```
   Help
#include    "cdo.h"

/*Implied Copula*/



/* HUll && white  -fonctions utiles pour la méthode de     BASE CORRELATION */
/* &&  implied copula voir (perfect copula )*/

/* Hull & White implied copula */

/* Dans cette partie nous définirons dans un premiers temps
     nos n lambda
   implicite selon les valeurs max qu'on fixera et dans un
    second temps nous
   définirons la matrice du gradient utile pour résoudre
    notre problème
   d'optimisation voir Papier Hull& White implied Copula */

double          *lambdaimpl(const prod *produit,
                            const double *s1,
                            const double s2,
                            const int n,
                            const int choix)
{
  /* n définit le nombre d'intensité implicite que l'on veu
    t avoir,ce nombre
      dépend de l'utilisateur plus il est grand plus la
    calibration est
      meilleure */
  /* on définit l'intensité minimale et l'intensité maxima
    le de défaut qu'on
     peut avoir grâce aux données du marché */
  int k,i,j;
  int MAX_ITERATIONS=10;
  double ACCURACY=0.001;
  double *result;
  double x,x1,x2,y1,y2,f,f1,f2,xl,xh,rts,temp,dx,dxold,df;
  double lamb_min=0.0;
```

```
double lamb_max=0.18;
double V1;
double V2;
prod **prods;

/* On définit la somme de la valeur des contrats de nos
  six dérivés de
   crédit d'une part en considérant l'intensité minimale
  et d'une autre part
   en considérant l'intensité maximale */

V1=prix_contrat_CDS(produit,lamb_min,s2);
V2=prix_contrat_CDS(produit,lamb_max,s2);

prods=(prod**)malloc(5*sizeof(prod*));

for(i=0;i<5;i++){
  prods[i]=(prod*)malloc(sizeof(prod));
}
prods[0]->att=0.0;

if(choix==1){
  for(i=1;i<5;i++){
    prods[i-1]->det=prods[i]->att=0.03*i;
  }
  prods[4]->det=0.22;
}
else if(choix==2){
  prods[0]->det=prods[1]->att=0.03;
  prods[1]->det=prods[2]->att=0.07;
  prods[2]->det=prods[3]->att=0.1;
  prods[3]->det=prods[4]->att=0.15;
  prods[4]->det=0.3;
}



for(i=0;i<5;i++){
  prods[i]->maturite=produit->maturite;
  prods[i]->rate=produit->rate;
  prods[i]->recov=produit->recov;
```

```
  prods[i]->nb=produit->nb;
  prods[i]->nominal=produit->nominal;
}

for(i=0;i<5;i++){
  V1+=prix_contrat_CDO(prods[i],s1[i],0.0,lamb_min);
  V2+=prix_contrat_CDO(prods[i],s1[i],0.0,lamb_max);
}

/* on retournera les lambda implicites tels que ces derni
  ers couvrent
   uniformément [V1,V2] */

result=malloc(n*sizeof(double));


/*initialisation */

for(i=0;i<n;i++){

  result[i]=0.0;
}



for(i=1;i<n+1;i++){

  x1=lamb_min;
  x2=lamb_max;
  x=V1+(i-1)*(V2-V1)*1./(n-1);
  k=0;
  f1=prix_contrat_CDS(produit,x1,s2);
  f2=prix_contrat_CDS(produit,x2,s2);
  for(j=0;j<5;j++){
    f1+=prix_contrat_CDO(prods[j],s1[j],0.0,x1);
    f2+=prix_contrat_CDO(prods[j],s1[j],0.0,x2);
  }

  f1-=x;
  f2-=x;
```

```
if((f1*f2)>=0){

  if(f1==0)   result[i-1]=x1 ;
  if(f2==0)   result[i-1]=x2;
}



if(f1<0.0){
  xl=x1;
  xh=x2;
}
else {
  xh=x1;
  xl=x2;
}

rts=0.5*(x1+x2);
dxold=fabs(x2-x1);
dx=dxold;
f=prix_contrat_CDS(produit,rts,s2)-x;
y1=prix_contrat_CDS(produit,rts-ACCURACY,s2)-x;
y2=prix_contrat_CDS(produit,rts+ACCURACY,s2)-x;

for(j=0;j<5;j++){
  f+=prix_contrat_CDO(prods[j],s1[j],0,rts);
  y1+=prix_contrat_CDO(prods[j],s1[j],0,rts-ACCURACY);
  y2+=prix_contrat_CDO(prods[j],s1[j],0,rts+ACCURACY);
}

df=(y2-y1)/(2*ACCURACY);



do{

  if((((rts-xh)*df-f)*((rts-xl)*df-f)>0)||((fabs(2.0*f)
>fabs(dxold*df)))){
    dxold=dx;
    dx=0.5*(xh-xl);
    rts=xl+dx;
```

```
      if(xl==rts) break ;
    }
    else{
      dxold=dx;
      dx=f/df;
      temp=rts;
      rts-=dx;
      if(temp==rts) break;
    }
    if(fabs(dx)<ACCURACY)  break;
    f=prix_contrat_CDS(produit,rts,s2)-x;
    y1=prix_contrat_CDS(produit,rts-ACCURACY,s2)-x;
    y2=prix_contrat_CDS(produit,rts+ACCURACY,s2)-x;

    for(j=0;j<5;j++){
      f+=prix_contrat_CDO(prods[j],s1[j],0,rts);
      y1+=prix_contrat_CDO(prods[j],s1[j],0,rts-ACCURACY)
  ;
      y2+=prix_contrat_CDO(prods[j],s1[j],0,rts+ACCURACY)
  ;
    }

    df=(y2-y1)/(2*ACCURACY);

    if(f<0) xl=rts;
    else    xh=rts;

    k=k+1;

  }while(k<MAX_ITERATIONS);

  result[i-1]=rts;


}
for(i=0;i<5;i++){
  free(prods[i]);
}


free(prods);
```

```
  return (result) ;
}


/* Connaisant les valeurs des intensités implicites suscept
    ibles de bien "fiter" nos produits nous chercherons dans
    la suite à définir le
    poids de chaque intensité ,résoudre ce problème revient
     à résoudre un problème d'optimisation ! défissons la matr
    ice du gradient */



double **gradient(const prod* produit,
                  const double *s1,
                  const double s2,
                  const int n,
                  const int choix )
{

  double *d;
  double **A;
  double **a;
  double **c;
  int i,l,j;
  prod*  (*prods);
  double cte=0.5*0.000000001;

  A=(double**)malloc((n)*sizeof(double*));
  a=(double**)malloc(n*sizeof(double*));
  c=(double**)malloc(n*sizeof(double*));
  d=(double*)malloc(n* sizeof(double));

  for(i=0;i<n;i++){
    A[i]=(double*)malloc((n)*sizeof(double));
    c[i]=(double*)malloc(n*sizeof(double));
  }
  for(i=0;i<n;i++){
    a[i]=(double*)malloc(6*sizeof(double));
```

```
}

for(i=0;i<n;i++){
  d[i]=i*0.1*1./(n-1);
}

/*Initilisation de la matrice du gradient   */

for( i=0;i<n;i++){
  for(j=0;j<n;j++){
    A[i][j]=0;
  }
}

prods=(prod**)malloc(5*sizeof(prod));

for(i=0;i<5;i++){
  prods[i]=(prod*)malloc(sizeof(prod));
}
prods[0]->att=0.0;


if(choix==1){
  for(i=1;i<5;i++){
    prods[i-1]->det=prods[i]->att=0.03*i;
  }
  prods[4]->det=0.22;
}

else {
  prods[0]->det=prods[1]->att=0.03;
  prods[1]->det=prods[2]->att=0.07;
  prods[2]->det=prods[3]->att=0.1;
  prods[3]->det=prods[4]->att=0.15;
  prods[4]->det=0.3;
}


for(i=0;i<5;i++){
  prods[i]->maturite=produit->maturite;
  prods[i]->rate=produit->rate;
```

```
  prods[i]->recov=produit->recov;
  prods[i]->nb=produit->nb;
  prods[i]->nominal=produit->nominal;
}


for(i=0;i<n;i++){

  a[i][5]=prix_contrat_CDS(produit,d[i],s2);

  for(j=0;j<5;j++){

    a[i][j]=prix_contrat_CDO(prods[j],s1[j],0,d[i]);
  }
}

for( i=0;i<n;i++){
  for(j=0;j<n;j++){
    c[i][j]=0;
    for(l=0;l<6;l++){
      c[i][j]=c[i][j]+a[i][l]*a[j][l];
    }
  }
}

/*Définition de la matrice obtenue en prenant le gradient
    de notre fonction à optimiser voir perfect copula on
  prend cste de regularisation=0.5*/

/*Première ligne-ligne 0*/

A[0][0]=2*c[0][0]+(2/(d[2]-d[0]))*(cte/0.5);
A[0][1]=2*c[0][1]-(4/(d[2]-d[0]))*(cte/0.5);
A[0][2]=2*c[0][2]+(2/(d[2]-d[0]))*(cte/0.5);

for(j=3;j<n;j++){
  A[0][j]=2*c[0][j];
}

/*Deuxième ligne-ligne 1*/
```

```
A[1][0]=2*c[1][0]-(4/(d[2]-d[0]))*(cte/0.5);
A[1][1]=2*c[1][1]+((2/(d[3]-d[1]))+(8/(d[2]-d[0])))*(cte/
  0.5);
A[1][2]=2*c[1][2]+((-4/(d[2]-d[0]))-(4/(d[3]-d[1])))*(ct
  e/0.5);
A[1][3]=2*c[1][3]+(2/(d[3]-d[1]))*(cte/0.5);

for(j=4;j<n;j++){
  A[1][j]=2*c[1][j];
}


/*Ligne 2 à ligne n-3*/
for(i=2;i<n-2;i++){
  for (l=0;l<i-2;l++){
    A[i][l]=2*c[i][l];
  }

  A[i][i-2]=2*c[i][i-2]+((2/(d[i]-d[i-2])))*(cte/0.5);
  A[i][i-1]=2*c[i][i-1]+((-4/(d[i+1]-d[i-1]))-(4/(d[i]-d[
  i-2])))*(cte/0.5);
  A[i][i]=2*c[i][i]+((2/(d[i]-d[i-2]))+8/((d[i+1]-d[i-1])
  )+(2/(d[i+2]-d[i])))*(cte/0.5);
  A[i][i+1]=2*c[i][i+1]+((-4/(d[i+1]-d[i-1]))-(4/(d[i+2]-
  d[i])))*(cte/0.5);
  A[i][i+2]=2*c[i][i+2]+(2/(d[i+2]-d[i]))*(cte/0.5);

  for (l=i+3;l<n;l++){
    A[i][l]=2*c[i][l];
  }
}

/*Ligne n-2*/
A[n-2][n-1]=2*c[n-2][n-1]-(4/(d[n-1]-d[n-3]))*(cte/0.5);
A[n-2][n-2]=2*c[n-2][n-2]+(8/(d[n-1]-d[n-3])+2/(d[n-2]-d[
  n-4]))*(cte/0.5);
A[n-2][n-3]=2*c[n-2][n-3]+(-4/(d[n-2]-d[n-4])-4/(d[n-1]-
  d[n-3]))*(cte/0.5);
A[n-2][n-4]=2*c[n-2][n-4]+(2/(d[n-2]-d[n-4]))*(cte/0.5);

for(j=0;j<n-4;j++){
```

```
    A[n-2][j]=2*c[n-2][j];
  }

  /*Ligne n-1*/

  A[n-1][n-1]=2*c[n-1][n-1]+(2/(d[n-1]-d[n-3]))*(cte/0.5);
  A[n-1][n-2]=2*c[n-1][n-2]-(4/(d[n-1]-d[n-3]))*(cte/0.5);
  A[n-1][n-3]=2*c[n-1][n-3]+(2/(d[n-1]-d[n-3]))*(cte/0.5);

  for(j=0;j<n-3;j++){
    A[n-1][j]=2*c[n-1][j];
  }

  for(i=0;i<n;i++){
    free(c[i]);
    free(a[i]);
  }
  free(c);
  free(a);

  return A;

}


double              *probaimpl(const prod* produit,
                               const double *s1,
                               const double s2,
                               const int n,
                               const int choix)
{

  /* Dans cette partie on cherche à résoudre le problème d'
    optimisation voir
     HUll & White Perfect Copula pour déterminer les probab
    ilités associées à
     chaque intensité on utilisera la méthode du gradient
    conjugué */
  int i,l;
  int j=0;
  double *P;
```

```
double *g;
double *g1;
double epsilon=0.000001;
int Max_iterations=100;
double *w;
double *v;
double s;
double *P1;
double **A;
double sum1;
double sum2;
double *lamb;

A=malloc(n*sizeof(double*));
P=malloc(n*sizeof(double));
g=malloc(n*sizeof(double));
w=malloc(n*sizeof(double));
v=malloc(n*sizeof(double));
P1=malloc(n*sizeof(double));
g1=malloc(n*sizeof(double));

/*Initialisation du vecteur de proba*/
for(i=0;i<n;i++){
  A[i]=malloc(n*sizeof(double));
}

for(i=0;i<n;i++){
  P[i]=1./n;
  g[i]=1./n;
  g1[i]=1./n;
}


lamb=malloc(n*sizeof(double));
for(i=0;i<n;i++){
  lamb[i]=i*0.1*1./(n-1);
}

/*Défintion du vecteur normal*/

A=gradient(produit,s1,s2,n,choix);
```

```
do{
  for(i=0;i<n;i++){
    w[i]=0;
    for(l=0;l<n;l++){
      w[i]+=A[i][l]*P[l];
    }
  }

  s=0;
  sum1=0;
  sum2=0;
  for(i=0;i<n;i++){
    sum1+=P[i]*w[i];
    sum2+=g[i]*g[i];
  }

  for(i=0;i<n;i++){
    g1[i]=g[i]-sum2*w[i]/sum1;
    s+=g1[i]*g1[i];
  }



  for(i=0;i<n;i++){
    P1[i]=g1[i]+s*P[i]/sum2;
  }
  s=0;
  sum1=0;

  for(i=0;i<n;i++){
    if (P1[i]<0) P1[i]=0.00001;
    s+=P1[i];
  }

  for(i=0;i<n;i++){
    P1[i]=P1[i]/s;
    sum1+=(P1[i]-P[i])*(P1[i]-P[i]);
  }
  if(sum1<epsilon)  break;
```

```
    else{

      for(i=0;i<n;i++){
        P[i]=P1[i];
      }
    }
    j+=1;

  }while(j<Max_iterations);

  for(i=0;i<n;i++){
    free(A[i]);
  }
  free(A);
  free(P);
  free(g);
  free(g1);
  free(v);
  free(w);
  free(lamb);

  return P1;

}


double pay_leg_impl(const prod* produit,const double *p,
    const int n){
  int i;
  double s=0;
  double *lamb;
  lamb=malloc(n*sizeof(double));

  for(i=0;i<n;i++){
    lamb[i]=i*0.1*1./(n-1);
  }
  for(i=0;i<n;i++){
    s+=p[i]*payment_leg_CDO(produit,0,lamb[i]);
  }

  free(lamb);
```

```
  return s;
}

double dl_leg_impl(const prod* produit,const double *p,int
    n){
  int i;
  double s=0;
  double *lamb;
  lamb=malloc(n*sizeof(double));

  for(i=0;i<n;i++){
    lamb[i]=i*0.1*1./(n-1);
  }
  for(i=0;i<n;i++){
    s+=p[i]*loss(produit,0,lamb[i]);
  }
  free(lamb);

  return s;
}
```

References