```
  Help
#include <stdlib.h>
#include  "bs1d_std.h"
#include "error_msg.h"
#define PRECISION 1.0e-7 /*Precision for the localization
    of FD methods*/


static void restriction(int l,double *d,double *u,double *
    f,double alpha,double beta,double gamma)
{
  int nl1,nl,i;
  double *Aux;

  nl=pow(2, l+1)-1;
  nl1=pow(2,l)-1;

  Aux= malloc((nl+2)*sizeof(double));

  for (i=1;i<=nl;i++)
    Aux[i]=alpha*u[i-1]+beta*u[i]+gamma*u[i+1]-f[i];


  for (i=1;i<=nl1;i++)
    d[i]=Aux[2*i]/2.0+(Aux[2*i-1]+Aux[2*i+1])/4.0;

  free(Aux);

  return;
}


static void substract_prolongation(int l,double *u,double *
    v)
{
  int nl1,i;

  nl1=pow(2, l)-1;

  for (i=0;i<=nl1;i++)
    {
```

```
      u[2*i]=u[2*i]-v[i];
      u[2*i+1]=u[2*i+1]-(v[i]+v[i+1])/2.0;
    }
  return;
}


static int MGM(int l,double *u,double *f,double t,double r,
    double divid,double sigma,int N,int M,double theta)
{
  int nl,nl1,i,j;
  double *v,*d;
  double h,k,vv,limit,alpha,beta,gamma,z,upwind_alphacoef;

  nl=pow(2, l+1)-1;
  nl1=pow(2, l)-1;

  /*Memory Allocation*/
  v= malloc((nl1+2)*sizeof(double));
  if (v==NULL)
    return MEMORY_ALLOCATION_FAILURE;
  d= malloc((nl1+2)*sizeof(double));
  if (d==NULL)
    return MEMORY_ALLOCATION_FAILURE;

  /*Time Step*/
  k=t/(double)M;

  /*Space Localisation*/
  z=(r-divid)-SQR(sigma)/2.0;
  limit=sigma*sqrt(t)*sqrt(log(1.0/PRECISION))+fabs(z)*t;

  /*Space Step*/
  h=2.*limit/(double)(nl+1);


  /*Peclet Condition-Coefficient of diffusion augmented */
  vv=0.5*SQR(sigma);
  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);

/*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));



if (l==0) {u[1]=f[1]/(1+k*theta*(vv/h/h+r));}
else
  {
    /* 2 iterations of Gauss-Seidel*/
    for (i=1;i<3;i++)
{
  for (j=1;j<=nl;j++)
    u[j]=(-u[j-1]*alpha-u[j+1]*gamma+f[j])/beta;
}

    restriction(l,d,u,f,alpha,beta,gamma);

    for (i=0;i<nl1+2;i++)
v[i]=0.0;

    MGM(l-1,v,d,t,r,divid,sigma,N,M,theta);

    substract_prolongation(l,u,v);

    /* 2 iterations of Gauss-Seidel*/
    for (i=1;i<3;i++)
{
  for (j=1;j<=nl;j++)
    u[j]=(-u[j-1]*alpha-u[j+1]*gamma+f[j])/beta;
}

  }

free(v);
```

```
  free(d);

  return OK;
}


static int mult_euro1(double s,NumFunc_1  *p,double t,
    double r,double divid, double sigma,int l,int M,double theta,
    double *ptprice,double *ptdelta)
{
  double k,z,limit,h,x,alpha1,beta1,gamma1,vv,upwind_alpha
    coef;
  double *P,*Rhs;
  int Index,i,j,N;

  N=pow(2, l+1)-1+1;
  /*Memory Allocation*/
  P= malloc((N+1)*sizeof(double));
  if (P==NULL)
    return MEMORY_ALLOCATION_FAILURE;
  Rhs= malloc((N+1)*sizeof(double));
  if (Rhs==NULL)
    return MEMORY_ALLOCATION_FAILURE;

  /*Time Step*/
  k=t/(double)M;

  /*Space Localisation*/
  z=(r-divid)-SQR(sigma)/2.0;
  limit=sigma*sqrt(t)*sqrt(log(1.0/PRECISION))+fabs(z)*t;

  /*Space Step*/
  h=2*limit/(double)N;

  /*Peclet Condition-Coefficient of diffusion augmented */
  vv=0.5*SQR(sigma);
  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);

/*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));

/*Terminal Values*/
x=log(s);

for (i=0;i<N+1;i++)
  P[i]=(p->Compute)(p->Par,exp(x-limit+(double)i*h));


/*Finite Difference Cycle*/
for (i=1;i<=M;i++)
  {
    /*Init Rhs*/
    for(j=1;j<N;j++)
Rhs[j]=alpha1*P[j-1]+beta1*P[j]+gamma1*P[j+1];

    /*Multi-grid method*/
    MGM(l,P,Rhs,t,r,divid,sigma,N,M,theta);
  }
/*End Finite Difference Cycle*/

Index=(int) floor ((double)N/2.0);

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

/*Delta*/
*ptdelta=(P[Index+1]-P[Index-1])/(2.0*s*h);

/*Memory Desallocation*/
free(P);
free(Rhs);

return OK;
```

```
}



int CALC(FD_Multigrid_Euro)(void *Opt,void *Mod,Pricing
    Method *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 mult_euro1(ptMod->S0.Val.V_PDOUBLE,ptOpt->PayOff.
    Val.V_NUMFUNC_1,
        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_INT,Met->
    Par[2].Val.V_RGDOUBLE,
        &(Met->Res[0].Val.V_DOUBLE),&(Met->Res[1].Val.V_
    DOUBLE));
}



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

  if ((opt->EuOrAm). Val.V_BOOL==EURO)
    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=6;
      Met->Par[1].Val.V_INT2=128;
      Met->Par[2].Val.V_RGDOUBLE=0.5;


    }

  return OK;
}



PricingMethod MET(FD_Multigrid_Euro)=
{
  "FD_Multigrid_Euro",
  {{"Number of Grids",INT2,{100},ALLOW    },{"TimeStepNumb
    er",INT2,{100},ALLOW},{"Theta",RGDOUBLE,{100},ALLOW},{" ",
    PREMIA_NULLTYPE,{0},FORBID}},
  CALC(FD_Multigrid_Euro),
  {{"Price",DOUBLE,{100},FORBID},{"Delta",DOUBLE,{100},FORB
    ID} ,{" ",PREMIA_NULLTYPE,{0},FORBID}},
  CHK_OPT(FD_Multigrid_Euro),
  CHK_fdiff,
  MET(Init)
};
```

References