Help

```
#include <stdlib.h>
#include  "bscir2d_std.h"
#include "error_msg.h"
#include "pnl/pnl_matrix.h"

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

static double dt;
static double **V, * *S, * **P;
static double **y, * *f;
static int **f_down, * *f_up;
static int **y_down, * *y_up;
static double **pu_y, * *pd_y;
static double **pu_f, * *pd_f;
static int Ns;

/*Model parameters*/
static double omega, kappa, theta, rho, r0;
static double sigma, s0, tt;
static int am;
static int dummy;
static double sqrt_dt;

/*Memory Allocation*/
static void memory_allocation_wei(int Nst)
{
  int i, j;

  V = (double **)calloc(Nst + 1, sizeof(double *));
  for (i = 0; i < Nst + 1; i++)
```

```
  {
    V[i] = (double *)calloc(Nst + 1, sizeof(double));
  }

S = (double **)calloc(Nst + 1, sizeof(double *));
for (i = 0; i < Nst + 1; i++)
  {
    S[i] = (double *)calloc(Nst + 1, sizeof(double));
  }

pu_y = (double **)calloc(Nst + 1, sizeof(double *));
for (i = 0; i < Nst + 1; i++)
  {
    pu_y[i] = (double *)calloc(Nst + 1, sizeof(double));

  }

pd_y = (double **)calloc(Nst + 1, sizeof(double *));
for (i = 0; i < Nst + 1; i++)
  {
    pd_y[i] = (double *)calloc(Nst + 1, sizeof(double));
  }

pu_f = (double **)calloc(Nst + 1, sizeof(double *));
for (i = 0; i < Nst + 1; i++)
  {
    pu_f[i] = (double *)calloc(Nst + 1, sizeof(double));

  }

pd_f = (double **)calloc(Nst + 1, sizeof(double *));
for (i = 0; i < Nst + 1; i++)
  {
    pd_f[i] = (double *)calloc(Nst + 1, sizeof(double));

  }

y = (double **)calloc(Nst + 1, sizeof(double *));
for (i = 0; i < Nst + 1; i++)
  {
    y[i] = (double *)calloc(Nst + 1, sizeof(double));
```

```
  }

f = (double **)calloc(Nst + 1, sizeof(double *));
for (i = 0; i < Nst + 1; i++)
  {
    f[i] = (double *)calloc(Nst + 1, sizeof(double));

  }

f_down = (int **)calloc(Nst + 1, sizeof(int *));
for (i = 0; i < Nst + 1; i++)
  {
    f_down[i] = (int *)calloc(Nst + 1, sizeof(int));

  }

f_up = (int **)calloc(Nst + 1, sizeof(int *));
for (i = 0; i < Nst + 1; i++)
  {
    f_up[i] = (int *)calloc(Nst + 1, sizeof(int));

  }

y_down = (int **)calloc(Nst + 1, sizeof(int *));
for (i = 0; i < Nst + 1; i++)
  {
    y_down[i] = (int *)calloc(Nst + 1, sizeof(int));
  }

y_up = (int **)calloc(Nst + 1, sizeof(int *));
for (i = 0; i < Nst + 1; i++)
  {
    y_up[i] = (int *)calloc(Nst + 1, sizeof(int));

  }

P = (double ** *)malloc((Nst + 1) * sizeof(double **));
for (i = 0; i <= Nst; i++)
  P[i] = (double **)malloc((Nst + 1) * sizeof(double *));
for (i = 0; i <= Nst; i++)
```

```
    for (j = 0; j <= Nst; j++)
      P[i][j] = (double *)malloc((Nst + 1) * sizeof(double));
}

static void memory_free_wei(long Nst)
{
  int i, j;

  for (i = 0; i < Nst + 1; i++)
    free(S[i]);
  free(S);

  for (i = 0; i < Nst + 1; i++)
    free(V[i]);
  free(V);

  for (i = 0; i < Nst + 1; i++)
    free(pu_y[i]);
  free(pu_y);

  for (i = 0; i < Nst + 1; i++)
    free(pd_y[i]);
  free(pd_y);

  for (i = 0; i < Nst + 1; i++)
    free(y[i]);
  free(y);

  for (i = 0; i < Nst + 1; i++)
    free(y_up[i]);
  free(y_up);

  for (i = 0; i < Nst + 1; i++)
    free(y_down[i]);
  free(y_down);

  for (i = 0; i < Nst + 1; i++)
    free(pu_f[i]);
  free(pu_f);

  for (i = 0; i < Nst + 1; i++)
```

```
    free(pd_f[i]);
  free(pd_f);

  for (i = 0; i < Nst + 1; i++)
    free(f[i]);
  free(f);

  for (i = 0; i < Nst + 1; i++)
    free(f_up[i]);
  free(f_up);

  for (i = 0; i < Nst + 1; i++)
    free(f_down[i]);
  free(f_down);

  for (i = 0; i < Nst + 1; i++)
    for (j = 0; j < Nst + 1; j++)
      free(P[i][j]);
  for (j = 0; j < Nst + 1; j++)
    free(P[j]);
  free(P);
}


static double compute_f(double r)
{
  return 2.*sqrt(r) / omega;
}

static double compute_r(double R)
{
  double val;

  if (R > 0.)
    val = SQR(R) * SQR(omega) / 4.;
  else
    val = 0.;
  return val;
}

static double compute_y(double s, double r)
```

```
{
  double Y;

  Y = (log(s) / sigma);

  return Y;
}

static double compute_S(double Y, double R)
{
  double val;

  val = exp(sigma * Y);

  return val;
}

/*Calibration of the tree*/
static int calibration()
{
  int i, j;
  int z;
  double Ru, Rd;
  double mu_r, r_curr;

  /*Fixed tree for R=f*/
  f[0][0] = compute_f(r0);

  dt = tt / (double)Ns;
  sqrt_dt = sqrt(dt);

  V[0][0] = compute_r(f[0][0]);
  f[1][0] = f[0][0] - sqrt_dt;
  f[1][1] = f[0][0] + sqrt_dt;
  V[1][0] = compute_r(f[1][0]);
  V[1][1] = compute_r(f[1][1]);
  for (i = 1; i < Ns; i++)
    for (j = 0; j <= i; j++)
      {
        f[i + 1][j] = f[i][j] - sqrt_dt;
        f[i + 1][j + 1] = f[i][j] + sqrt_dt;
```

```
        V[i + 1][j] = compute_r(f[i + 1][j]);
        V[i + 1][j + 1] = compute_r(f[i + 1][j + 1]);
      }

  /*Evolve tree for f*/
  for (i = 0; i < Ns; i++)
    {
      for (j = 0; j <= i; j++)
        {
          /*Compute mu_f*/
          r_curr = V[i][j];

          mu_r = kappa * (theta - r_curr);

          z = 0;
          while ((V[i][j] + mu_r * dt < V[i + 1][j - z])
                 && (j - z >= 0))
            {

              z = z + 1;
            }
          f_down[i][j] = -z;
          Rd = V[i + 1][j - z];

          z = 0;
          while ((V[i][j] + mu_r * dt > V[i + 1][j + z])
                 && (j + z <= i))
            {
              z = z + 1;
            }

          Ru = V[i + 1][j + z];

          f_up[i][j] = z;
          pu_f[i][j] = (V[i][j] + mu_r * dt - Rd) / (Ru - Rd);

          if ((Ru - 1.e-9 > V[i + 1][i + 1]) || (j + f_up[i][j] > i + 1))
            {
              pu_f[i][j] = 1;

              f_up[i][j] = i + 1 - j;
```

```
              f_down[i][j] = i - j;
            }

          if ((Rd + 1.e-9 < V[i + 1][0]) || (j + f_down[i][j] < 0))
            {
              pu_f[i][j] = 0.;
              f_up[i][j] = 1 - j;
              f_down[i][j] = 0 - j;
            }
          pd_f[i][j] = 1. - pu_f[i][j];

        }
    }

  y[0][0] = compute_y(s0, r0);
  S[0][0] = compute_S(y[0][0], f[0][0]);

  y[1][0] = y[0][0] - sqrt_dt;
  y[1][1] = y[0][0] + sqrt_dt;
  S[1][0] = compute_S(y[1][0], f[1][0]);
  S[1][1] = compute_S(y[1][1], f[1][1]);

  for (i = 1; i < Ns; i++)
    for (j = 0; j <= i; j++)
      {
        y[i + 1][j] = y[i][j] - sqrt_dt;
        y[i + 1][j + 1] = y[i][j] + sqrt_dt;
        S[i + 1][j] = compute_S(y[i + 1][j], f[i + 1][j]);
        S[i + 1][j + 1] = compute_S(y[i + 1][j + 1], f[i + 1][j + 1]);
      }
  return 1;
}


/*Compute Price Bond*/
static int compute_price(NumFunc_1  *p, double *price)
{
  int i, j, k;
  double puu, pud, pdu, pdd, stock;
  int fv_up, fv_down, yv_up, yv_down;
  double pu_ys;
```

```
int z;
double Ru, Rd;
double mu_s, s_curr, r_curr;
double alpha, beta, gama, eta, csi;
double a, b, c, DSu, DSd, Dru, Drd;

/*Maturity conditions for Call options*/
for (j = 0; j <= Ns; j++)
  for (k = 0; k <= Ns; k++)
    {
      stock = compute_S(y[Ns][j], f[Ns][k]);
      P[Ns][j][k] = (p->Compute)(p->Par, stock);

    }

/*Dynamic Programming*/
for (i = Ns - 1; i >= 0; i--)
  {
    for (j = 0; j <= i; j++)
      for (k = 0; k <= i; k++)
        {
          /*node Y*/
          s_curr = compute_S(y[i][j], f[i][k]);
          r_curr = compute_r(f[i][k]);
          mu_s = s_curr * r_curr;

          z = 0;
          while ((compute_S(y[i][j], f[i][k]) + mu_s * dt < compute_S(y[i + 1]
            {
              z++;
            }
          yv_down = -z;
          Rd = y[i + 1][j] - 2 * (double)z * sqrt_dt;

          z = 0;
          while ((compute_S(y[i][j], f[i][k]) + mu_s * dt > compute_S(y[i + 1]
            {
              z++;
            }
          yv_up = z + 1;
          Ru = y[i + 1][j + 1] + 2 * (double)z * sqrt_dt;
```

```
pu_ys = (compute_S(y[i][j], f[i][k]) + mu_s * dt - compute_S(Rd, f[i
        (compute_S(Ru, f[i][k]) - compute_S(Rd, f[i][k]));



if (Ru - 1.e-9 > y[i + 1][i + 1])
  {
    pu_ys = 1.;

    yv_up = i + 1 - j;
    yv_down = i - j;

  }

if (Rd + 1.e-9 < y[i + 1][0])
  {
    pu_ys = 0.;

    yv_up = -j + 1;
    yv_down = -j;

  }

fv_up = f_up[i][k];
fv_down = f_down[i][k];

DSu = compute_S(Ru, f[i][k]) - s_curr;
DSd = compute_S(Rd, f[i][k]) - s_curr;

Dru = compute_r(f[i + 1][k + fv_up]) - r_curr;
Drd = compute_r(f[i + 1][k + fv_down]) - r_curr;


a = pu_f[i][k] + pu_ys - 1.;
b = 1. - pu_f[i][k];
c = 1. - pu_ys;

alpha = DSu * Dru;
beta = DSu * Drd;
```

```
            gama = DSd * Dru;
            eta = DSd * Drd;

            csi = rho * sigma * s_curr * omega * sqrt(r_curr) * dt;

            pdd = (csi - alpha * a - beta * b - gama * c) / (alpha - beta - gama

            puu = pdd + a;
            pud = -pdd + b;
            pdu = -pdd + c;

            P[i][j][k] = exp(-V[i][k] * dt) * (puu * P[i + 1][j + yv_up][k + fv_
            if (am)
              {
                stock = compute_S(y[i][j], f[i][k]);
                P[i][j][k] = MAX(P[i][j][k], (p->Compute)(p->Par, stock));
              }
          }
    }

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

  return 1;
}

static int tr_acz(int am_in, double s0_in, NumFunc_1  *p, double tt_in, double s

{
  double price;

  am = am_in;
  s0 = s0_in;
  tt = tt_in;
  sigma = sigma_in;
  r0 = r0_in;
  kappa = kappa_in;
  theta = theta_in;
  omega = omega_in; //r volatility
  rho = rho_in;
  Ns = Ns_in;
```

```
  dt = tt / (double)Ns;
  sqrt_dt = sqrt(dt);

  /*Memory Allocation*/
  memory_allocation_wei(Ns);

  //Calibrate interest rate tree
  dummy = calibration();

  //Compute Price
  dummy = compute_price(p, &price);

  *ptprice = price;

  //Memory desallocation
  memory_free_wei(Ns);

  return OK;
}

int CALC(TR_acz)(void *Opt, void *Mod, PricingMethod *Met)
{
  TYPEOPT *ptOpt = (TYPEOPT *)Opt;
  TYPEMOD *ptMod = (TYPEMOD *)Mod;

  double kappa, theta, omega;

  omega = ptMod->Sigma.Val.V_PDOUBLE;
  kappa = ptMod->k.Val.V_PDOUBLE;
  theta = ptMod->theta.Val.V_PDOUBLE;
  if (2 * kappa * theta < SQR(omega))
    {
      Fprintf(TOSCREEN, "UNTREATED CASE\ n\ n\ n");
      return WRONG;
    }
  else
    return tr_acz(ptOpt->EuOrAm.Val.V_BOOL, ptMod->S0.Val.V_PDOUBLE, ptOpt->PayO
}

static int CHK_OPT(TR_acz)(void *Opt, void *Mod)
{
```

```
  return OK;
}

#endif //PremiaCurrentVersion

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

      Met->Par[0].Val.V_INT2 = 50;
    }

  return OK;
}

PricingMethod MET(TR_acz) =
{
  "TR_ACZ",
  {{"Step Numbers", INT2, {100}, ALLOW}, {" ", PREMIA_NULLTYPE, {0}, FORBID}},
  CALC(TR_acz),
  {{"Price", DOUBLE, {100}, FORBID}, {" ", PREMIA_NULLTYPE, {0}, FORBID}},
  CHK_OPT(TR_acz),
  CHK_tree,
  MET(Init)
};
```