Help

```
#include <stdlib.h>
#include  "
href../../../mod/bs1d/bs1d_doublim/bs1d_doublim_h_src.pdfbs1d_doublim.h"
#include "
href../../../common/error_msg_h_src.pdferror_msg.h"

static int Ritchken_95_Out(int am, double s, NumFunc_1 *L, NumFunc_1 *U, NumFunc
                           double r, double divid, double sigma, int N, double l
{
  int i, j, npoints, eta0, A0;
  double h, puu, pum, pud, pdu, pdd, pdm, rebate, z, up, down, stock, lowerstock
  double *P, *G, *iv;

  /*return values: 0-ok 1-unable to allocate memory 2-barrier l to
    close to s*/


  /*Price, intrinsic value arrays*/
  npoints = 2 * N + 1;
  P = malloc(npoints * sizeof(double));
  if (P == NULL)
    return MEMORY_ALLOCATION_FAILURE;
  G = malloc(npoints * sizeof(double));
  if (G == NULL)
    return MEMORY_ALLOCATION_FAILURE;
  iv = malloc(npoints * sizeof(double));
  if (iv == NULL)
    return MEMORY_ALLOCATION_FAILURE;

  /*ASSUMES THE BARRIER ARE CONSTANTS*/
  up = (U->Compute)(U->Par, 0.); /*Upper barrier*/
  down = (L->Compute)(L->Par, 0.); /*Lower barrier*/
  rebate = (Rebate->Compute)(Rebate->Par, 0.); /*Same rebate at the two barriers

  /*Up and Down factors*/
  h = t / (double) N;
  eta = log(up / s) / (sigma * sqrt(h));
  eta0 = (int) floor(eta);
```

```
/*The Up barrier is too close to S0-the algorithm fails*/
if (eta0 < 2)
  return STEP_NUMBER_TOO_SMALL;

/*Adjustment of lambda to set a level of the tree at the barrier*/
/*In case the step number is not sufficient, then take the usual parameter*/
if (eta0 > N)
  {
    eta0 = N;
    /*In this case lambda keeps the value given in parameter*/
  }
else
  lambda = eta / (double)eta0;

lsh = lambda * sigma * sqrt(h);
u = exp(lsh);
d = 1. / u;

/*Discounted Probability*/
z = (r - divid) - SQR(sigma) / 2.;
puu = (1. / (2.*SQR(lambda)) + z * sqrt(h) / (2.*lambda * sigma));
pum = (1. - 1. / SQR(lambda));
pud = (1. - puu - pum);
puu *= exp(-r * h);
pum *= exp(-r * h);
pud *= exp(-r * h);

/*Stretching factor gamma*/
A = log(s / down) / lsh;
A0 = (int)floor(A) - 1;

/*The Down barrier is too close to S0-the algorithm fails*/
if (A0 < 0)
  return STEP_NUMBER_TOO_SMALL;

if (A0 > (N - 1))
  {
    A0 = N - 1;
    gamma = 1.;
  }
else
```

```
  {
    sd = s * exp(-A0 * lsh);
    gamma = log(sd / down) / lsh;
  }

/*Corrected Discounted Probability for the breaching downard mesh*/
a = z * sqrt(h) / (lambda * sigma);
b = 1. / SQR(lambda);
pdu = (b + a * gamma) / (1 + gamma);
pdd = (b - a) / (gamma * (1 + gamma));
pdm = (1. - pdu - pdd);
pdu *= exp(-r * h);
pdm *= exp(-r * h);
pdd *= exp(-r * h);

/*Intrinsic value initialization and terminal values*/
lowerstock = s;
npoints = A0;
for (i = 0; i < npoints; i++)
  {
    lowerstock *= d;
  }
stock = lowerstock;

npoints = eta0 + A0;
for (i = 0; i < npoints; i++)
  {
    iv[i] = (p->Compute)(p->Par, stock);
    P[i] = iv[i];
    G[i] = P[i];
    stock *= u;
  }

if (eta0 < N)
  {
    P[npoints] = rebate;
    G[npoints] = rebate;
  }
else
  {
    price = (p->Compute)(p->Par, up);
```

```
    P[npoints] = price;
    G[npoints] = price;
  }

/*Backward Resolution*/
/*First Part: At least one of the Barrier is active*/

/*First Case: the first (forward) active Barrier is the lower Barrier*/
if (A0 <= eta0)
  {
    /*The two Barriers are active*/
    for (i = 1; i <= N - eta0; i++)
      {
        P[0] = pdd * rebate + pdm * G[0] + pdu * G[1];
        if (am)
          P[0] = MAX(iv[0], P[0]);
        for (j = 1; j < npoints; j++)
          {
            P[j] = pud * G[j - 1] + pum * G[j] + puu * G[j + 1];
            if (am)
              P[j] = MAX(iv[j], P[j]);
          }
        for (j = 0; j < npoints; j++)
          G[j] = P[j];
      }
    /*Only the Lower Barrier is active*/
    for (i = N - eta0 + 1; i <= N - A0; i++)
      {
        npoints -= 1;
        P[0] = pdd * rebate + pdm * G[0] + pdu * G[1];
        if (am)
          P[0] = MAX(iv[0], P[0]);
        for (j = 1; j <= npoints; j++)
          {
            P[j] = pud * G[j - 1] + pum * G[j] + puu * G[j + 1];
            if (am)
              P[j] = MAX(iv[j], P[j]);
          }
        for (j = 0; j <= npoints; j++)
          G[j] = P[j];
      }
```

```
  }/*endif*/
else
  /*Second Case: the first (forward) active Barrier is the upper Barrier*/
  if (A0 > eta0)
    {
      /*The two Barriers are active*/
      for (i = 1; i <= N - A0; i++)
        {
          P[0] = pdd * rebate + pdm * G[0] + pdu * G[1];
          if (am)
            P[0] = MAX(iv[0], P[0]);
          for (j = 1; j < npoints; j++)
            {
              P[j] = pud * G[j - 1] + pum * G[j] + puu * G[j + 1];
              if (am)
                P[j] = MAX(iv[j], P[j]);
            }
          for (j = 0; j < npoints; j++)
            G[j] = P[j];
        }
      /*Only the upper Barrier is active*/
      for (i = N - A0 + 1; i <= N - eta0; i++)
        {
          npoints -= 1;
          for (j = 0; j < npoints; j++)
            {
              P[j] = pud * P[j] + pum * P[j + 1] + puu * P[j + 2];
              if (am)
                P[j] = MAX(iv[j + i], P[j]);
            }
          P[npoints] = rebate;
        }
    }/*endelse*/

/*Second Part: None of the Barriers are active*/
if (A0 > eta0) A0 = eta0;

npoints++;
for (i = N - A0 + 1; i < N; i++)
  {
    npoints -= 2;
```

```
      for (j = 0; j < npoints; j++)
        {
          P[j] = pud * P[j] + pum * P[j + 1] + puu * P[j + 2];
          if (am)
            P[j] = MAX(iv[j + i - (N - A0 + 1)], P[j]);
        }
    }

  /*Delta*/
  *ptdelta = (P[2] - P[0]) / (s * u - s * d);

  /*First time step*/
  P[0] = pud * P[0] + pum * P[1] + puu * P[2];
  if (am)
    P[0] = MAX(iv[A0], P[0]);

  /*Price*/
  *ptprice = P[0];

  /*Memory Desallocation*/
  free(P);
  free(G);
  free(iv);

  return OK;
}


int CALC(TR_Ritchken_Out)(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 Ritchken_95_Out(ptOpt->EuOrAm.Val.V_BOOL,
                         ptMod->S0.Val.V_PDOUBLE,
                         ptOpt->LowerLimit.Val.V_NUMFUNC_1, ptOpt->UpperLimit.Va
                         ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
```

```
                          r, divid,
                          ptMod->Sigma.Val.V_PDOUBLE,
                          Met->Par[0].Val.V_INT, Met->Par[1].Val.V_RGDOUBLE12, &(
}

static int CHK_OPT(TR_Ritchken_Out)(void *Opt, void *Mod)
{
  Option *ptOpt = (Option *)Opt;
  TYPEOPT *opt = (TYPEOPT *)(ptOpt->TypeOpt);

  if ((opt->TwoDoubleStep).Val.V_BOOL == FALSE)
    if ((opt->Parisian).Val.V_BOOL == FALSE)
      if ((opt->OutOrIn).Val.V_BOOL == OUT)
        return OK;

  return WRONG;
}

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

      Met->Par[0].Val.V_INT2 = 100;
      Met->Par[1].Val.V_RGDOUBLE12 = 1.22474;
    }

  return OK;
}

PricingMethod MET(TR_Ritchken_Out) =
{
  "TR_Ritchken_Out",
  {
    {"StepNumber", INT2, {100}, ALLOW},
    {"Lambda", RGDOUBLE12, {1}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(TR_Ritchken_Out),
  {
```

```
    {"Price", DOUBLE, {100}, FORBID},
    {"Delta", DOUBLE, {100}, FORBID} ,
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(TR_Ritchken_Out),
  CHK_tree,
  MET(Init)
};
```