Help

```
#include <stdlib.h>
#include  "bs1d_lim.h"
#include "error_msg.h"
#define PRECISION 1.0e-7 /*Precision for the localization of FD methods*/

static int Cryer_DownOut(double s, NumFunc_1  *p, double l, double rebate, doubl
{
  int      Index, PriceIndex, TimeIndex, ssl;
  double   k, vv, loc, h, z, alpha, beta, gamma, y, down, upwind_alphacoef;
  double   *Obst, *A, *B, *C, *P, *S, *Z, *Q, pricenh, pricen2h, priceph;

  /*Memory Allocation*/
  Obst = malloc((N + 1) * sizeof(double));
  if (Obst == NULL)
    return MEMORY_ALLOCATION_FAILURE;
  A = malloc((N + 1) * sizeof(double));
  if (A == NULL)
    return MEMORY_ALLOCATION_FAILURE;
  B = malloc((N + 1) * sizeof(double));
  if (B == NULL)
    return MEMORY_ALLOCATION_FAILURE;
  C = malloc((N + 1) * sizeof(double));
  if (C == NULL)
    return MEMORY_ALLOCATION_FAILURE;
  P = malloc((N + 1) * sizeof(double));
  if (P == NULL)
    return MEMORY_ALLOCATION_FAILURE;
  S = malloc((N + 1) * sizeof(double));
  if (S == NULL)
    return MEMORY_ALLOCATION_FAILURE;
  Z = malloc((N + 1) * sizeof(double));
  if (Z == NULL)
    return MEMORY_ALLOCATION_FAILURE;
  Q = malloc((N + 1) * sizeof(double));
  if (Q == NULL)
    return MEMORY_ALLOCATION_FAILURE;


  /*Time Step*/
```

```
k = t / (double)M;

/*Space Localisation*/
vv = 0.5 * sigma * sigma;
z = (r - divid) - vv;
loc = sigma * sqrt(t) * sqrt(log(1.0 / PRECISION)) + fabs(z * t);

/*Space Step*/
y = log(s);
down = log(l);
h = (y + loc - down) / (double)(N);

/*Peclet Condition-Coefficient of diffusion augmented */
if ((h * fabs(z)) <= vv)
  upwind_alphacoef = 0.5;
else
  {
    if (z > 0.) upwind_alphacoef = 0.0;
    else  upwind_alphacoef = 1.0;
  }
vv -= z * h * (upwind_alphacoef - 0.5);

/*Lhs Factor of theta-schema*/
alpha = k * (-vv / (h * h) + z / (2.0 * h));
beta = 1.0 + k * (r + 2.*vv / (h * h));
gamma = k * (-vv / (h * h) - z / (2.0 * h));

for (PriceIndex = 0; PriceIndex <= N - 2; PriceIndex++)
  {
    A[PriceIndex] = alpha;
    B[PriceIndex] = beta;
    C[PriceIndex] = gamma;
  }

/*Ternminal Values*/
y = log(s);
for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
  Obst[PriceIndex - 1] = (p->Compute)(p->Par, exp(down + PriceIndex * h));

for (PriceIndex = 2; PriceIndex <= N - 2; PriceIndex++)
  {
```

```
    P[PriceIndex - 1] = alpha * Obst[PriceIndex - 2] +
                        beta * Obst[PriceIndex - 1] + gamma * Obst[PriceIndex]
  }

P[0] = beta * Obst[0] + gamma * Obst[1];
P[N - 2] = alpha * Obst[N - 3] + beta * Obst[N - 2];

for (PriceIndex = 0; PriceIndex <= N - 2; PriceIndex++)
  {
    S[PriceIndex] = 0.0;
    Z[PriceIndex] = 0.0;
  }
ssl = false;

/*Finite Difference Cycle*/
for (TimeIndex = 1; TimeIndex <= M; TimeIndex++)
  {
    for (PriceIndex = 0; PriceIndex <= N - 2; PriceIndex++)
      Z[PriceIndex] = Z[PriceIndex] + Obst[PriceIndex];

    for (PriceIndex = 0; PriceIndex <= N - 2; PriceIndex++)
      Q[PriceIndex] = P[PriceIndex] - Z[PriceIndex];
    Q[0] += alpha * rebate;
    Q[N - 2] += gamma * (p->Compute)(p->Par, exp(y + loc));

    AlgCrayer(N, Z, ssl, A, B, C, Q, S);

    for (PriceIndex = 0; PriceIndex <= N - 2; PriceIndex++)
      S[PriceIndex] = Z[PriceIndex];

    ssl = true;
  }

for (PriceIndex = 0; PriceIndex <= N - 2; PriceIndex++)
  P[PriceIndex] = Z[PriceIndex] + Obst[PriceIndex];

Index = (int)floor((y - down) / h) - 1;

/*Price*/
*ptprice = P[Index] + (P[Index + 1] - P[Index]) * (exp(y) - exp(down + h + Ind
```

```
  /*Delta*/
  pricenh = P[Index + 1] + (P[Index + 2] - P[Index + 1]) * (exp(y + h) - exp(dow
  if (Index > 0)
    {
      priceph = P[Index - 1] + (P[Index] - P[Index - 1]) * (exp(y - h) - exp(dow
      *ptdelta = (pricenh - priceph) / (2 * s * h);
    }
  else
    {
      pricen2h = P[Index + 2] + (P[Index + 3] - P[Index + 2]) * (exp(y + 2 * h)
      *ptdelta = (4 * pricenh - pricen2h - 3 * (*ptprice)) / (2 * s * h);
    }

  /*Memory Desallocation*/
  free(Obst);
  free(A);
  free(B);
  free(C);
  free(P);
  free(S);
  free(Z);
  free(Q);

  return OK;
}


int CALC(FD_Cryer_DownOut)(void *Opt, void *Mod, PricingMethod *Met)
{
  TYPEOPT *ptOpt = (TYPEOPT *)Opt;
  TYPEMOD *ptMod = (TYPEMOD *)Mod;
  double r, divid, limit, rebate;

  r = log(1. + ptMod->R.Val.V_DOUBLE / 100.);
  divid = log(1. + ptMod->Divid.Val.V_DOUBLE / 100.);
  limit = ((ptOpt->Limit.Val.V_NUMFUNC_1)->Compute)((ptOpt->Limit.Val.V_NUMFUNC_
  rebate = ((ptOpt->Rebate.Val.V_NUMFUNC_1)->Compute)((ptOpt->Rebate.Val.V_NUMFU

  return Cryer_DownOut(ptMod->S0.Val.V_PDOUBLE, ptOpt->PayOff.Val.V_NUMFUNC_1,
                       limit, rebate, ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.
                       Met->Par[0].Val.V_INT2, Met->Par[1].Val.V_INT2,
```

```
                        &(Met->Res[0].Val.V_DOUBLE), &(Met->Res[1].Val.V_DOUBLE))
}

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

  if ((opt->Parisian).Val.V_BOOL == FALSE)
    if ((strcmp(((Option *)Opt)->Name, "CallDownOutAmer") == 0) || (strcmp(((Opt
      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_INT2 = 100;

    }

  return OK;
}

PricingMethod MET(FD_Cryer_DownOut) =
{
  "FD_Cryer_DownOut",
  { {"SpaceStepNumber", INT2, {100}, ALLOW    }, {"TimeStepNumber", INT2, {100},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(FD_Cryer_DownOut),
  {{"Price", DOUBLE, {100}, FORBID}, {"Delta", DOUBLE, {100}, FORBID} , {" ", PR
  CHK_OPT(FD_Cryer_DownOut),
  CHK_split,
  MET(Init)
};
```