const double sigma,
                          const double delta,
                          const double r,
                          const double x,
                          const double L)
{
  double sig2 = sigma * sigma;
  double loglam = log(lambda);
  double ct =
    (L - K) * (exp(2 * phi / sig2 * loglam) * cdf_nor(d0) + exp(2 * alpha / sig2
    + x * exp(-delta * T) * (cdf_nor(d1mL - sst) - cdf_nor(d1mK - sst))
    - exp(-2 * (r - delta) / sig2 * loglam - delta * T) * L * (cdf_nor(d1pL - ss
    - K * exp(-r * T) * (cdf_nor(d1mL) - cdf_nor(d1mK) - exp((1 - 2 * (r - delta
  return ct;
}
```

```
/*dCdL*/
static double dCdL(const double sst,
                   const double lambda,
                   const double alpha,
                   const double phi,
                   const double f,
                   const double b,
                   const double d0,
                   const double d1pL,
                   const double d1pK,
                   const double d1mL,
                   const double d1mK,
                   const double K,
                   const double T,
                   const double sigma,
                   const double delta,
                   const double r,
                   const double x,
                   const double L)
{

  double sig2 = sigma * sigma;
  double loglam = log(lambda);
  double dCsdL =
    (1 - (L - K) / L * (2.*phi / sig2)) * exp(2.*phi * loglam / sig2) * cdf_nor(
    + (1 - (L - K) / L * (2.*alpha / sig2)) * exp(2.*alpha * loglam / sig2) * cd
    + exp(-delta * T) * 2.*(b - sig2) / sig2 * exp(-2.*(r - delta) * loglam / si
    * (cdf_nor(d1pL - sst) - cdf_nor(d1pK - sst))
    - exp(-r * T) * 2.*b * K / (sig2 * L) * exp(2.*b * loglam / sig2) * (cdf_nor
  return dCsdL;
}

/*maximise_C*/
static void maximise_C(double *Lmax,
                       const double sst,
                       double *lambda,
                       const double alpha,
                       const double phi,
                       const double f,
                       const double b,
                       double *d0,
```

```
                        double *d1pL,
                        double *d1pK,
                        double *d1mL,
                        double *d1mK,
                        const double K,
                        const double T,
                        const double sigma,
                        const double delta,
                        const double r,
                        const double x)
{
  double L1, L2, Ltmp, pas, derive;
  int i;
  L1 = x;
  L2 = 1000 * (x + K);
  pas = L2 - L1;
  for (i = 0; i <= 42; i++)
    {
      pas = pas / 2.;
      Ltmp = L1 + pas;
      assign_var_temp_L(sst, alpha, phi, f, b, lambda, d0, d1pL, d1pK, d1mL, d1m
      derive = dCdL(sst, *lambda, alpha, phi, f, b, *d0, *d1pL, *d1pK, *d1mL, *d
      if (derive <= 0) L2 = Ltmp;
      else L1 = Ltmp;

    };
  *Lmax = Ltmp;
}

/*call_lower_bound*/
static double call_lower_bound(const double K,
                               const double T,
                               const double sigma,
                               const double delta,
                               const double r,
                               const double x)
{
  double sst, lambda, alpha, phi, f, b, d0, d1pL, d1pK, d1mL, d1mK, L;
  double CLow, LLow;

  assign_var_temp(&sst, &alpha, &phi, &f, &b, K, T, sigma, delta, r);
```

```
  L = x * 1.5;
  assign_var_temp_L(sst, alpha, phi, f, b, &lambda, &d0, &d1pL, &d1pK, &d1mL, &d
  maximise_C(&LLow, sst, &lambda, alpha, phi, f, b, &d0, &d1pL, &d1pK, &d1mL, &d
  L = LLow;
  CLow = call_up_out(sst, lambda, alpha, phi, f, b, d0, d1pL, d1pK, d1mL, d1mK,
  return CLow;
}

/*D*/
static double D(const double sst,
                const double alpha,
                const double phi,
                const double f,
                const double b,
                const double K,
                const double T,
                const double sigma,
                const double delta,
                const double r,
                const double L)
{
  double dl, d1pL, d1pK, sig2;
  d1pK = (log(K / L) + b * T) / sst;
  d1pL = b * T / sst;
  sig2 = sigma * sigma;
  dl = (1 - (L - K) / L * 2 * phi / sig2) * cdf_nor(-f * sqrt(T) / sigma)
       + (1 - (L - K) / L * 2 * alpha / sig2) * cdf_nor(f * sqrt(T) / sigma)
       + exp(-delta * T) * 2 * (b - sig2) / sig2 * (cdf_nor(d1pL - sst) - cdf_no
       - exp(-r * T) * 2 * b * K / (sig2 * L) * (cdf_nor(d1pL) - cdf_nor(d1pK));
  return dl;
}

/*zero_de_D*/
static double zero_de_D(const double sst,
                        const double alpha,
                        const double phi,
                        const double f,
                        const double b,
                        const double K,
                        const double T,
                        const double sigma,
```

```
                        const double delta,
                        const double r,
                        const double ti)
{
  double L1, L2, Ltmp, pas, valtmp;
  int i;
  if (ti == T) return (MAX(r * K / delta, K));
  else
    {
      L2 = 1000 * K;
      if (D(sst, alpha, phi, f, b, K, T - ti, sigma, delta, r, L2) >= 0)
        for (i = 0; i < 40; i++)
          {
            if (D(sst, alpha, phi, f, b, K, T - ti, sigma, delta, r, L2) <= 0) b
            else L2 = 2 * L2;
          };

      L1 = K;
      pas = L2 - L1;
      for (i = 0; i <= 40; i++)
        {
          pas = pas / 2.;
          Ltmp = L1 + pas;
          valtmp = D(sst, alpha, phi, f, b, K, T - ti, sigma, delta, r, Ltmp);
          if (valtmp >= 0) L1 = Ltmp;
        };
      return L1;
    };
}

/*Ls*/
static double Ls(const double sst,
                 const double alpha,
                 const double phi,
                 const double f,
                 const double b,
                 const double K,
                 const double T,
                 const double sigma,
                 const double delta,
                 const double r,
```

```
                const double s)
{
  double L = zero_de_D(sst, alpha, phi, f, b, K, T, sigma, delta, r, s);
  return L;
}

/*d2*/
static double d2(const double xt,
                 const double Bs,
                 const double dt,
                 const double sigma,
                 const double delta,
                 const double r)
{
  double val;
  val = (log(xt / Bs) + (r - delta + sigma * sigma / 2.) * dt) / (sigma * sqrt(d
  return val;
}

/*d3*/
static double d3(const double xt,
                 const double Bs,
                 const double dt,
                 const double sigma,
                 const double delta,
                 const double r)
{
  double val;
  val = d2(xt, Bs, dt, sigma, delta, r) - sigma * sqrt(dt);
  return val;
}

/*integr*/
static double integr(const double alpha,
                     const double phi,
                     const double f,
                     const double b,
                     const double K,
                     const double T,
                     const double sigma,
                     const double delta,
```

```
                    const double r,
                    const double x)
{
  double inte, dx, mil, ray, t1, t2, f1, f2, Ls1, Ls2;
  int i;
  double xi[6] = {0,
                  0.1488743389,
                  0.4333953941,
                  0.6794095682,
                  0.8650633666,
                  0.9739065285
                 };
  double W[6] = {0,
                 0.2955242247,
                 0.2692667193,
                 0.2190863625,
                 0.1494513491,
                 0.0666713443
                };

  mil = T * 0.5;
  ray = T * 0.5;
  inte = 0;

  for (i = 1; i <= 5; i++)
    {
      dx = ray * xi[i];
      t1 = mil + dx;
      t2 = mil - dx;
      Ls1 = Ls(sigma * sqrt(T - t1), alpha, phi, f, b, K, T, sigma, delta, r, t1
      Ls2 = Ls(sigma * sqrt(T - t2), alpha, phi, f, b, K, T, sigma, delta, r, t2
      f1 = delta * x * exp(-delta * t1) * cdf_nor(d2(x, Ls1, t1, sigma, delta, r
      f2 = delta * x * exp(-delta * t2) * cdf_nor(d2(x, Ls2, t2, sigma, delta, r
      inte = inte + W[i] * (f1 + f2);
    };

  inte = inte * ray;
  return inte;
}

/*call_upper_bound*/
```

```
static double call_upper_bound(const double alpha,
                               const double phi,
                               const double f,
                               const double b,
                               const double K,
                               const double T,
                               const double sigma,
                               const double delta,
                               const double r,
                               const double x)
{
  double upper, Ceuro, c_delta;
  double Camer = integr(alpha, phi, f, b, K, T, sigma, delta, r, x);
  pnl_cf_call_bs(x, K, T, r, delta, sigma, &Ceuro, &c_delta);
  upper = Ceuro + Camer;
  return upper;
}

/*dCdx*/
static double dCdx(const double sst,
                   const double alpha,
                   const double phi,
                   const double f,
                   const double b,
                   const double K,
                   const double T,
                   const double sigma,
                   const double delta,
                   const double r,
                   const double x)
{
  double L1, L2, C1, C2, d0, d1pL, d1pK, d1mL, d1mK, lambda, derive;
  maximise_C(&L1, sst, &lambda, alpha, phi, f, b, &d0, &d1pL, &d1pK, &d1mL, &d1m
  C1 = call_up_out(sst, lambda, alpha, phi, f, b, d0, d1pL, d1pK, d1mL, d1mK, K,
  maximise_C(&L2, sst, &lambda, alpha, phi, f, b, &d0, &d1pL, &d1pK, &d1mL, &d1m
  C2 = call_up_out(sst, lambda, alpha, phi, f, b, d0, d1pL, d1pK, d1mL, d1mK, K,
  derive = (C2 - C1) / 0.0001;
  return derive;
}

/*coeff_upper*/
```

```
static double coeff_upper(const double sst,
                          const double alpha,
                          const double phi,
                          const double f,
                          const double b,
                          const double K,
                          const double T,
                          const double sigma,
                          const double delta,
                          const double r,
                          const double x,
                          const double CUp,
                          const double *CLower)
{
  double ceuro, Ceuro, c_delta, Lo,
         CLow,
         x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13, x14, x15, y2, c
  ceuro = pnl_cf_call_bs(x, K, T, r, delta, sigma, &Ceuro, &c_delta);
  CLow = call_lower_bound(K, T, sigma, delta, r, x);

  if ((CLow == ceuro) || (CLow <= (x - K)))(coef = 1);
  else
    {
      Lo = Ls(sst, alpha, phi, f, b, K, T, sigma, delta, r, 0);
      x1 = T;
      x2 = sqrt(T);
      x3 = r;
      x4 = delta;
      x5 = MIN(r / MAX(delta, 0.00001), 5);
      x6 = x5 * x5;
      x7 = dCdx(sst, alpha, phi, f, b, K, T, sigma, delta, r, x);
      x8 = x7 * x7;
      x9 = (CLow - ceuro) / K;
      x10 = x9 * x9;
      x11 = CLow / ceuro;
      x12 = (CUp - CLow) / K;
      x13 = CUp / CLow;
      x14 = x / Lo;
      x15 = x14 * x14;
      y2 = 0.8664 - 0.07668 * x1 + 0.3092 * x2
           - 0.3356 * x3 + 1.2 * x4 - 0.03507 * x5
```

```
          - 0.09755 * x6 - 0.7208 * x7 + 0.6071 * x8
          + 7.379 * x9 - 49.99 * x10 + 0.1148 * x11
          - 50.37 * x12 - 0.6629 * x13
          - 0.4745 * x14 + 0.5995 * x15;
      coef = MAX(MIN(y2, 1), 0);
    };
  return coef;
}


/*call_low_up_approx*/
static double call_low_up_approx(const double K,
                                 const double T,
                                 const double sigma,
                                 const double delta,
                                 const double r,
                                 const double x)
{
  double sst, alpha, phi, f, b, CUp, coef, LUBA, CLow, UnUsed;
  assign_var_temp(&sst, &alpha, &phi, &f, &b, K, T, sigma, delta, r);
  CUp = call_upper_bound(alpha, phi, f, b, K, T, sigma, delta, r, x);
  CLow = call_lower_bound(K, T, sigma, delta, r, x);

  coef = coeff_upper(sst, alpha, phi, f, b, K, T, sigma, delta, r, x, CUp, &UnUs
  coef = 0.5;

  LUBA = coef * CLow + (1 - coef) * CUp;
  return LUBA;
}

/*call_low_up_delta*/
static double call_low_up_delta(const double K,
                                const double T,
                                const double sigma,
                                const double delta,
                                const double r,
                                const double x,
                                const double luba)
{
  double luba1, low_up_delta;
  luba1 = call_low_up_approx(K, T, sigma, delta, r, x * (1. + INC));
```

```
  low_up_delta = (luba1 - luba) / (x * INC);
  return low_up_delta;
}

/*put_low_up_delta*/
static double put_low_up_delta(const double K,
                               const double T,
                               const double sigma,
                               const double delta,
                               const double r,
                               const double x,
                               const double luba)
{
  double luba1, low_up_delta;
  luba1 = call_low_up_approx(x * (1. + INC), T, sigma, r, delta, K);
  low_up_delta = (luba1 - luba) / (x * INC);
  return low_up_delta;
}


static int CallAmer_Luba(double x,
                         NumFunc_1 *p,
                         double T,
                         double r,
                         double delta,
                         double sigma,
                         double *call_price,
                         double *call_delta)
{
  double K;
  if ((p->Compute) == &Call)
    {
      K = p->Par[0].Val.V_DOUBLE;
      if (delta == 0.)
        {
          pnl_cf_call_bs(x, K, T, r, delta, sigma, call_price, call_delta);
        }
      else
        {
          *call_price = call_low_up_approx(K, T, sigma, delta, r, x);
          *call_delta = call_low_up_delta(K, T, sigma, delta, r, x, *call_price)
```

```
        }
    }
  else if ((p->Compute) == &Put)
    {
      K = p->Par[0].Val.V_DOUBLE;
      if (r == 0.)
        {
          pnl_cf_call_bs(K, x, T, delta, r, sigma, call_price, call_delta);
        }
      else
        {
          *call_price = call_low_up_approx(x, T, sigma, r, delta, K);
          *call_delta = put_low_up_delta(K, T, sigma, delta, r, x, *call_price);
        }
    }

  return OK;
}

int CALC(AP_Luba_CallAmer)(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 CallAmer_Luba(ptMod->S0.Val.V_PDOUBLE,
                       ptOpt->PayOff.Val.V_NUMFUNC_1,
                       ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE, r, divi
                       ptMod->Sigma.Val.V_PDOUBLE,
                       &(Met->Res[0].Val.V_DOUBLE), &(Met->Res[1].Val.V_DOUBLE))
}

static int CHK_OPT(AP_Luba_CallAmer)(void *Opt, void *Mod)
{
  if ((strcmp(((Option *)Opt)->Name, "CallAmer") == 0)
      || (strcmp(((Option *)Opt)->Name, "PutAmer") == 0))
    return OK;
  return WRONG;
```

```
}

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

  return OK;
}

PricingMethod MET(AP_Luba_CallAmer) =
{
  "AP_Luba",
  {{" ", PREMIA_NULLTYPE, {0}, FORBID}},
  CALC(AP_Luba_CallAmer),
  {{"Price", DOUBLE, {100}, FORBID}, {"Delta", DOUBLE, {100}, FORBID} , {" ", PR
  CHK_OPT(AP_Luba_CallAmer),
  CHK_ok ,
  MET(Init)
};
```