Help

```
#include <stdlib.h>
#include  "
href../../../mod/bs1d/bs1d_lim/bs1d_lim_h_src.pdfbs1d_lim.h"
#include "
href../../../common/error_msg_h_src.pdferror_msg.h"
#define PRECISION 1.0e-7 /*Precision for the localization of FD methods*/

static int Gauss_DownOut(int am, double s, NumFunc_1  *p, double l, double rebat
{
  int      Index, PriceIndex, TimeIndex;
  double   k, vv, loc, h, z, alpha, beta, gamma, y, alpha1, beta1, gamma1, down,
  double   *Obst, *A, *B, *C, *P, *S, pricenh, pricen2h, priceph;

  /*Memory Allocation*/
  Obst = malloc((N + 2) * sizeof(double));
  if (Obst == NULL)
    return MEMORY_ALLOCATION_FAILURE;
  A = malloc((N + 2) * sizeof(double));
  if (A == NULL)
    return MEMORY_ALLOCATION_FAILURE;
  B = malloc((N + 2) * sizeof(double));
  if (B == NULL)
    return MEMORY_ALLOCATION_FAILURE;
  C = malloc((N + 2) * sizeof(double));
  if (C == NULL)
    return MEMORY_ALLOCATION_FAILURE;
  P = malloc((N + 2) * sizeof(double));
  if (P == NULL)
    return MEMORY_ALLOCATION_FAILURE;
  S = malloc((N + 2) * sizeof(double));
  if (S == 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 + 1);

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

for (PriceIndex = 1; PriceIndex <= N; PriceIndex++)
  {
    A[PriceIndex] = alpha;
    B[PriceIndex] = beta;
    C[PriceIndex] = gamma;
  }
/*Rhs Factor of theta-schema*/
alpha1 = k * (1.0 - theta) * (vv / (h * h) - z / (2.0 * h));
beta1 = 1.0 - k * (1.0 - theta) * (r + 2.*vv / (h * h));
gamma1 = k * (1.0 - theta) * (vv / (h * h) + z / (2.0 * h));

/*Set Gauss*/
for (PriceIndex = N - 1; PriceIndex >= 1; PriceIndex--)
  B[PriceIndex] = B[PriceIndex] - C[PriceIndex] * A[PriceIndex + 1] / B[PriceI
for (PriceIndex = 1; PriceIndex <= N; PriceIndex++)
  A[PriceIndex] = A[PriceIndex] / B[PriceIndex];
for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
  C[PriceIndex] = C[PriceIndex] / B[PriceIndex + 1];
```

```
/*Terminal Values*/
for (PriceIndex = 1; PriceIndex <= N + 1; PriceIndex++)
  {
    Obst[PriceIndex] = (p->Compute)(p->Par, exp(down + (double)PriceIndex * h)
    P[PriceIndex] = Obst[PriceIndex];
  }

/*Dirichlet Boundary Condition*/
P[0] = rebate;

/*Finite Difference Cycle*/
for (TimeIndex = 1; TimeIndex <= M; TimeIndex++)
  {
    /*Set Rhs*/
    S[1] = beta1 * P[1] + gamma1 * P[2] + alpha1 * P[0];
    S[1] = beta1 * P[1] + gamma1 * P[2] + alpha1 * P[0] - alpha * P[0];
    for (PriceIndex = 2; PriceIndex < N; PriceIndex++)
      S[PriceIndex] = alpha1 * P[PriceIndex - 1] + beta1 * P[PriceIndex] +
                      gamma1 * P[PriceIndex + 1];
    S[N] = alpha1 * P[N - 1] + beta1 * P[N] + gamma1 * P[N + 1] - gamma * P[N


    /*Solve the system*/
    for (PriceIndex = N - 1; PriceIndex >= 1; PriceIndex--)
      S[PriceIndex] = S[PriceIndex] - C[PriceIndex] * S[PriceIndex + 1];

    P[1] = S[1] / B[1];

    for (PriceIndex = 2; PriceIndex <= N; PriceIndex++)
      P[PriceIndex] = S[PriceIndex] / B[PriceIndex] - A[PriceIndex] * P[PriceI
    /*Splitting for the american case*/
    if (am)
      for (PriceIndex = 1; PriceIndex <= N; PriceIndex++)
        P[PriceIndex] = MAX(Obst[PriceIndex], P[PriceIndex]);
  }
Index = (int)floor((y - down) / h);

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

/*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);

  return OK;
}

int CALC(FD_Gauss_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 Gauss_DownOut(ptOpt->EuOrAm.Val.V_BOOL, ptMod->S0.Val.V_PDOUBLE, ptOpt-
                       limit, rebate, ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.
                       Met->Par[0].Val.V_INT2, Met->Par[1].Val.V_INT2, Met->Par[
                       &(Met->Res[0].Val.V_DOUBLE), &(Met->Res[1].Val.V_DOUBLE))
}

static int CHK_OPT(FD_Gauss_DownOut)(void *Opt, void *Mod)
```

```
{
  Option *ptOpt = (Option *)Opt;
  TYPEOPT *opt = (TYPEOPT *)(ptOpt->TypeOpt);

  if ((opt->OutOrIn).Val.V_BOOL == OUT)
    if ((opt->DownOrUp).Val.V_BOOL == DOWN)
      if ((opt->Parisian).Val.V_BOOL == FALSE)
        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;
      Met->Par[2].Val.V_RGDOUBLE = 0.5;


    }

  return OK;
}

PricingMethod MET(FD_Gauss_DownOut) =
{
  "FD_Gauss_DownOut",
  { {"SpaceStepNumber", INT2, {100}, ALLOW    }, {"TimeStepNumber", INT2, {100},
    {"Theta", RGDOUBLE051, {100}, ALLOW}, {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(FD_Gauss_DownOut),
  {{"Price", DOUBLE, {100}, FORBID}, {"Delta", DOUBLE, {100}, FORBID} , {" ", PR
  CHK_OPT(FD_Gauss_DownOut),
  CHK_split,
  MET(Init)
};
```