Help

```
#include    "
href../../../mod/copula/copula_stdndc/price_cdo_h_src.pdfcdo.h"
#include "pnl/pnl_cdf.h"

/**intensité implicite déduite des données du marché sur le CDS on utilise la
   méthode de Newton pour pouvoir calculer le zero contrat CDS */

double       intens1(const prod *produit,
                     const double s)
{
  double xlow = 0.;
  double F;
  double F1;
  double F2;
  int MAX_ITERATIONS = 10;
  double ACCURACY = 0.001;
  double x = xlow;
  int i;
  for (i = 0; i < MAX_ITERATIONS; i++)
    {
      F = spread_CDS(produit, x) - s;
      if (fabs(F) < ACCURACY) return (x);
      F1 = spread_CDS(produit, x + ACCURACY) - s;
      F2 = spread_CDS(produit, x - ACCURACY) - s;
      x = x - (2 * ACCURACY * F) / (F1 - F2);
    }
  return (0);
}

double        intens2(const prod *produit,
                      const double s)
{

  int MAX_ITERATIONS = 10;
  double xl, xh;
  double x1 = 0;
  double x2 = 0.5;
  double f1 = prix_contrat_CDS(produit, x1, s);
  double f2 = prix_contrat_CDS(produit, x2, s);
```

```
double dx, f, rts, df, dxold, temp;
double ACCURACY = 0.00001;
int j;
if (f1 * f2 > 0) return (0);
if (f1 == 0) return (x1);
if (f2 == 0) return (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, s);
df = (prix_contrat_CDS(produit, rts + ACCURACY, s) - prix_contrat_CDS(produit,
for (j = 0; j < MAX_ITERATIONS; j++)
  {
    if ((((rts - xh)*df - f) * ((rts - xl)*df - f) > 0) || ((fabs(2.0 * f) > f
      {
        dxold = dx;
        dx = 0.5 * (xh - xl);
        rts = xl + dx;
        if (xl == rts) return (rts);
      }
    else
      {
        dxold = dx;
        dx = f / df;
        temp = rts;
        rts -= dx;
        if (temp == rts) return (rts);
      }
    if (fabs(dx) < ACCURACY) return (rts);
    f = prix_contrat_CDS(produit, rts, s);
```

```
      df = (prix_contrat_CDS(produit, rts + ACCURACY, s) - prix_contrat_CDS(prod
      if (f < 0) xl = rts;
      else    xh = rts;
    }
  return (0);
}

/**Méthode de Newton + dichotomie  pour deteminer les tranches-correlations **/

/**On suppose qu'on a les données sur les 5 tranches + l'index on cherche à
   determiner la correlation implicite sur  chaque tranche par inversion de la
   formule su spread **/


double     *tranche_correl(const prod *produit,
                           const double *s1,
                           const double s2, int choix)
{
  /** s1 tableau des spreads des 5 tranches ,s2 spread de l'index CDS **/

  double *result = NULL;
  prod*  (*prods);
  int i, k, j;
  int MAX_ITERATIONS = 10;
  double ACCURACY = 0.000001;
  double x1, x2, f1, xl, xh, dx, f, rts, df, dxold, temp;
  double lambda = 0;


  lambda = intens2(produit, s2);

  /** on définit les cinq produits CDO dont on cherche la correl !**/

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



result = malloc(5 * sizeof(double));

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

    x1 = 0.0;
    x2 = 0.99;
    f1 = spread_CDO(prods[k], x1, lambda) - s1[k];
    //f2=spread_CDO(prods[k],x2,lambda)-s1[k];


    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 = spread_CDO(prods[k], rts, lambda) - s1[k];
df = (spread_CDO(prods[k], rts + ACCURACY, lambda) - spread_CDO(prods[k],
j = 0;
do
  {
    j = j + 1;
    if ((((rts - xh)*df - f) * ((rts - xl)*df - f) > 0) || ((fabs(2.0 * f)
      {
        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 = spread_CDO(prods[k], rts, lambda) - s1[k];
    df = (spread_CDO(prods[k], rts + ACCURACY, lambda) - spread_CDO(prods[
    if (f < 0) xl = rts;
    else    xh = rts;

  }
while (j < MAX_ITERATIONS);
```

```
      result[k] = rts * rts;
    }
  for (i = 0; i < 5; i++)
    {
      free(prods[i]);
    }
  free(prods);

  return (result);


}

/** Base correlation voir papier Hull and White perfect copula **/


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

  /** s1 sera un tableau dont les valeurs représenteront les spread des 6 tranch
      s2 sera le spread du CDS**/


  double *result;

  double x1, x2, f1, xl, xh, dx, f, rts, df, dxold, temp, c;
  int MAX_ITERATIONS = 10;
  double ACCURACY = 0.0001;
  prod* (*prods);
  prod* (*nvprods);

  int i, k, j;

  double lambda = intens2(produit, s2);


  /** on définit les cinq produits CDO dont on cherche la correl !**/
```

```
prods = malloc(5 * sizeof(prod));
for (i = 0; i < 5; i++)
  {
    prods[i] = malloc(sizeof(prod));

  }
nvprods = malloc(5 * sizeof(prod));
for (i = 0; i < 4; i++)
  {
    nvprods[i] = 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;

    for (i = 0; i < 4; i++)
      {
        nvprods[i]->att = 0.0;

        if (i != 3)           nvprods[i]->det = 0.03 * (i + 2);
        else if (i == 3)      nvprods[i]->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;
```

```
    nvprods[0]->att = nvprods[1]->att = nvprods[2]->att = nvprods[3]->att = 0;
    nvprods[0]->det = 0.07;
    nvprods[1]->det = 0.1;
    nvprods[2]->det = 0.15;
    nvprods[3]->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;

  }

/** on doit définir quatres differents types de prod :[0,6],[0,9],[0,12],[0,22

for (i = 0; i < 4; i++)
  {
    nvprods[i]->maturite = produit->maturite;
    nvprods[i]->rate = produit->rate;
    nvprods[i]->recov = produit->recov;
    nvprods[i]->nb = produit->nb;
    nvprods[i]->nominal = produit->nominal;

  }


result = malloc(5 * sizeof(double));
result = tranche_correl(produit, s1, s2, choix);

for (i = 0; i < 5; i++)
  {
    result[i] = sqrt(result[i]);
  }

c = 0;
```

```
for (k = 1; k < 5; k++)
  {
    x1 = 0.0; /**la correl est compris entre 0 et 1 **/
    x2 = 0.99;


    if (k == 1)
      {
        for (j = 0; j <= k; j++)
          {
            c += loss(prods[j], result[j], lambda);

          }

      }
    else
      {
        c += loss(prods[k], result[k], lambda);
      }


    f1 = loss(nvprods[k - 1], x1, lambda) - c;
    //f2=loss(nvprods[k-1],x2,lambda)-c;


    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 = loss(nvprods[k - 1], rts, lambda) - c;
    df = (loss(nvprods[k - 1], rts + ACCURACY, lambda) - loss(nvprods[k - 1],
```

```
    j = 0;
    do
      {
        j = j + 1;
        if ((((rts - xh)*df - f) * ((rts - xl)*df - f) > 0) || ((fabs(2.0 * f)
          {
            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 = loss(nvprods[k - 1], rts, lambda) - c;
        df = (loss(nvprods[k - 1], rts + ACCURACY, lambda) - loss(nvprods[k -
        if (f < 0) xl = rts;
        else    xh = rts;

      }
    while (j < MAX_ITERATIONS);

    result[k] = rts * rts;
  }


result[0] = result[0] * result[0];

for (i = 0; i < 5; i++)
  {
    free(prods[i]);
  }
for (i = 0; i < 4; i++)
  {
    free(nvprods[i]);
```

```
    }

  free(prods);
  free(nvprods);

  return (result);

}

/**Partie1 evaluation du prix du contrat du spread et du loss sur une tranche de

double perte(const prod *produit, const double f_t)
{

  double a = produit->att;
  double b = produit->det;
  int    M = produit->nb;
  double K = produit->nominal;
  double R = produit->recov;
  int i, j;

  double *q;
  double s;

  q = malloc((M + 1) * sizeof(double));

  if ((f_t == 1) || (f_t == 0))
    {
      for (i = 0; i < M + 1; i++)
        {
          q[i] = 0;
        }
    }
  else
    {
      q[0] = exp(M * log(1 - f_t));
      q[M] = exp(M * log(f_t));

      for (i = 1; i < M; i++)
        {
          q[i] = 1;
```

```
          for (j = 1; j <= i; j++)
            {
              q[i] = q[i] * (M - j + 1) / (j);
            }
          q[i] = q[i] * exp(i * log(f_t)) * exp((M - i) * log(1 - f_t));
        }
    }


  s = 0;

  for (j = 0; j < M + 1; j++)
    {
      s = s + q[j] * (MAX(K * (1 - R) * j - a, 0) - MAX(K * (1 - R) * j - b, 0))
    }

  free(q);

  return s;

}


double cond_prob1(const double lambda, const double t, const double v, const dou
{


  double p;
  double a, u_rho, g_rho;
  double f_t = 0;
  g_rho = sqrt(1 - rho * rho);
  u_rho = rho / g_rho;
  f_t = 1 - exp(-lambda * t);

  a = (pnl_inv_cdfnor(f_t)) / g_rho;

  p = cdf_nor(a - u_rho * v);

  return p;

}
```

```
/**valeur du contrat CDO pour un spread fixé et pour une proba de défaillance fi
   x représente le facteur ,conditionnement à ce facteur les défauts sont indépe


double eval_contrat(const prod *produit, const double s, const double rho, const
{
  double b = produit->det;
  double a = produit->att;
  int    T = produit->maturite;
  double r = produit->rate;
  int n = 4 * T;
  double *t;
  double *q;
  double *z;
  double V;
  int i;
  double A = 0, B = 0, C = 0;


  t = malloc((n + 1) * sizeof(double));
  q = malloc((n + 1) * sizeof(double));
  z = malloc((n + 1) * sizeof(double));


  for (i = 0; i < n + 1; i++)
    {
      t[i] = i * 0.25;
      q[i] = cond_prob1(lambda, t[i], v, rho);
      z[i] = perte(produit, q[i]);

    }

  for (i = 1; i < n + 1; i++)
    {
      A += (t[i] - t[i - 1]) * (b - a - z[i]) * exp(-r * t[i]);
      B += (t[i] - t[i - 1]) * (z[i] - z[i - 1]) * exp(-(t[i] + t[i - 1]) * 0.5
      C += (z[i] - z[i - 1]) * exp(-(t[i] + t[i - 1]) * 0.5 * r);
    }
  V = s * (A + B) - C;
  free(t);
```

```
  free(q);
  free(z);
  return V;
}

double xgk[12] = {                0.0,
                                  0.995657163025808080735527280 70,
                                  0.973906528517171720079640121 0,
                                  0.930157491355708226001207180 10,
                                  0.865063366688984510732096688 40,
                                  0.780817726586416897063717578 30,
                                  0.679409568299024406234327365 10,
                                  0.562757134668604683339000993 0,
                                  0.433395394129247190799265943 20,
                                  0.294392862701460198131126603 10,
                                  0.148874338981631210884826001 10,
                                  0.0
                  };
double wgk[12] = {                0.0,
                                  0.011694638867371874278064396060,
                                  0.032558162307964727478818972460,
                                  0.054755896574351996031381300240,
                                  0.075039674810919952767043140920,
                                  0.093125454583697605535065465080,
                                  0.109387158802297641899210590 30,
                                  0.123491976262065851077958109 80,
                                  0.134709217311473325928054001 80,
                                  0.142775938577060080797094273 10,
                                  0.147739104901338491374841516 00,
                                  0.149445554002916905664936468 40
                  };


double prix_contrat_CDO(const prod *produit, const double s, const double rho, c
{
  double b1 = produit->det;
  double K = produit->nominal;
  int   M = produit->nb;
  double result;
  double a = -4;
  double b = 4;
```

```
  double s1 = s / 10000;
  double absc, resk;
  double centre = (a + b) * 0.5;
  double h = (b - a) * 0.5;
  int j;
  double fval1, fval2, fsum;
  double fc = pnl_normal_density(centre) * eval_contrat(produit, s1, rho, lambda
  resk = wgk[11] * fc;
  for (j = 1; j <= 10; j++)
    {
      absc = h * xgk[j];
      fval1 = pnl_normal_density(centre - absc) * eval_contrat(produit, s1, rho,
      fval2 = pnl_normal_density(centre + absc) * eval_contrat(produit, s1, rho,
      fsum = fval1 + fval2;
      resk = resk + wgk[j] * fsum;
    }
  if (b1 > 0.03)
    {
      result = resk * h;
    }
  else
    {

      return result = 0.05 * payment_leg_CDO(produit, rho, lambda) + 0.01 * s *

    }
  return result;
}




double prix_contrat_CDS(const prod *produit, const double lambda, const double s
{

  double K = produit->nominal;
  int    T = produit->maturite;
  double r = produit->rate;
  double R = produit->recov;
  double s1 = s * 0.0001;
  int n = 4 * T;
```

```
  double *q;
  double *p;
  double *t;
  int i;
  double A = 0, B = 0, C = 0;


  q = malloc((n + 1) * sizeof(double));
  p = malloc((n + 1) * sizeof(double));
  t = malloc((n + 1) * sizeof(double));
  for (i = 0; i < n + 1; i++)
    {
      t[i] = i * 0.25;
      q[i] = 1 - exp(-lambda * t[i]);
      p[i] = (1 - q[i]) * K;

    }
  for (i = 1; i < n + 1; i++)
    {
      A += (t[i] - t[i - 1]) * p[i] * exp(-r * t[i]);
      B += 0.5 * (t[i] - t[i - 1]) * (p[i - 1] - p[i]) * exp(-r * (t[i - 1] + t[
      C += (1 - R) * (p[i - 1] - p[i]) * exp(-r * 0.5 * (t[i] + t[i - 1]));
    }
  free(t);
  free(q);
  free(p);

  return s1 * (A + B) - C;

}

/**spread_CDS**/

double spread_CDS(const prod *produit, const double lambda)
{
  double K = produit->nominal;
  int    T = produit->maturite;
  double r = produit->rate;
  double R = produit->recov;
  int n = 4 * T;
  double *q;
```

```
  double *p;
  double *t;
  int i;
  double A = 0, B = 0, C = 0;

  q = malloc((n + 1) * sizeof(double));
  p = malloc((n + 1) * sizeof(double));
  t = malloc((n + 1) * sizeof(double));

  for (i = 0; i < n + 1; i++)
    {
      t[i] = i * 0.25;
      q[i] = 1 - exp(-lambda * t[i]);
      p[i] = (1 - q[i]) * K;

    }
  for (i = 1; i < n + 1; i++)
    {
      A += (t[i] - t[i - 1]) * p[i] * exp(-r * t[i]);
      B += 0.5 * (t[i] - t[i - 1]) * (p[i - 1] - p[i]) * exp(-r * (t[i - 1] + t[
      C += (1 - R) * (p[i - 1] - p[i]) * exp(-r * 0.5 * (t[i] + t[i - 1]));
    }

  free(q);
  free(p);
  free(t);

  return 10000 * C / (A + B);
}


/**Pour determiner C(a,b,v) utile pour les bases correlations **/


double eval_loss(const prod *produit, const double rho, const double lambda , co
{
  int    T = produit->maturite;
  double r = produit->rate;
  int n = 4 * T;
  double *t;
  double *q;
```

```
  double *z;
  int i;
  double C = 0;

  t = malloc((n + 1) * sizeof(double));
  q = malloc((n + 1) * sizeof(double));
  z = malloc((n + 1) * sizeof(double));

  for (i = 0; i < n + 1; i++)
    {
      t[i] = i * 0.25;
      q[i] = cond_prob1(lambda, t[i], v, rho);
      z[i] = perte(produit, q[i]);
    }

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

      C += (z[i] - z[i - 1]) * exp(-(t[i] + t[i - 1]) * 0.5 * r);
    }
  free(t);
  free(q);
  free(z);
  return C;
}

/**Pour evaluer C(a,b) pour les bases correlations ***/

double loss(const prod *produit, const double rho, const double lambda)
{
  double fsum, fval1, fval2;
  double a = -4;
  double b = 4;
  double absc, resk;
  int j;
  double centre = 0;
  double h = 0;
  double fc = 0;

  centre = (a + b) * 0.5;
  h = (b - a) * 0.5;
```

```
  fc = pnl_normal_density(centre) * eval_loss(produit, rho, lambda, centre);
  resk = wgk[11] * fc;
  for (j = 1; j <= 10; j++)
    {
      absc = h * xgk[j];
      fval1 = pnl_normal_density(centre - absc) * eval_loss(produit, rho, lambda
      fval2 = pnl_normal_density(centre + absc) * eval_loss(produit, rho, lambda
      fsum = fval1 + fval2;
      resk = resk + wgk[j] * fsum;
    }
  return resk * h;

}

/**Evaluation du spread_CDO **/


double payleg_cond_CDO(const prod *produit, const double rho, const double lambd
{
  double a = produit->att;
  double b = produit->det;
  int    T = produit->maturite;
  double r = produit->rate;

  int i;
  int n = 4 * T;
  double *t;
  double *q;
  double *z;
  double V;
  double A = 0, B = 0;

  t = malloc((n + 1) * sizeof(double));
  q = malloc((n + 1) * sizeof(double));
  z = malloc((n + 1) * sizeof(double));

  for (i = 0; i < n + 1; i++)
    {
      t[i] = i * 0.25;
      q[i] = cond_prob1(lambda, t[i], v, rho);
      z[i] = perte(produit, q[i]);
```

```
    }

  for (i = 1; i < n + 1; i++)
    {
      A += (t[i] - t[i - 1]) * (b - a - z[i]) * exp(-r * t[i]);
      B += (t[i] - t[i - 1]) * (z[i] - z[i - 1]) * exp(-(t[i] + t[i - 1]) * 0.5
    }

  V = (A + B);

  free(t);
  free(q);
  free(z);

  return V;
}


/**spread**/



double payment_leg_CDO(const prod *produit, const double rho, const double lambd
{
  double a = -4;
  double b = 4;
  double fval1, fval2, fsum;
  double absc, resk;
  double centre = (a + b) * 0.5;
  double h = (b - a) * 0.5;
  double fc = pnl_normal_density(centre) * payleg_cond_CDO(produit, rho, lambda,
  int j;
  resk = wgk[11] * fc;
  for (j = 1; j <= 10; j++)
    {
      absc = h * xgk[j];
      fval1 = pnl_normal_density(centre - absc) * payleg_cond_CDO(produit, rho,
      fval2 = pnl_normal_density(centre + absc) * payleg_cond_CDO(produit, rho,
      fsum = fval1 + fval2;
      resk = resk + wgk[j] * fsum;
    }
```

```
  return resk * h;

}



double spread_CDO(const prod *produit, const double rho, const double lambda)
{

  double b = produit->det;
  int    M = produit->nb;
  double K = produit->nominal;


  if (b > 0.03)
    {
      return 10000 * (loss(produit, rho, lambda)) / (payment_leg_CDO(produit, rh
    }
  return (100 * (loss(produit, rho, lambda) - 0.05 * (payment_leg_CDO(produit, r
  /**Upfront pour la tranche equity et spread pour les autres tranches **/

}

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

  int i, j = 0, k = 0;
  double a = produit->att;
  double b = produit->det;
  prod *nvprod1;
  prod *nvprod2;
  double *rho_B;  /**pour recupérer les bases correlatons */
  double C, C1, C2; /** pour déterminer la perte sur la tranche en utilisant la
  double lambda;
  double rho, rho1 = 0, rho2 = 0;
  double  x1, x2, f, f1, f2, xl, xh, rts, temp, dx, dxold, df;
  double ACCURACY = 0.00001;
  int MAX_ITERATIONS = 20;
```

```
nvprod1 = malloc(sizeof(prod));
nvprod2 = malloc(sizeof(prod));
lambda = intens1(produit, s2);

rho_B = malloc(5 * sizeof(double));
rho_B = Base_correl(produit, s1, s2, choix);

for (i = 0; i < 5; i++)
  {
    rho_B[i] = sqrt(rho_B[i]);
  }


if (choix == 1)
  {
    for (i = 0; i < 4; i++)
      {
        if (a == 0) j = 0;
        if ((0.03 * i <= a) && (0.03 * (i + 1) > a))  j = i;
        if ((0.03 * i <= b) && (0.03 * (i + 1) > b))  k = i;
      }

    if ((0.12 <= a) && (0.22 > a))          j = 4;
    if ((0.12 <= b) && (0.22 > b))          k = 4;
    if (0.22 <= a)                       j = 5;
    if (0.22 <= b)                       k = 5;

    nvprod1->maturite = nvprod2->maturite = produit->maturite;
    nvprod1->rate = nvprod2->rate = produit->rate;
    nvprod1->recov = nvprod2->recov = produit->recov;
    nvprod1->nb = nvprod2->nb = produit->nb;
    nvprod1->nominal = nvprod2->nominal = produit->nominal;
    nvprod1->att = nvprod2->att = 0.0;
    nvprod1->det = a;
    nvprod2->det = b;



    /** Interpolation pour trouver les bases correl des tranches non standards
```

```
  /** On connait la base-correl de la tranche [0,0.03],on déduira ainsi les
      autres bases correl de cette correl initiale**/

  if (j == 0)
    {
      rho1 = rho_B[j] + (rho_B[j] / 0.03) * (a - 0.03);
    }
  else if (j == 4)
    {
      rho1 = rho_B[j - 1] + ((rho_B[j] - rho_B[j - 1]) / (0.1)) * (a - 0.03
    }
  else
    {
      rho1 = rho_B[j - 1] + ((rho_B[j] - rho_B[j - 1]) / (0.03)) * (a - 0.03
    }
  if (j == 5)
    {
      rho1 = rho_B[j - 1] + (rho_B[j - 1] / 0.22) * (a - 0.22);
    }


  if (k == 0)
    {
      rho2 = rho_B[k] + (rho_B[k] / (0.03)) * (b - 0.03);
    }
  else if (k != 4)
    {
      rho2 = rho_B[k - 1] + ((rho_B[k] - rho_B[k - 1]) / (0.03)) * (b - 0.03
    }
  else
    {
      rho2 = rho_B[k - 1] + ((rho_B[k] - rho_B[k - 1]) / (0.1)) * (b - 0.03
    }


  if (k == 5)
    {
      rho2 = rho_B[k - 1] + (rho_B[k - 1] / 0.22) * (b - 0.22);
    }
}
```

```
else
  {
    if (a == 0) j = 0;
    if ((a >= 0.03) && (0.07 > a))        j = 1;
    if ((a >= 0.07) && (0.1 > a))         j = 2;
    if ((a >= 0.1) && (0.15 > a))         j = 3;
    if ((a >= 0.15) && (0.3 > a))         j = 4;
    if (0.3 <= a)                      j = 5;

    if ((b > 0.03) && (0.07 >= b))        k = 1;
    if ((b > 0.07) && (0.1 >= b))         k = 2;
    if ((b > 0.1) && (0.15 >= b))         k = 3;
    if ((b > 0.15) && (0.3 >= b))         k = 4;
    if (0.3 <= b)                      k = 5;
    nvprod1->maturite = nvprod2->maturite = produit->maturite;
    nvprod1->rate = nvprod2->rate = produit->rate;
    nvprod1->recov = nvprod2->recov = produit->recov;
    nvprod1->nb = nvprod2->nb = produit->nb;
    nvprod1->nominal = nvprod2->nominal = produit->nominal;
    nvprod1->att = nvprod2->att = 0.0;
    nvprod1->det = a;
    nvprod2->det = b;


    /** Interpolation pour trouver les bases correl des tranches non standards
    /** On connait la base-correl de la tranche [0,0.03],on déduira ainsi les


    if (j == 0) rho1 = rho_B[j] + (rho_B[j] / 0.03) * (a - 0.03);
    if (j == 1) rho1 = rho_B[j - 1] + ((rho_B[j] - rho_B[j - 1]) / (0.04)) * (
    if (j == 2) rho1 = rho_B[j - 1] + ((rho_B[j] - rho_B[j - 1]) / (0.03)) * (
    if (j == 3) rho1 = rho_B[j - 1] + ((rho_B[j] - rho_B[j - 1]) / (0.05)) * (
    if (j == 4) rho1 = rho_B[j - 1] + ((rho_B[j] - rho_B[j - 1]) / (0.15)) * (
    if (j == 5) rho1 = rho_B[j - 1] + (rho_B[j - 1] / 0.3) * (a - 0.3);


    if (k == 0) rho2 = rho_B[k] + (rho_B[k] / (0.03)) * (b - 0.03);
    if (k == 1) rho2 = rho_B[k - 1] + ((rho_B[k] - rho_B[k - 1]) / (0.04)) * (
    if (k == 2) rho2 = rho_B[k - 1] + ((rho_B[k] - rho_B[k - 1]) / (0.03)) * (
    if (k == 3) rho2 = rho_B[k - 1] + ((rho_B[k] - rho_B[k - 1]) / (0.05)) * (
    if (k == 4) rho2 = rho_B[k - 1] + ((rho_B[k] - rho_B[k - 1]) / (0.15)) * (
```

```
    if (k == 5) rho2 = rho_B[k - 1] + (rho_B[k - 1] / 0.3) * (b - 0.3);


  }

/** perte sur la tranche [a,b]**/

C1 = loss(nvprod1, rho1, lambda);
C2 = loss(nvprod2, rho2, lambda);
C = C2 - C1;
x1 = 0.0;
x2 = 0.99;

f1 = loss(produit, x1, lambda) - C;
f2 = loss(produit, x2, lambda) - C;

/**tranche correl sur [a,b]**/

if (f1 == 0) rho = x1;
if (f2 == 0) rho = 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 = loss(produit, rts, lambda) - C;
df = (loss(produit, rts + ACCURACY, lambda) - loss(produit, rts - ACCURACY, la
i = 0;
do
  {
```

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

      f = loss(produit, rts, lambda) - C;
      df = (loss(produit, rts + ACCURACY, lambda) - loss(produit, rts - ACCURACY



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

      i = i + 1;

    }
  while (i < MAX_ITERATIONS);

  rho = rts;

  free(rho_B);
  free(nvprod1);
  free(nvprod2);

  return rho;
}

double pay_leg_base(const prod *produit,
```

```
                    const double *s1,
                    const double s2,
                    const int choix)
{
  double lambda = intens1(produit, s2);
  double rho = rhobase(produit, s1, s2, choix);
  return (payment_leg_CDO(produit, rho, lambda));
}

double dl_leg_base(const prod *produit,
                   const double *s1,
                   const double s2,
                   const int choix)
{
  double lambda = intens1(produit, s2);
  double rho = rhobase(produit, s1, s2, choix);
  return (loss(produit, rho, lambda));
}
```