Help

```
#include  "
href../../../mod/doublehes1d/doublehes1d_std/doublehes1d_std_h_src.pdfhes1d_std.
#include "pnl/pnl_mathtools.h"
#include "pnl/pnl_tridiag_matrix.h"
#include "pnl/pnl_finance.h"


#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2013+2) //The "#els

static int CHK_OPT(FD_IkonenToivanen_Heston)(void *Opt, void *Mod)
{
  return NONACTIVE;
}

int CALC(FD_IkonenToivanen_Heston)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}

#else

/*
 *  main.cpp
 *  IT
 *
 *  Created by Ludovic Goudenège on 03/08/12.
 *  Copyright 2012 __MyCompanyName__. All rights reserved.
 *
 */

static int size_grid_generation_IT(double X, double E, int mt,
                                   double kappa, double rho, double gamma, doubl
{
  int i;
  double temp = sqrt(kappa - 1.0);
  double a = temp / (X - E) * (atan(temp) + atan(E * temp / (X - E))) / ((double
  double ctilde = (X - E) * (X - E) / (kappa - 1.0) + E * E; // ctilde := c/a
  double pointx;
```

```
  // If rho<0 we use the grid for -rho.
  double absrho = ABS(rho);

  pointx = 0.0; // x_0
  pointx = x_1; // x_1 I&T said that it should be h/10.0
  i = 1;
  while (pointx < X)
    {
      pointx = pointx
               + MIN(
                 MAX(
                   a * (pointx * pointx - 2 * E * pointx + ctilde),
                   (1.1 / (1.0 - absrho / gamma * h)) * (absrho / gamma) * (h *
                 ),
                 0.9 * (h / absrho) * (pointx / gamma)
               );
      i++;
    }
  return i;
}


static  void grid_generation_IT(double *pointsx, double X, double E, int mt,
                                double kappa, double rho, double gamma, double h
{
  int i;
  double temp = sqrt(kappa - 1.0);
  double a = temp / (X - E) * (atan(temp) + atan(E * temp / (X - E))) / ((double
  double ctilde = (X - E) * (X - E) / (kappa - 1.0) + E * E; // ctilde := c/a

  // If rho<0 we use the grid for -rho.
  double absrho = ABS(rho);

  pointsx[0] = 0.0; // x_0
  pointsx[1] = x_1; // x_1 I&T said that it should be h/10.0
  i = 1;
  while (pointsx[i] < X)
    {
      pointsx[i + 1] = pointsx[i]
                       + MIN(
                         MAX(
```

```
                              a * (pointsx[i] * pointsx[i] - 2 * E * pointsx[i] + c
                              (1.1 / (1.0 - absrho / gamma * h)) * (absrho / gamma)
                            ),
                            0.9 * (h / absrho) * (pointsx[i] / gamma)
                          );
      i++;
    }
}


static void grid_generation_uniform(double *pointsz, double Z, double E, int Nb)
{
  int j;
  for (j = 0; j <= Nb; j++)
    {
      pointsz[j] = j * Z / ((double)Nb);
    }
}

void grid_generation_HF_spot(double *pointsx, double Smax, double K, int m1, dou
{
  // Tests on parameters would be useful.

  int i;
  double temp = pnl_asinh(-K / c);
  double deltaxi = (pnl_asinh((Smax - K) / c) - temp) / m1;

  // Definition of uniform grid xi.
  for (i = 0; i <= m1; i++)
    {
      pointsx[i] = temp + i *  deltaxi;
    }

  // Definition of the spot grid with the uniform grid xi.
  pointsx[0] = 0;
  for (i = 1; i <= m1; i++)
    {
      pointsx[i] = K + c * sinh(pointsx[i]);
    }
}
```

```
static int lower_index(double *grid, int size, double value)
{
  double value_nearest = ABS(grid[0] - value);
  int index_nearest = -1;
  int i;
  for (i = 0; i < size; i++)
    {
      if (ABS(grid[i] - value) <= value_nearest)
        {
          value_nearest = ABS(grid[i] - value);
          index_nearest = i;
        }
    }
  if (grid[index_nearest] > value)
    {
      return index_nearest - 1;
    }
  else
    {
      return index_nearest;
    }
}

static void reorder_unknowns_x_to_xy(PnlVect *Unknowns, int M, int N, PnlVect *S
{

  int k, l;

  /*
  * * * * * //M=4
  * * * * *
  * * * * *
  * * * * *
  //N=3

  // xy order
   0  2  5  9 13
   1  4  8 12 16
   3  7 11 15 18
   6 10 14 17 19
```

```
  // x order
  15 16 17 18 19
  10 11 12 13 14
   5  6  7  8  9
   0  1  2  3  4

  // y order
   3  7 11 15 19
   2  6 10 14 18
   1  5  9 13 17
   0  4  8 12 16
  */

  int first_index = M * N + N; // Index of the first unknown of the first diagon
  int cumul_index = 0;

  for (k = 0; k < M + N + 1; k++) // Number of diagonal lines.
    {
      for (l = 0; l < MIN(MIN(k + 1, MIN(M, N) + 1), M + N + 1 - k); l++) // Len
        {
          // If l is increasing by 1, then the index of the unknown (in the x or
          // If k is increasing by 1, then the index of the first unknown of a d
          // by M+1 (if k is lesser or equal than N) or is increasing by 1 (if k
          pnl_vect_set(Sortie, cumul_index, pnl_vect_get(Unknowns, first_index +
          cumul_index++;
        }
      if (k < N)
        {
          first_index = first_index - M - 1;
        }
      else
        {
          first_index = first_index + 1;
        }
    }
}

static void reorder_unknowns_xy_to_y(PnlVect *Unknowns, int M, int N, PnlVect *S
{

  int k, l;
```

```
/*
* * * * * //M=4
* * * * *
* * * * *
* * * * *
//N=3

// xy order
 0  2  5  9 13
 1  4  8 12 16
 3  7 11 15 18
 6 10 14 17 19

// x order
15 16 17 18 19
10 11 12 13 14
 5  6  7  8  9
 0  1  2  3  4

// y order
 3  7 11 15 19
 2  6 10 14 18
 1  5  9 13 17
 0  4  8 12 16
*/


int first_index = N; // Index (in the y order) of the first unknown of the fir
int cumul_index = 0;

for (k = 0; k < M + N + 1; k++) // Number of diagonal lines.
  {
    for (l = 0; l < MIN(MIN(k + 1, MIN(M, N) + 1), M + N + 1 - k); l++) // Len
      {
        // If l is increasing by 1, then the index of the unknown (in the y or
        // If k is increasing by 1, then the index of the first unknown of a d
        // by 1 (if k is lesser or equal than N) or is increasing by N+1 (if k
        pnl_vect_set(Sortie, first_index + l * (N + 2), pnl_vect_get(Unknowns,
        cumul_index++;
      }
```

```
      if (k < N)
        {
          first_index = first_index - 1;
        }
      else
        {
          first_index = first_index + N + 1;
        }
    }
}

static void reorder_unknowns_y_to_x(PnlVect *Unknowns, int M, int N, PnlVect *So
{

  int i, j;

  /*
  * * * * * //M=4
  * * * * *
  * * * * *
  * * * * *
  //N=3

  // xy order
   0  2  5  9 13
   1  4  8 12 16
   3  7 11 15 18
   6 10 14 17 19

  // x order
  15 16 17 18 19
  10 11 12 13 14
   5  6  7  8  9
   0  1  2  3  4

  // y order
   3  7 11 15 19
   2  6 10 14 18
   1  5  9 13 17
   0  4  8 12 16
  */
```

```
  for (i = 0; i <= M; i++) // Number of colomns.
    {
      for (j = 0; j <= N; j++) // Number of lines.
        {
          pnl_vect_set(Sortie, i + j * (M + 1), pnl_vect_get(Unknowns, i * (N +
        }
    }
}

static void construction_Ax_coefficients(
  double r, double divid,
  double alpha, double beta, double gamma, double rho,
  // index_pointx is useless here
  double X, int M, double *pointsx, int index_pointx,
  double Y, int N, double *pointsy, int index_pointy,
  double omega,
  double *lower_d, double *diagonal, double *upper_d)
{
  // The points are ordering from left to right then from bottom to top.
  double actualpointx;
  double actualpointy = pointsy[index_pointy];
  double hl, hr, kl, kr;
  double omega_local;
// double mixed_derivative;
  double convection_x, diffusion_x, order_0;
  int backward_center_forward;
  int i;

  actualpointx = 0.0;

  if (index_pointy > 0)
    {
      for (i = 1; i < M; i++)
        {
          omega_local = omega;
          if (i == 1)
            {
              omega_local = 0.0;  // Avoid non positive off-diagonal elements.
            }
          if ((i == M - 1) || (index_pointy == N))
```

```
  {
    omega_local = 1.0;
  }
// I&T said that it is for Neumann boundary conditions...
actualpointx = pointsx[i];
if (index_pointy == N)
  {
    kr = 10000000000.0; // Since omega_local = 1.0, kr will be useless
    // To be sure that the coefficient involving kr is null,
    // we set kr = 10000000000.0 (division by kr makes it little).
  }
else
  {
    kr = pointsy[index_pointy + 1] - pointsy[index_pointy];
  }
kl = pointsy[index_pointy] - pointsy[index_pointy - 1];
hr = pointsx[i + 1] - pointsx[i];
hl = pointsx[i] - pointsx[i - 1];

// Because positive elements must be avoided,
// we should choose backward, center of forward finite difference sche

// Test between diffusion and convection.
convection_x = -(r - divid) * actualpointx - omega_local * rho * gamma
               + (1.0 - omega_local) * rho * gamma * actualpointy * a

diffusion_x = -actualpointy * actualpointx * actualpointx / 2.0
              + omega_local * rho * gamma * actualpointy * actualpoint
              + (1.0 - omega_local) * rho * gamma * actualpointy * act
order_0 = r / 3.0;

backward_center_forward = 0;
// If centered scheme is chosen.
if (-actualpointy * actualpointx + omega_local * rho * gamma * actualp
  {
    backward_center_forward = 1;
  }
if (-actualpointy * actualpointx + (1.0 - omega_local)*rho * gamma * a
  {
    backward_center_forward = -1;
  }
```

```
      if (backward_center_forward == -1)
        {
          // Backward scheme.
          // Lower diagonal (left point).
          lower_d[i - 1] = -convection_x / hl + 2.0 / (hl + hr) * diffusion_
          // Diagonal (centered point).
          diagonal[i] = convection_x / hl - 2.0 / (hl * hr) * diffusion_x +
          // Upper diagonal (right point).
          upper_d[i] = 0.0 + 2.0 / (hl + hr) * diffusion_x / hr;
        }
      if (backward_center_forward == 0)
        {
          // Centered scheme.
          // Lower diagonal (left point).
          lower_d[i - 1] = -convection_x / (hl + hr) * hr / hl + 2.0 / (hl +
          // Diagonal (centered point).
          diagonal[i] = -convection_x / (hl + hr) * (hl / hr - hr / hl) - 2.
          // Upper diagonal (right point).
          upper_d[i] = convection_x / (hl + hr) * hl / hr + 2.0 / (hl + hr)
        }
      if (backward_center_forward == 1)
        {
          // Forward scheme.
          // Lower diagonal (left point).
          lower_d[i - 1] = 0.0 + 2.0 / (hl + hr) * diffusion_x / hl;
          // Diagonal (centered point).
          diagonal[i] = -convection_x / hr - 2.0 / (hl * hr) * diffusion_x +
          // Upper diagonal (right point).
          upper_d[i] = convection_x / hr + 2.0 / (hl + hr) * diffusion_x / h
        }
    }

  // Boundary conditions are treated after.
  // But for minimal spot, this is always Dirichlet.
  diagonal[0] = 0.0;
  upper_d[0] = 0.0;
  // For maximal spot, since it could change we set 0.
  diagonal[M] = 0.0;
  lower_d[M - 1] = 0.0;
}
```

```
else // index_pointy==0 // All this part can be simplified, since (if rho>=0)
  {
    for (i = 1; i < M; i++)
      {
        omega_local = omega;
        if (i == 1)
          {
            omega_local = 0.0;  // Avoid non positive off-diagonal elements.
          }
        if (i == M - 1)
          {
            omega_local = 1.0;
          }
        // I&T said that it is for Neumann boundary conditions...
        actualpointx = pointsx[i];
        kr = pointsy[index_pointy + 1] - pointsy[index_pointy];
        kl = 10000000000.0;
        hr = pointsx[i + 1] - pointsx[i];
        hl = pointsx[i] - pointsx[i - 1];

        // Because positive elements must be avoided,
        // we should choose backward, center of forward finite difference sche

        // Test between diffusion and convection.
        convection_x = -(r - divid) * actualpointx - omega_local * rho * gamma
                       + (1.0 - omega_local) * rho * gamma * actualpointy * a
        diffusion_x = -actualpointy * actualpointx * actualpointx / 2.0
                      + omega_local * rho * gamma * actualpointy * actualpoint
                      + (1.0 - omega_local) * rho * gamma * actualpointy * act
        order_0 = r / 3.0;
        backward_center_forward = 0;
        // If centered scheme is chosen.
        if (-actualpointy * actualpointx + omega_local * rho * gamma * actualp
          {
            backward_center_forward = 1;
          }
        if (-actualpointy * actualpointx + (1.0 - omega_local)*rho * gamma * a
          {
            backward_center_forward = -1;
          }
        if (backward_center_forward == -1)
```

```
            {
              // Backward scheme.
              // Lower diagonal (left point).
              lower_d[i - 1] = -convection_x / hl + 2.0 / (hl + hr) * diffusion_
              // Diagonal (centered point).
              diagonal[i] = convection_x / hl - 2.0 / (hl * hr) * diffusion_x +
              // Upper diagonal (right point).
              upper_d[i] = 0.0 + 2.0 / (hl + hr) * diffusion_x / hr;
            }
          if (backward_center_forward == 0)
            {
              // Centered scheme.
              // Lower diagonal (left point).
              lower_d[i - 1] = -convection_x / (hl + hr) * hr / hl + 2.0 / (hl +
              // Diagonal (centered point).
              diagonal[i] = -convection_x / (hl + hr) * (hl / hr - hr / hl) - 2.
              // Upper diagonal (right point).
              upper_d[i] = convection_x / (hl + hr) * hl / hr + 2.0 / (hl + hr)
            }
          if (backward_center_forward == 1)
            {
              // Forward scheme.
              // Lower diagonal (left point).
              lower_d[i - 1] = 0.0 + 2.0 / (hl + hr) * diffusion_x / hl;
              // Diagonal (centered point).
              diagonal[i] = -convection_x / hr - 2.0 / (hl * hr) * diffusion_x +
              // Upper diagonal (right point).
              upper_d[i] = convection_x / hr + 2.0 / (hl + hr) * diffusion_x / h
            }
        }
      // Boundary conditions are treated after.
      // But for minimal spot, this is always Dirichlet.
      diagonal[0] = 0.0;
      upper_d[0] = 0.0;
      // For maximal spot, since it could change we set 0.
      diagonal[M] = 0.0;
      lower_d[M - 1] = 0.0;
    }
}

static void construction_Ay_coefficients(
```

```
  double r, double divid,
  double alpha, double beta, double gamma, double rho,
  double X, int M, double *pointsx, int index_pointx,
  // pointy is useless here
  double Y, int N, double *pointsy, int index_pointy,
  double omega,
  double *lower_d, double *diagonal, double *upper_d)
{
  // The points are ordering from bottom to top then left to right
  double actualpointx = pointsx[index_pointx];
  double actualpointy;
  double hl, hr, kl, kr;
  double omega_local;
  double convection_y, diffusion_y, order_0;
  int backward_center_forward;
  int j;

  actualpointy = 0.0;

  for (j = 1; j < N; j++)
    {
      omega_local = omega;
      if (j == 1)
        {
          omega_local = 0.0;  // Avoid non positive off-diagonal elements.
        }
      if ((j == N - 1) || (index_pointx == M))
        {
          omega_local = 1.0;  // I&T said that it is for Neumann boundary condit
        }
      //actualpointx = pointsx[index_pointx];
      actualpointy = pointsy[j];
      if (index_pointx == M)
        {
          hr = 10000000000.0; // Since omega_local = 1.0, hr will be useless her
          // To be sure that the coefficient involving hr is null,
          // we set hr = 10000000000.0 (division by hr makes it little).
        }
      else
        {
          hr = pointsx[index_pointx + 1] - pointsx[index_pointx];
```

```
  }
hl = pointsx[index_pointx] - pointsx[index_pointx - 1];
kr = pointsy[j + 1] - pointsy[j];
kl = pointsy[j] - pointsy[j - 1];
// Because positive elements must be avoided,
// we should choose backward, center of forward finite difference schemes.

// Test between diffusion and convection.
convection_y = -alpha * (beta - actualpointy) - omega_local * rho * gamma
                + (1.0 - omega_local) * rho * gamma * actualpointy * actua
diffusion_y = -actualpointy * gamma * gamma / 2.0
              + omega_local * rho * gamma * actualpointy * actualpointx *
              + (1.0 - omega_local) * rho * gamma * actualpointy * actualp
order_0 = r / 3.0;

backward_center_forward = 0;
if (-gamma * gamma * actualpointy + 2.0 * rho * gamma * actualpointy * act
  {
    backward_center_forward = 1;
  }
if (-gamma * gamma * actualpointy + 2.0 * rho * gamma * actualpointy * act
  {
    backward_center_forward = -1;
  }


if (backward_center_forward == -1)
  {
    // Backward scheme.
    // Lower diagonal (left point).
    lower_d[j - 1] = -convection_y / kl + 2.0 / (kl + kr) * diffusion_y /
    // Diagonal (centered point).
    diagonal[j] = convection_y / kl - 2.0 * diffusion_y / (kl * kr) + orde
    // Upper diagonal (right point).
    upper_d[j] = 0.0 + 2.0 / (kl + kr) * diffusion_y / kr;
  }
if (backward_center_forward == 0)
  {
    // Centered scheme.
    // Lower diagonal (left point).
    lower_d[j - 1] = -convection_y / (kl + kr) * kr / kl + 2.0 / (kl + kr)
```

```
        // Diagonal (centered point).
        diagonal[j] = -convection_y / (kl + kr) * (kl / kr - kr / kl) - 2.0 /
        // Upper diagonal (right point).
        upper_d[j] = convection_y / (kl + kr) * kl / kr + 2.0 / (kl + kr) * di
      }
    if (backward_center_forward == 1)
      {
        // Forward scheme.
        // Lower diagonal (left point).
        lower_d[j - 1] = 0.0 + 2.0 / (kl + kr) * diffusion_y / kl;
        // Diagonal (centered point).
        diagonal[j] = -convection_y / kr  - 2.0 / (kl * kr) * diffusion_y + or
        // Upper diagonal (right point).
        upper_d[j] = convection_y / kr + 2.0 / (kl + kr) * diffusion_y / kr;
      }
  }

// Boundary conditions are treated after.
// But for minimal var/vol, this is often outflow.
{
  j = 0;
  actualpointy = pointsy[j]; // Actually it is 0.0
  kr = pointsy[j + 1] - pointsy[j];
  //kl = pointsy[j]-pointsy[j-1]; // Undefined

  convection_y = -alpha * (beta - actualpointy);
  // - omega_local*rho*gamma*actualpointy*actualpointx/hl
  // +(1.0 - omega_local)*rho*gamma*actualpointy*actualpointx/hr;
  diffusion_y = 0.0;
  // -actualpointy*gamma*gamma/2.0
  // +omega_local*rho*gamma*actualpointy*actualpointx*kl/(2.0*hl)
  // +(1.0 - omega_local)*rho*gamma*actualpointy*actualpointx*kr/(2.0*hr);
  order_0 = (r / 3.0);

  backward_center_forward = 1; // Always forward since var/vol=0.0;
  if (backward_center_forward == 1)
    {
      // Forward scheme.
      // Lower diagonal (left point).
      //lower_d[j-1] = 0.0 + 2.0/(kl+kr) * diffusion_y/kl; // Undefined
      // Diagonal (centered point).
```

```
        diagonal[j] = -convection_y / kr + order_0;
        //- 2.0/(kl*kr) * diffusion_y // Undefined or null
        // Upper diagonal (right point).
        upper_d[j] = convection_y / kr;
        //+ 2.0/(kl+kr) * diffusion_y/kr // Undefined or null
      }
  }

  //diagonal[0]=0.0;
  //upper_d[0]=0.0;
  // For maximal var/vol, since it could change we set 0.
  diagonal[N] = 0.0;
  lower_d[N - 1] = 0.0;
}

static void construction_Axy_coefficients(double r, double divid,
    double alpha, double beta, double gamma, double rho,
    // index of the first point in the diagonal
    double X, int M, double *pointsx, int index_pointx,
    // first y point in the diagonal
    double Y, int N, double *pointsy, int index_pointy,
    int number_coeff_on_diagonal,
    double omega,
    double *lower_d, double *diagonal, double *upper_d)
{
  // The points are ordering from left bottom to right top among diagonals from
  double actualpointx, actualpointy;
  double hl, hr, kl, kr;
  double omega_local;
  double order_0;
  //double order_0_offdiag;
  int l;

  actualpointx = 0.0;
  actualpointy = 0.0;

  if (number_coeff_on_diagonal > 2) // Three or more than three points.
    {

      for (l = 1; l < number_coeff_on_diagonal - 1; l++)
        {
```

```
    omega_local = omega;
    if (index_pointx + l == 1)
      {
        omega_local = 0.0;  // Avoid non positive off-diagonal elements.
      }
    if (l == number_coeff_on_diagonal - 2)
      {
        // if l==number_coeff_on_diagonal-2 (i.e. index_pointx+l==M-1)
        omega_local = 1.0;
      } // I&T said that it is for Neumann boundary conditions...

    actualpointx = pointsx[index_pointx + l];
    actualpointy = pointsy[index_pointy + l];

    hr = pointsx[index_pointx + l + 1] - pointsx[index_pointx + l];
    hl = pointsx[index_pointx + l] - pointsx[index_pointx + l - 1];
    kr = pointsy[index_pointy + l + 1] - pointsy[index_pointy + l];
    kl = pointsy[index_pointy + l] - pointsy[index_pointy + l - 1];

    order_0 = (r / 3.0 + omega_local * rho * gamma * actualpointy * actual
               + (1.0 - omega_local) * rho * gamma * actualpointy * actual
    // Lower diagonal (left point).
    lower_d[l - 1] = -rho * gamma * actualpointy * actualpointx / kl * ome
    // Diagonal (centered point).
    diagonal[l] = order_0;
    // Upper diagonal (right point).
    upper_d[l] = -rho * gamma * actualpointy * actualpointx / kr * (1.0 -
  }
// Boundary conditions are treated after.
// But for minimal var/vol, this is often outflow.
// Actually, at the first point of a diagonal, we have always actualpointx
// So there is no lower coefficient (we are at the first point)
// neither upper coefficient (it is null). Only the diagonal coefficient e
// Lower diagonal (left point) does not exist.
// Diagonal (centered point).
actualpointx = pointsx[index_pointx];
if (actualpointx > 0) // We are on the boundary with minimal var/vol = 0.
  {
    diagonal[0] = r / 3.0; // order_0 coefficient.
  }
else // We are on the boundary Spot=0. Dirichlet condition so coefficient
```

```
        {
          diagonal[0] = 0.0; // order_0 coefficient.
        }
      // Upper diagonal (upper-right point).
      upper_d[0] = 0.0;
      // For maximal var/vol/spot, since it could change we set 0.
      diagonal[number_coeff_on_diagonal - 1] = 0.0;
      lower_d[number_coeff_on_diagonal - 2] = 0.0;
    }
  else // number_coeff_on_diagonal==2 or 1.
    {
      if (number_coeff_on_diagonal == 2)
        {
          // Coefficients which correspond to the computation of minimal var/vol
          // As said previously, we have always actualpointx*actualpointy = 0 !
          // So the upper coefficient is null and diagonal coefficient is only o
          actualpointx = pointsx[index_pointx];
          if (actualpointx > 0) // We are on the boundary with minimal var/vol.
            {
              diagonal[0] = r / 3.0; // order_0 coefficient.
            }
          else // We are on the boundary Spot=0. Dirichlet condition so coeffici
            {
              diagonal[0] = 0.0; // order_0 coefficient.
            }
          upper_d[0] = 0.0;
          // Other coefficients correspond to the computation of maximal var/vol
          // Since it could change we set 0.
          lower_d[0] = 0.0;
          diagonal[1] = 0.0;
        }
      else // number_coeff_on_diagonal==1. Since it could change we set 0.
        {
          diagonal[0] = 0.0;
        }
    }
}

static void modify_solution_with_neumann_boundary_condition(
  double r, double divid, double time, int call_or_put,
  int M, double *pointsx, int N, double *pointsy, double coeff,
```

```
  PnlVect *VectOne, PnlVect *Sortie)
{
  int i, j;
  double boundary_condition;

  for (j = 2; j <= N - 2; j++)
    {
      // i = M (i.e. Spot max <=> Neumann condition)
      i = M;
      boundary_condition = (pointsx[i] * pointsx[i] * pointsy[j] / (2.0 * (point
                            + (r - divid) * pointsx[i]) * ((1 + call_or_put) / 2
      pnl_vect_set(
        Sortie, j * (M + 1) + i,
        pnl_vect_get(VectOne, j * (M + 1) + i)
        - coeff * (
          // Neumann boundary condition
          boundary_condition
        )
      );
    }
  // j=0
  {
    j = 0;
    // i = M
    i = M;
    boundary_condition = (pointsx[i] * pointsx[i] * pointsy[j] / (2.0 * (pointsx
                          + (r - divid) * pointsx[i]) * ((1 + call_or_put) / 2)
    pnl_vect_set(
      Sortie, j * (M + 1) + i,
      pnl_vect_get(VectOne, j * (M + 1) + i)
      - coeff * (
        // Neumann boundary condition
        + boundary_condition
      )
    );
  }
  // j=1
  {
    j = 1;
    // i = M
    i = M;
```

```
  boundary_condition = (pointsx[i] * pointsx[i] * pointsy[j] / (2.0 * (pointsx
                       + (r - divid) * pointsx[i]) * ((1 + call_or_put) / 2)
  pnl_vect_set(
    Sortie, j * (M + 1) + i,
    pnl_vect_get(VectOne, j * (M + 1) + i)
    - coeff * (
      // Neumann boundary condition
      + boundary_condition
    )
  );
}
// j=N-1
{
  j = N - 1;
  // i = M
  i = M;
  boundary_condition = (pointsx[i] * pointsx[i] * pointsy[j] / (2.0 * (pointsx
                       + (r - divid) * pointsx[i]) * ((1 + call_or_put) / 2)
  pnl_vect_set(
    Sortie, j * (M + 1) + i,
    pnl_vect_get(VectOne, j * (M + 1) + i)
    - coeff * (
      // Neumann boundary condition
      + boundary_condition
    )
  );
}
// j=N
{
  j = N;
  // i = M
  i = M;
  boundary_condition = (pointsx[i] * pointsx[i] * pointsy[j] / (2.0 * (pointsx
                       + (r - divid) * pointsx[i]) * ((1 + call_or_put) / 2)
  pnl_vect_set(
    Sortie, j * (M + 1) + i,
    pnl_vect_get(VectOne, j * (M + 1) + i)
    - coeff * (
      // Dirichlet condition wins versus Neumann boundary condition.
      0.0
    )
```

```
    );
  }
}

void do_rannacher_iteration_new(double r, double divid, double deltat, int call_
                                double alpha, double beta, double gamma, double
                                int M, double *pointsx, int N, double *pointsy,
                                double coeff, PnlVect *VectTwo, PnlVect *Sortie)
{

  // First, we build the full matrix of the pde.
  // In "construction_matrix_A_and_A0", the matrix is of size (M+1)*(N+1) times
  // minimize the space in memory. And the explicit computations are developped
  // "computation_explicit_syslin_total_matrix"
  // But here, we need to "inverse" the full matrix, so we must build it in its

  // This procedure computes the solution "Sortie" of
  // Sortie = VectTwo - coeff * (MatrixTotal * Sortie + Boundary conditions)
  // which is also the solution of
  // (Id + coeff * MatrixTotal) Sortie = VectTwo - coeff * Boundary conditions

  // Be careful, it is akward but there is a minus (before coeff in the second m
  // in order to be consistent with the other procedure which use a negative "co

  // Build lines of constant volatility and increasing spot.
  //    *   6
  //  * * * 9 5 10
  //  * * *     <=>1 0 2
  //  * * * 7 4 8
  //    *   3
  // MatrixTotal(0, . ) <= (i,j)
  // MatrixTotal(1, . ) <= (i-1,j)
  // MatrixTotal(2, . ) <= (i+1,j)
  // MatrixTotal(3, . ) <= (i,j-2)
  // MatrixTotal(4, . ) <= (i,j-1)
  // MatrixTotal(5, . ) <= (i,j+1)
  // MatrixTotal(6, . ) <= (i,j+2)
  // MatrixTotal(7, . ) <= (i-1,j-1)
  // MatrixTotal(8, . ) <= (i+1,j-1)
  // MatrixTotal(9, . ) <= (i-1,j+1)
  // MatrixTotal(10, . ) <= (i+1,j+1)
```

```
double actualpointx;
double actualpointy;
double Dxi, Dxip1;
double Dyjm1, Dyj, Dyjp1, Dyjp2;
double convection_x, diffusion_x, convection_y, diffusion_y, mixted, order_0;
// Coefficients
double xbetaim1, xbetai0, xbetaip1;
double xdeltaim1, xdeltai0, xdeltaip1;
double yalphajm2, yalphajm1, yalphaj0;
double ybetajm1, ybetaj0, ybetajp1;
double ygammaj0, ygammajp1, ygammajp2;
double ydeltajm1, ydeltaj0, ydeltajp1;

int i, j;

PnlVect *Ones;
PnlVect *SecondMember;
PnlMat *Working_Matrix;
PnlMat *MatrixTotal;
PnlVectInt *permutation;

MatrixTotal = pnl_mat_create((M + 1) * (N + 1), (M + 1) * (N + 1));

// Interior domain : Since scheme is centered or backward in vol : 2 <= j <= N
for (j = 2; j <= N - 1; j++)
  {
    // Interior domain : Since scheme is centered in spot : 1 <= i <= M-1

    for (i = 1; i <= M - 1; i++)
      {
        //Dxim1 = pointsx[i-1]-pointsx[i-2]; // Not used
        Dxi = pointsx[i] - pointsx[i - 1]; // Used
        Dxip1 = pointsx[i + 1] - pointsx[i]; // Used
        //Dxip2 = pointsx[i+2]-pointsx[i+1]; // Not used
        Dyjm1 = pointsy[j - 1] - pointsy[j - 2]; // Used
        Dyj = pointsy[j] - pointsy[j - 1]; // Used
        Dyjp1 = pointsy[j + 1] - pointsy[j]; // Used
        //Dyjp2 = pointsy[j+2]-pointsy[j+1]; // Not used
```

```
            //xalphaim2 = Dxi/(Dxim1*(Dxim1+Dxi)); // Not used
            //xalphaim1 = -(Dxim1+Dxi)/(Dxim1*Dxi); // Not used
            //xalphai0 = (Dxim1+2.0*Dxi)/(Dxi*(Dxim1+Dxi)); // Not used
            xbetaim1 = -Dxip1 / (Dxi * (Dxi + Dxip1)); // Used in Mixted
            xbetai0 = (Dxip1 - Dxi) / (Dxi * Dxip1); // Used in Mixted
            xbetaip1 = Dxi / (Dxip1 * (Dxi + Dxip1)); // Used in Mixted
            //xgammai0 = -(2.0*Dxip1+Dxip2)/(Dxip1*(Dxip1+Dxip2));
// Not used
            //xgammaip1 = (Dxip1+Dxip2)/(Dxip1*Dxip2); // Not used
            //xgammaip2 = -Dxip1/(Dxip2*(Dxip1+Dxip2)); // Not used
            xdeltaim1 = 2.0 / (Dxi * (Dxi + Dxip1)); // Used in Total : centered s
            xdeltai0 = -2.0 / (Dxi * Dxip1); // Used in Total : centered scheme
            xdeltaip1 = 2.0 / (Dxip1 * (Dxi + Dxip1)); // Used in Total : centered

            yalphajm2 = Dyj / (Dyjm1 * (Dyjm1 + Dyj)); // Used in Total : upwind s
            yalphajm1 = -(Dyjm1 + Dyj) / (Dyjm1 * Dyj); // Used in Total : upwind
            yalphaj0 = (Dyjm1 + 2.0 * Dyj) / (Dyj * (Dyjm1 + Dyj));
// Used in Total : upwind scheme
            ybetajm1 = -Dyjp1 / (Dyj * (Dyj + Dyjp1)); // Used in Mixted
            ybetaj0 = (Dyjp1 - Dyj) / (Dyj * Dyjp1); // Used in Mixted
            ybetajp1 = Dyj / (Dyjp1 * (Dyj + Dyjp1)); // Used in Mixted
            //ygammaj0 = -(2.0*Dyjp1+Dyjp2)/(Dyjp1*(Dyjp1+Dyjp2));
// Not used
            //ygammajp1 = (Dyjp1+Dyjp2)/(Dyjp1*Dyjp2); // Not used
            //ygammajp2 = -Dyjp1/(Dyjp2*(Dyjp1+Dyjp2)); // Not used
            ydeltajm1 = 2.0 / (Dyj * (Dyj + Dyjp1)); // Used in Total : centered s
            ydeltaj0 = -2.0 / (Dyj * Dyjp1); // Used in Total : centered scheme
            ydeltajp1 = 2.0 / (Dyjp1 * (Dyj + Dyjp1)); // Used in Total : centered

            actualpointy = pointsy[j];
            actualpointx = pointsx[i];

            order_0 = -r;
            convection_x = (r - divid) * actualpointx;
            convection_y = alpha * (beta - actualpointy);
            diffusion_x = actualpointx * actualpointx * actualpointy / 2.0;
            diffusion_y = gamma * gamma * actualpointy / 2.0;
            mixted = rho * gamma * actualpointx * actualpointy;
```

```
if ((actualpointy > 1) && (convection_y < 0))
  // In this case, we use the upwind scheme whenever the flow
  // in the v-direction is towards v=V. (i.e. alpha(beta - v) < 0).
  {

    // MatrixTotal
    // x y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i,
                diffusion_x * xdeltai0
                + mixted * xbetai0 * ybetaj0
                + diffusion_y * ydeltaj0
                + convection_x * xbetai0
                + convection_y * yalphaj0
                + order_0);
    // x-1 y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i - 1,
                diffusion_x * xdeltaim1
                + mixted * xbetaim1 * ybetaj0
                + convection_x * xbetaim1);
    // x+1 y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i + 1,
                diffusion_x * xdeltaip1
                + mixted * xbetaip1 * ybetaj0
                + convection_x * xbetaip1);
    // x y-2 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 2) * (M + 1) + i,
                convection_y * yalphajm2);
    // x y-1 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i,
                mixted * xbetai0 * ybetajm1
                + diffusion_y * ydeltajm1
                + convection_y * yalphajm1);
    // x y+1 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i,
                mixted * xbetai0 * ybetajp1
                + diffusion_y * ydeltajp1);

    // x y+2 coefficient
    if (j < N - 1) // Coefficient does not exist if j=N-1.
      {
        pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 2) * (M + 1) +
```

```
      }
    //####
    //####
    //####
    //####
    //####
    // x-1 y-1 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i -
                mixted * xbetaim1 * ybetajm1);
    // x+1 y-1 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i +
                mixted * xbetaip1 * ybetajm1);
    // x-1 y+1 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i -
                mixted * xbetaim1 * ybetajp1);
    // x+1 y+1 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i +
                mixted * xbetaip1 * ybetajp1);
    //####
    //####
    //####
    //####
  }
else // centered scheme in volatility
  {
    // MatrixTotal
    // x y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i, 0.0);
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i,
                diffusion_x * xdeltai0
                + mixted * xbetai0 * ybetaj0
                + diffusion_y * ydeltaj0
                + convection_x * xbetai0
                + convection_y * ybetaj0
                + order_0);
    // x-1 y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i - 1,
                diffusion_x * xdeltaim1
                + mixted * xbetaim1 * ybetaj0
                + convection_x * xbetaim1);
    // x+1 y coefficient
```

```
pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i + 1,
            diffusion_x * xdeltaip1
            + mixted * xbetaip1 * ybetaj0
            + convection_x * xbetaip1);
// x y-2 coefficient
pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 2) * (M + 1) + i,
            0.0);
// x y-1 coefficient
pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i,
            mixted * xbetai0 * ybetajm1
            + diffusion_y * ydeltajm1
            + convection_y * ybetajm1);
// x y+1 coefficient
pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i,
            mixted * xbetai0 * ybetajp1
            + diffusion_y * ydeltajp1
            + convection_y * ybetajp1);

// x y+2 coefficient
if (j < N - 1) // Coefficient does not exist if j=N-1.
  {
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 2) * (M + 1) +
  }
//####
//####
//####
//####
//####
// x-1 y-1 coefficient
pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i -
            mixted * xbetaim1 * ybetajm1);
// x+1 y-1 coefficient
pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i +
            mixted * xbetaip1 * ybetajm1);
// x-1 y+1 coefficient
pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i -
            mixted * xbetaim1 * ybetajp1);
// x+1 y+1 coefficient
pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i +
            mixted * xbetaip1 * ybetajp1);
//####
```

```
        //####
        //####
        //####

      }
  }


// For the minimal spot (i==0), there is a Dirichlet condition.
// So the matrix is null on this area. The condition is in the second memb
i = 0;
pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i, 0.0);
// Other coefficient are null by default.


// For the maximal spot (i==M), there is a Neumann condition modifying the
i = M;
Dxi = pointsx[i] - pointsx[i - 1]; // Only coefficient used in x-direction
Dyjm1 = pointsy[j - 1] - pointsy[j - 2]; // Used
Dyj = pointsy[j] - pointsy[j - 1]; // Used
Dyjp1 = pointsy[j + 1] - pointsy[j]; // Used
//Dyjp2 = pointsy[j+2]-pointsy[j+1]; // Not used

// In the x-direction, all coefficient have been simplified.
yalphajm2 = Dyj / (Dyjm1 * (Dyjm1 + Dyj)); // Used in Total : upwind schem
yalphajm1 = -(Dyjm1 + Dyj) / (Dyjm1 * Dyj); // Used in Total : upwind sche
yalphaj0 = (Dyjm1 + 2.0 * Dyj) / (Dyj * (Dyjm1 + Dyj)); // Used in Total :
ybetajm1 = -Dyjp1 / (Dyj * (Dyj + Dyjp1)); // Used in Mixted
ybetaj0 = (Dyjp1 - Dyj) / (Dyj * Dyjp1); // Used in Mixted
ybetajp1 = Dyj / (Dyjp1 * (Dyj + Dyjp1)); // Used in Mixted
//ygammaj0 = -(2.0*Dyjp1+Dyjp2)/(Dyjp1*(Dyjp1+Dyjp2)); // Not used
//ygammajp1 = (Dyjp1+Dyjp2)/(Dyjp1*Dyjp2); // Not used
//ygammajp2 = -Dyjp1/(Dyjp2*(Dyjp1+Dyjp2)); // Not used
ydeltajm1 = 2.0 / (Dyj * (Dyj + Dyjp1)); // Used in Total : centered schem
ydeltaj0 = -2.0 / (Dyj * Dyjp1); // Used in Total : centered scheme
ydeltajp1 = 2.0 / (Dyjp1 * (Dyj + Dyjp1)); // Used in Total : centered sch

actualpointy = pointsy[j];
actualpointx = pointsx[i];

order_0 = -r;
```

```
convection_x = (r - divid) * actualpointx; // It will be in the coefficien
convection_y = alpha * (beta - actualpointy);
diffusion_x = actualpointx * actualpointx * actualpointy / 2.0;
diffusion_y = gamma * gamma * actualpointy / 2.0;
mixted = rho * gamma * actualpointx * actualpointy;

//std::cout << "i = " << i << " j = " << j << std::endl;

if ((actualpointy > 1) && (convection_y < 0))
  // In this case, we use the upwind scheme whenever the flow
  // in the v-direction is towards v=V. (i.e. alpha(beta - v) < 0).
  {
    // MatrixTotal
    // x y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i,
                //diffusion_x * xdeltai0 // Diffusion term is modified :
                diffusion_x * (- 1.0 / (Dxi * Dxi))
                //+ mixted * xbetai0 * ybetaj0 // No mixed term
                + diffusion_y * ydeltaj0
                //+ convection_x * xbetai0 // No convection_x term. It is
                + convection_y * yalphaj0
                + order_0);
    // x-1 y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i - 1,
                //diffusion_x * xdeltaim1 // Diffusion term is modified :
                diffusion_x * (1.0 / (Dxi * Dxi)));
    //+ mixted * xbetaim1 * ybetaj0 // No mixed term
    //+ convection_x * xbetaim1); // No convection_x term. It is in the co
    // x+1 y coefficient
    // pnl_mat_set(MatrixTotal, j*(M+1)+i, 2, 0.0); // Does not exist !!!
    //diffusion_x * xdeltaip1 // Diffusion term is modified. This term bec
    //+ mixted * xbetaip1 * ybetaj0 // No mixed term
    //convection_x * xbetaip1);// No convection_x term. It is in the coeff
    // x y-2 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 2) * (M + 1) + i,
                convection_y * yalphajm2);
    // x y-1 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i,
                //mixted * xbetai0 * ybetajm1 // No mixed term
                diffusion_y * ydeltajm1
                + convection_y * yalphajm1);
```

```
    // x y+1 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i,
                //mixted * xbetai0 * ybetajp1 // No mixed term
                diffusion_y * ydeltajp1);
    // x y+2 coefficient
    if (j < N - 1) // Coefficient does not exist if j=N-1.
      {
        pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 2) * (M + 1) + i, 0
      }
    //####
    //####
    //####
    //####
    // No corner coefficients since there is no mixed term.
    //####
    //####
    //####
    //####
  }
else // centered scheme in volatility
  {
    // MatrixTotal
    // x y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i,
                //diffusion_x * xdeltai0 // Diffusion term is modified :
                diffusion_x * (- 1.0 / (Dxi * Dxi))
                //+ mixted * xbetai0 * ybetaj0 // No mixed term
                + diffusion_y * ydeltaj0
                //+ convection_x * xbetai0 // No convection_x term. It is
                + convection_y * ybetaj0
                + order_0);
    // x-1 y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i - 1,
                //diffusion_x * xdeltaim1 // Diffusion term is modified :
                diffusion_x * (1.0 / (Dxi * Dxi)));
    //+ mixted * xbetaim1 * ybetaj0 // No mixed term
    //+ convection_x * xbetaim1);// No convection_x term. It is in the coe
    // x+1 y coefficient
    // pnl_mat_set(MatrixTotal, j*(M+1)+i, j*(M+1)+i+1, 0.0); // Does not
    //diffusion_x * xdeltaip1 // Diffusion term is modified. This term bec
    //+ mixted * xbetaip1 * ybetaj0 // No mixed term
```

```
        //convection_x * xbetaip1); // No convection_x term. It is in the coef
        // x y-2 coefficient
        pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 2) * (M + 1) + i,
                    0.0);
        // x y-1 coefficient
        pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i,
                    //mixted * xbetai0 * ybetajm1 // No mixed term
                    diffusion_y * ydeltajm1
                    + convection_y * ybetajm1);
        // x y+1 coefficient
        pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i,
                    //mixted * xbetai0 * ybetajp1 // No mixed term
                    diffusion_y * ydeltajp1
                    + convection_y * ybetajp1);
        // x y+2 coefficient
        if (j < N - 1) // Coefficient does not exist if j=N-1.
          {
            pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 2) * (M + 1) + i, 0
          }
        //####
        //####
        //####
        //####
        // No corner coefficients since there is no mixed term.
        //####
        //####
        //####
        //####
      }
  }


// Second line of volatility (i.e. j=1) : Scheme is necessarily centered in vo
{
  j = 1;
  for (i = 1; i <= M - 1; i++)
    {
      //Dxim1 = pointsx[i-1]-pointsx[i-2]; // Not used
      Dxi = pointsx[i] - pointsx[i - 1]; // Used
      Dxip1 = pointsx[i + 1] - pointsx[i]; // Used
      //Dxip2 = pointsx[i+2]-pointsx[i+1]; // Not used
```

```
//Dyjm1 = pointsy[j-1]-pointsy[j-2]; // Not used
Dyj = pointsy[j] - pointsy[j - 1]; // Used
Dyjp1 = pointsy[j + 1] - pointsy[j]; // Used
//Dyjp2 = pointsy[j+2]-pointsy[j+1]; // Not used

//xalphaim2 = Dxi/(Dxim1*(Dxim1+Dxi)); // Not used
//xalphaim1 = -(Dxim1+Dxi)/(Dxim1*Dxi); // Not used
//xalphai0 = (Dxim1+2.0*Dxi)/(Dxi*(Dxim1+Dxi)); // Not used
xbetaim1 = -Dxip1 / (Dxi * (Dxi + Dxip1)); // Used in Mixted
xbetai0 = (Dxip1 - Dxi) / (Dxi * Dxip1); // Used in Mixted
xbetaip1 = Dxi / (Dxip1 * (Dxi + Dxip1)); // Used in Mixted
//xgammai0 = -(2.0*Dxip1+Dxip2)/(Dxip1*(Dxip1+Dxip2)); // Not used
//xgammaip1 = (Dxip1+Dxip2)/(Dxip1*Dxip2); // Not used
//xgammaip2 = -Dxip1/(Dxip2*(Dxip1+Dxip2)); // Not used
xdeltaim1 = 2.0 / (Dxi * (Dxi + Dxip1)); // Used in Total : centered sch
xdeltai0 = -2.0 / (Dxi * Dxip1); // Used in Total : centered scheme
xdeltaip1 = 2.0 / (Dxip1 * (Dxi + Dxip1)); // Used in Total : centered s

//yalphajm2 = Dyj/(Dyjm1*(Dyjm1+Dyj)); // Not used
//yalphajm1 = -(Dyjm1+Dyj)/(Dyjm1*Dyj); // Not used
//yalphaj0 = (Dyjm1+2.0*Dyj)/(Dyj*(Dyjm1+Dyj)); // Not used
ybetajm1 = -Dyjp1 / (Dyj * (Dyj + Dyjp1)); // Used in Mixted
ybetaj0 = (Dyjp1 - Dyj) / (Dyj * Dyjp1); // Used in Mixted
ybetajp1 = Dyj / (Dyjp1 * (Dyj + Dyjp1)); // Used in Mixted
//ygammaj0 = -(2.0*Dyjp1+Dyjp2)/(Dyjp1*(Dyjp1+Dyjp2)); // Not used
//ygammajp1 = (Dyjp1+Dyjp2)/(Dyjp1*Dyjp2); // Not used
//ygammajp2 = -Dyjp1/(Dyjp2*(Dyjp1+Dyjp2)); // Not used
ydeltajm1 = 2.0 / (Dyj * (Dyj + Dyjp1)); // Used in Total : centered sch
ydeltaj0 = -2.0 / (Dyj * Dyjp1); // Used in Total : centered scheme
ydeltajp1 = 2.0 / (Dyjp1 * (Dyj + Dyjp1)); // Used in Total : centered s

actualpointy = pointsy[j];
actualpointx = pointsx[i];

order_0 = -r;
convection_x = (r - divid) * actualpointx;
convection_y = alpha * (beta - actualpointy);
diffusion_x = actualpointx * actualpointx * actualpointy / 2.0;
diffusion_y = gamma * gamma * actualpointy / 2.0;
mixted = rho * gamma * actualpointx * actualpointy;
```

```
//std::cout << "i = " << i << " j = " << j << std::endl;

// MatrixTotal
{
  // x y coefficient
  pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i,
              diffusion_x * xdeltai0
              + mixted * xbetai0 * ybetaj0
              + diffusion_y * ydeltaj0
              + convection_x * xbetai0
              + convection_y * ybetaj0
              + order_0);
  // x-1 y coefficient
  pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i - 1,
              diffusion_x * xdeltaim1
              + mixted * xbetaim1 * ybetaj0
              + convection_x * xbetaim1);
  // x+1 y coefficient
  pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i + 1,
              diffusion_x * xdeltaip1
              + mixted * xbetaip1 * ybetaj0
              + convection_x * xbetaip1);
  // x y-2 coefficient
  //pnl_mat_set(MatrixTotal, j*(M+1)+i, (j-2)*(M+1)+i, 0.0); // Does not
  // x y-1 coefficient
  pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i,
              mixted * xbetai0 * ybetajm1
              + diffusion_y * ydeltajm1
              + convection_y * ybetajm1);
  // x y+1 coefficient
  pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i,
              mixted * xbetai0 * ybetajp1
              + diffusion_y * ydeltajp1
              + convection_y * ybetajp1);
  // x y+2 coefficient
  pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 2) * (M + 1) + i,
              0.0);
  //####
  //####
  //####
```

```
      //####
      //####
      // x-1 y-1 coefficient
      pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i - 1,
                  mixted * xbetaim1 * ybetajm1);
      // x+1 y-1 coefficient
      pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i + 1,
                  mixted * xbetaip1 * ybetajm1);
      // x-1 y+1 coefficient
      pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i - 1,
                  mixted * xbetaim1 * ybetajp1);
      // x+1 y+1 coefficient
      pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i + 1,
                  mixted * xbetaip1 * ybetajp1);
      //####
      //####
      //####
      //####
    }
  }

// For the minimal spot (i==0), there is a Dirichlet condition.
// So the matrix is null on this area. The condition is in the second member
i = 0;
pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i, 0.0);
// Other coefficient are null by default.

// For the maximal spot (i==M), there is a Neumann condition modifying the p
i = M;
Dxi = pointsx[i] - pointsx[i - 1]; // Only coefficient used in x-direction
//Dyjm1 = pointsy[j-1]-pointsy[j-2]; // Not used
Dyj = pointsy[j] - pointsy[j - 1]; // Used
Dyjp1 = pointsy[j + 1] - pointsy[j]; // Used
//Dyjp2 = pointsy[j+2]-pointsy[j+1]; // Not used

// In the x-direction, all coefficient have been simplified.
//yalphajm2 = Dyj/(Dyjm1*(Dyjm1+Dyj)); // Not used
//yalphajm1 = -(Dyjm1+Dyj)/(Dyjm1*Dyj); // Not used
//yalphaj0 = (Dyjm1+2.0*Dyj)/(Dyj*(Dyjm1+Dyj)); // Not used
ybetajm1 = -Dyjp1 / (Dyj * (Dyj + Dyjp1)); // Used in Mixted
ybetaj0 = (Dyjp1 - Dyj) / (Dyj * Dyjp1); // Used in Mixted
```

```
ybetajp1 = Dyj / (Dyjp1 * (Dyj + Dyjp1)); // Used in Mixted
//ygammaj0 = -(2.0*Dyjp1+Dyjp2)/(Dyjp1*(Dyjp1+Dyjp2)); // Not used
//ygammajp1 = (Dyjp1+Dyjp2)/(Dyjp1*Dyjp2); // Not used
//ygammajp2 = -Dyjp1/(Dyjp2*(Dyjp1+Dyjp2)); // Not used
ydeltajm1 = 2.0 / (Dyj * (Dyj + Dyjp1)); // Used in Total : centered scheme
ydeltaj0 = -2.0 / (Dyj * Dyjp1); // Used in Total : centered scheme
ydeltajp1 = 2.0 / (Dyjp1 * (Dyj + Dyjp1)); // Used in Total : centered schem

actualpointy = pointsy[j];
actualpointx = pointsx[i];

order_0 = -r;
convection_x = (r - divid) * actualpointx; // It will be in the coefficient
convection_y = alpha * (beta - actualpointy);
diffusion_x = actualpointx * actualpointx * actualpointy / 2.0;
diffusion_y = gamma * gamma * actualpointy / 2.0;
mixted = rho * gamma * actualpointx * actualpointy;

//std::cout << "i = " << i << " j = " << j << std::endl;
// Matrix Total
{
  // x y coefficient
  pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i,
              //diffusion_x * xdeltai0 // Diffusion term is modified :
              diffusion_x * (- 1.0 / (Dxi * Dxi))
              //+ mixted * xbetai0 * ybetaj0 // No mixed term
              + diffusion_y * ydeltaj0
              //+ convection_x * xbetai0 // No convection_x term. It is in t
              + convection_y * ybetaj0
              + order_0);
  // x-1 y coefficient
  pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i - 1,
              //diffusion_x * xdeltaim1 // Diffusion term is modified :
              diffusion_x * (1.0 / (Dxi * Dxi)));
  //+ mixted * xbetaim1 * ybetaj0 // No mixed term
  //+ convection_x * xbetaim1);// No convection_x term. It is in the coeffic
  // x+1 y coefficient
  // pnl_mat_set(MatrixTotal, j*(M+1)+i, j*(M+1)+i+1, 0.0); // Does not exis
  //diffusion_x * xdeltaip1 // Diffusion term is modified. This term becomes
  //+ mixted * xbetaip1 * ybetaj0 // No mixed term
  //convection_x * xbetaip1); // No convection_x term. It is in the coeffici
```

```
    // x y-2 coefficient
    //pnl_mat_set(MatrixTotal, j*(M+1)+i, (j-2)*(M+1)+i, 0.0); // Does not exi
    // x y-1 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i,
                //mixted * xbetai0 * ybetajm1 // No mixed term
                diffusion_y * ydeltajm1
                + convection_y * ybetajm1);
    // x y+1 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i,
                //mixted * xbetai0 * ybetajp1 // No mixed term
                diffusion_y * ydeltajp1
                + convection_y * ybetajp1);
    // x y+2 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 2) * (M + 1) + i,
                0.0);
    //####
    //####
    //####
    //####
    // No corner coefficients since there is no mixed term.
    //####
    //####
    //####
    //####
  }
}


// First line of volatility (i.e. j=0) : Scheme is forward in vol.
// Many coefficients of the pde are null.
{
  j = 0;
  for (i = 1; i <= M - 1; i++)
    {
      //Dxim1 = pointsx[i-1]-pointsx[i-2]; // Not used
      Dxi = pointsx[i] - pointsx[i - 1]; // Used
      Dxip1 = pointsx[i + 1] - pointsx[i]; // Used
      //Dxip2 = pointsx[i+2]-pointsx[i+1]; // Not used
      //Dyjm1 = pointsy[j-1]-pointsy[j-2]; // Not Used
      //Dyj = pointsy[j]-pointsy[j-1]; // Not Used
      Dyjp1 = pointsy[j + 1] - pointsy[j]; // Used
```

```
        Dyjp2 = pointsy[j + 2] - pointsy[j + 1]; // Used

        //xalphaim2 = Dxi/(Dxim1*(Dxim1+Dxi)); // Not used
        //xalphaim1 = -(Dxim1+Dxi)/(Dxim1*Dxi); // Not used
        //xalphai0 = (Dxim1+2.0*Dxi)/(Dxi*(Dxim1+Dxi)); // Not used
        xbetaim1 = -Dxip1 / (Dxi * (Dxi + Dxip1)); // Used in Mixted
        xbetai0 = (Dxip1 - Dxi) / (Dxi * Dxip1); // Used in Mixted
        xbetaip1 = Dxi / (Dxip1 * (Dxi + Dxip1)); // Used in Mixted
        //xgammai0 = -(2.0*Dxip1+Dxip2)/(Dxip1*(Dxip1+Dxip2)); // Not used
        //xgammaip1 = (Dxip1+Dxip2)/(Dxip1*Dxip2); // Not used
        //xgammaip2 = -Dxip1/(Dxip2*(Dxip1+Dxip2)); // Not used
        //xdeltaim1 = 2.0/(Dxi*(Dxi+Dxip1)); // Not used : Coefficient is nulll
        //xdeltai0 = -2.0/(Dxi*Dxip1); // Not used : Coefficient is nulll
        //xdeltaip1 = 2.0/(Dxip1*(Dxi+Dxip1)); // Not used : Coefficient is null

        //yalphajm2 = Dyj/(Dyjm1*(Dyjm1+Dyj)); // Not used
        //yalphajm1 = -(Dyjm1+Dyj)/(Dyjm1*Dyj); // Not used
        //yalphaj0 = (Dyjm1+2.0*Dyj)/(Dyj*(Dyjm1+Dyj)); // Not used
        //ybetajm1 = -Dyjp1/(Dyj*(Dyj+Dyjp1)); // Not used
        //ybetaj0 = (Dyjp1-Dyj)/(Dyj*Dyjp1); // Not used
        //ybetajp1 = Dyj/(Dyjp1*(Dyj+Dyjp1)); // Not used
        ygammaj0 = -(2.0 * Dyjp1 + Dyjp2) / (Dyjp1 * (Dyjp1 + Dyjp2));
// Used : Forward scheme
        ygammajp1 = (Dyjp1 + Dyjp2) / (Dyjp1 * Dyjp2); // Used : Forward scheme
        ygammajp2 = -Dyjp1 / (Dyjp2 * (Dyjp1 + Dyjp2)); // Used : Forward scheme
        //ydeltajm1 = 2.0/(Dyj*(Dyj+Dyjp1)); // Not used : Coefficient is nulll
        //ydeltaj0 = -2.0/(Dyj*Dyjp1); // Not used : Coefficient is nulll
        //ydeltajp1 = 2.0/(Dyjp1*(Dyj+Dyjp1)); // Not used : Coefficient is null

        actualpointy = pointsy[j]; // =0
        actualpointx = pointsx[i];

        order_0 = -r;
        convection_x = (r - divid) * actualpointx;
        convection_y = alpha * (beta - actualpointy);
        diffusion_x = actualpointx * actualpointx * actualpointy / 2.0; // =0
        diffusion_y = gamma * gamma * actualpointy / 2.0; // =0
        mixted = rho * gamma * actualpointx * actualpointy; // =0


        //std::cout << "i = " << i << " j = " << j << std::endl;
```

```
    //std::cout << "Verification coefficient null = "
    //<< diffusion_x << " " << diffusion_y << " " << mixted << std::endl;

    // MatrixTotal
    {
      // x y coefficient
      pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i,
                  convection_x * xbetai0
                  + convection_y * ygammaj0
                  + order_0);
      // x-1 y coefficient
      pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i - 1,
                  convection_x * xbetaim1);
      // x+1 y coefficient
      pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i + 1,
                  convection_x * xbetaip1);
      // x y-2 coefficient
      //pnl_mat_set(MatrixTotal, j*(M+1)+i, (j-2)*(M+1)+i, 0.0); // Does not
      // x y-1 coefficient
      //pnl_mat_set(MatrixTotal, j*(M+1)+i, (j-1)*(M+1)+i, 0.0); // Does not
      // x y+1 coefficient
      pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i,
                  convection_y * ygammajp1);
      // x y+2 coefficient
      pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 2) * (M + 1) + i,
                  convection_y * ygammajp2);
      //####
      //####
      //####
      //####
      // No corner coefficients since there is no mixed term.
      //####
      //####
      //####
      //####
    }
  }

// For the minimal spot (i==0), there is a Dirichlet condition.
// So the matrix is null on this area. The condition is in the second member
i = 0;
```

```
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i, 0.0);
    // Other coefficient are null by default.

    // For the maximal spot (i==M), there is a Neumann condition modifying the p
    i = M;
    Dxi = pointsx[i] - pointsx[i - 1]; // Only coefficient used in x-direction
    //Dyjm1 = pointsy[j-1]-pointsy[j-2]; // Not Used
    //Dyj = pointsy[j]-pointsy[j-1]; // Not Used
    Dyjp1 = pointsy[j + 1] - pointsy[j]; // Used
    Dyjp2 = pointsy[j + 2] - pointsy[j + 1]; // Used

    // In the x-direction, all coefficient have been simplified.
    //yalphajm2 = Dyj/(Dyjm1*(Dyjm1+Dyj)); // Not used
    //yalphajm1 = -(Dyjm1+Dyj)/(Dyjm1*Dyj); // Not used
    //yalphaj0 = (Dyjm1+2.0*Dyj)/(Dyj*(Dyjm1+Dyj)); // Not used
    //ybetajm1 = -Dyjp1/(Dyj*(Dyj+Dyjp1)); // Not used
    //ybetaj0 = (Dyjp1-Dyj)/(Dyj*Dyjp1); // Not used
    //ybetajp1 = Dyj/(Dyjp1*(Dyj+Dyjp1)); // Not used
    ygammaj0 = -(2.0 * Dyjp1 + Dyjp2) / (Dyjp1 * (Dyjp1 + Dyjp2));
// Used : Forward scheme
    ygammajp1 = (Dyjp1 + Dyjp2) / (Dyjp1 * Dyjp2); // Used : Forward scheme
    ygammajp2 = -Dyjp1 / (Dyjp2 * (Dyjp1 + Dyjp2)); // Used : Forward scheme
    //ydeltajm1 = 2.0/(Dyj*(Dyj+Dyjp1)); // Not used : Coefficient is nulll
    //ydeltaj0 = -2.0/(Dyj*Dyjp1); // Not used : Coefficient is nulll
    //ydeltajp1 = 2.0/(Dyjp1*(Dyj+Dyjp1)); // Not used : Coefficient is nulll

    actualpointy = pointsy[j]; // =0
    actualpointx = pointsx[i];

    order_0 = -r;
    convection_x = (r - divid) * actualpointx; // It will be in the coefficient
    convection_y = alpha * (beta - actualpointy);
    diffusion_x = actualpointx * actualpointx * actualpointy / 2.0; // = 0
    diffusion_y = gamma * gamma * actualpointy / 2.0; // =0
    mixted = rho * gamma * actualpointx * actualpointy; // =0

    //std::cout << "i = " << i << " j = " << j << std::endl;
    //std::cout << "Verification coefficient null (border spot max)= "
    //<< diffusion_x << " " << diffusion_y << " " << mixted << std::endl;

    // MatrixTotal
```

```
  {
    // x y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i,
                // convection_x * xbetai0 // No convection_x term. It is in th
                + convection_y * ygammaj0
                + order_0);
    // x-1 y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i - 1, 0.0);
    //+ convection_x * xbetaim1); // No convection_x term. It is in the coeffi
    // x+1 y coefficient
    //pnl_mat_set(MatrixTotal, j*(M+1)+i, j*(M+1)+i+1, 0.0); // Does not exist
    //diffusion_x * xdeltaip1 // Diffusion term is modified. This term becomes
    //+ mixted * xbetaip1 * ybetaj0 // No mixed term
    //convection_x * xbetaip1); // No convection_x term. It is in the coeffici
    // x y-2 coefficient
    //pnl_mat_set(MatrixTotal, j*(M+1)+i, (j-2)*(M+1)+i, 0.0); // Does not exi
    // x y-1 coefficient
    //pnl_mat_set(MatrixTotal, j*(M+1)+i, (j-1)*(M+1)+i, 0.0); // Does not exi
    // x y+1 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i,
                convection_y * ygammajp1);
    // x y+2 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 2) * (M + 1) + i,
                convection_y * ygammajp2);
    //####
    //####
    //####
    //####
    // No corner coefficients since there is no mixed term.
    //####
    //####
    //####
    //####
  }
}
// Last line of volatility (i.e. j=N) : Dirichlet condition
{
  j = N;
  for (i = 0; i <= M; i++)
    {
      // For all spot, there is a Dirichlet condition.
```

```
        // So the matrix is null on this area. The condition is in the second me
        pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i, 0.0);
        // Other coefficient are null by default.
      }
    }

    // Compute the solution "Sortie" of
    // Sortie = VectTwo - coeff * (MatrixTotal * Sortie + Boundary conditions)
    // which is also the solution of
    // (Id + coeff * MatrixTotal) Sortie = VectTwo - coeff * Boundary conditions

    // We do TWO backward Euler scheme iteration, so we need to divide coeff by 2.
    // Sortie = VectTwo - coeff/2.0 * (MatrixTotal * Sortie + Boundary conditions)
    // which is also the solution of
    // (Id + coeff/2.0 * MatrixTotal) Sortie = VectTwo - coeff/2.0 * Boundary cond
    // Sortie2 = Sortie - coeff/2.0 * (MatrixTotal * Sortie2 + Boundary conditions
    // which is also the solution of
    // (Id + coeff/2.0 * MatrixTotal) Sortie2 = Sortie - coeff/2.0 * Boundary cond

    coeff = coeff / 2.0;

  //  pnl_mat_print(MatrixTotal);

    // Creation of identity matrix.
    Ones = pnl_vect_create_from_double((M + 1) * (N + 1), 1.0);
    SecondMember = pnl_vect_create((M + 1) * (N + 1));
    Working_Matrix = pnl_mat_create_diag(Ones); // Identity matrix.
    // Multiplication by coeff.
    pnl_mat_mult_double(MatrixTotal, coeff);
    // Sum with the identity matrix. Result is in the first argument.
    pnl_mat_plus_mat(Working_Matrix, MatrixTotal);

    // Computation with LU computation
    permutation = pnl_permutation_create((M + 1) * (N + 1));
    pnl_mat_lu(Working_Matrix, permutation);

    // Add the boundary conditions.
    pnl_vect_clone(SecondMember, VectTwo);
    modify_solution_with_neumann_boundary_condition(
      r, divid, deltat / 2.0, call_or_put, M, pointsx, N, pointsy, coeff, SecondMe
    // First implicit Euler iteration
```

```
  pnl_mat_lu_syslin(Ones, Working_Matrix, permutation, SecondMember);
  // Add the boundary conditions.
  modify_solution_with_neumann_boundary_condition(
    r, divid, deltat, call_or_put, M, pointsx, N, pointsy, coeff, Ones, Ones);
  // Second implicit Euler iteration
  pnl_mat_lu_syslin(Sortie, Working_Matrix, permutation, Ones);

  // Free memory
  pnl_vect_int_free(&permutation);
  pnl_mat_free(&MatrixTotal);
  pnl_mat_free(&Working_Matrix);
  pnl_vect_free(&SecondMember);
  pnl_vect_free(&Ones);
}

static void do_rannacher_iteration(double r, double divid, double time,
                                   double alpha, double beta, double gamma, doub
                                   int call_or_put,
                                   int M, double *pointsx, int N, double *points
                                   double coeff, PnlVect *VectTwo, PnlVect *Sort
{

  // First, we build the full matrix of the pde.
  // But here, we need to "inverse" the full matrix, so we must build it in its

  // This procedure computes the solution "Sortie" of
  // Sortie = VectTwo - coeff * (MatrixTotal * Sortie + Boundary conditions)
  // which is also the solution of
  // (Id + coeff * MatrixTotal) Sortie = VectTwo - coeff * Boundary conditions

  // Be careful, it is awkward but there is a minus (before coeff in the second

  // Build lines of constant volatility and increasing spot.
  //    *   6
  //  * * * 9 5 10
  //  * * *     <=>1 0 2
  //  * * * 7 4 8
  //    *   3
  // MatrixTotal(0, . ) <= (i,j)
  // MatrixTotal(1, . ) <= (i-1,j)
  // MatrixTotal(2, . ) <= (i+1,j)
```

```
// MatrixTotal(3, . ) <= (i,j-2)
// MatrixTotal(4, . ) <= (i,j-1)
// MatrixTotal(5, . ) <= (i,j+1)
// MatrixTotal(6, . ) <= (i,j+2)
// MatrixTotal(7, . ) <= (i-1,j-1)
// MatrixTotal(8, . ) <= (i+1,j-1)
// MatrixTotal(9, . ) <= (i-1,j+1)
// MatrixTotal(10, . ) <= (i+1,j+1)


double actualpointx;
double actualpointy;
double Dxi, Dxip1;
double Dyjm1, Dyj, Dyjp1, Dyjp2;
double convection_x, diffusion_x, convection_y, diffusion_y, mixted, order_0;
// Coefficients
double xbetaim1, xbetai0, xbetaip1;
double xdeltaim1, xdeltai0, xdeltaip1;
double yalphajm2, yalphajm1, yalphaj0;
double ybetajm1, ybetaj0, ybetajp1;
double ygammaj0, ygammajp1, ygammajp2;
double ydeltajm1, ydeltaj0, ydeltajp1;

int i, j;

PnlVect *Ones;
PnlVect *SecondMember;
PnlMat *Working_Matrix;
PnlMat *MatrixTotal;
PnlVectInt *permutation;

MatrixTotal = pnl_mat_create((M + 1) * (N + 1), (M + 1) * (N + 1));

// Interior domain : Since scheme is centered or backward in vol : 2 <= j <= N
for (j = 2; j <= N - 1; j++)
  {
    // Interior domain : Since scheme is centered in spot : 1 <= i <= M-1
    for (i = 1; i <= M - 1; i++)
      {
        //Dxim1 = pointsx[i-1]-pointsx[i-2]; // Not used
        Dxi = pointsx[i] - pointsx[i - 1]; // Used
```

```
            Dxip1 = pointsx[i + 1] - pointsx[i]; // Used
            //Dxip2 = pointsx[i+2]-pointsx[i+1]; // Not used
            Dyjm1 = pointsy[j - 1] - pointsy[j - 2]; // Used
            Dyj = pointsy[j] - pointsy[j - 1]; // Used
            Dyjp1 = pointsy[j + 1] - pointsy[j]; // Used
            //Dyjp2 = pointsy[j+2]-pointsy[j+1]; // Not used

            //xalphaim2 = Dxi/(Dxim1*(Dxim1+Dxi)); // Not used
            //xalphaim1 = -(Dxim1+Dxi)/(Dxim1*Dxi); // Not used
            //xalphai0 = (Dxim1+2.0*Dxi)/(Dxi*(Dxim1+Dxi)); // Not used
            xbetaim1 = -Dxip1 / (Dxi * (Dxi + Dxip1)); // Used in Mixted
            xbetai0 = (Dxip1 - Dxi) / (Dxi * Dxip1); // Used in Mixted
            xbetaip1 = Dxi / (Dxip1 * (Dxi + Dxip1)); // Used in Mixted
            //xgammai0 = -(2.0*Dxip1+Dxip2)/(Dxip1*(Dxip1+Dxip2));
// Not used
            //xgammaip1 = (Dxip1+Dxip2)/(Dxip1*Dxip2); // Not used
            //xgammaip2 = -Dxip1/(Dxip2*(Dxip1+Dxip2)); // Not used
            xdeltaim1 = 2.0 / (Dxi * (Dxi + Dxip1)); // Used in Total : centered s
            xdeltai0 = -2.0 / (Dxi * Dxip1); // Used in Total : centered scheme
            xdeltaip1 = 2.0 / (Dxip1 * (Dxi + Dxip1)); // Used in Total : centered

            yalphajm2 = Dyj / (Dyjm1 * (Dyjm1 + Dyj)); // Used in Total : upwind s
            yalphajm1 = -(Dyjm1 + Dyj) / (Dyjm1 * Dyj); // Used in Total : upwind
            yalphaj0 = (Dyjm1 + 2.0 * Dyj) / (Dyj * (Dyjm1 + Dyj));
// Used in Total : upwind scheme
            ybetajm1 = -Dyjp1 / (Dyj * (Dyj + Dyjp1)); // Used in Mixted
            ybetaj0 = (Dyjp1 - Dyj) / (Dyj * Dyjp1); // Used in Mixted
            ybetajp1 = Dyj / (Dyjp1 * (Dyj + Dyjp1)); // Used in Mixted
            //ygammaj0 = -(2.0*Dyjp1+Dyjp2)/(Dyjp1*(Dyjp1+Dyjp2));
// Not used
            //ygammajp1 = (Dyjp1+Dyjp2)/(Dyjp1*Dyjp2); // Not used
            //ygammajp2 = -Dyjp1/(Dyjp2*(Dyjp1+Dyjp2)); // Not used
            ydeltajm1 = 2.0 / (Dyj * (Dyj + Dyjp1)); // Used in Total : centered s
            ydeltaj0 = -2.0 / (Dyj * Dyjp1); // Used in Total : centered scheme
            ydeltajp1 = 2.0 / (Dyjp1 * (Dyj + Dyjp1)); // Used in Total : centered

            actualpointy = pointsy[j];
            actualpointx = pointsx[i];

            order_0 = -r;
            convection_x = (r - divid) * actualpointx;
```

```
convection_y = alpha * (beta - actualpointy);
diffusion_x = actualpointx * actualpointx * actualpointy / 2.0;
diffusion_y = gamma * gamma * actualpointy / 2.0;
mixted = rho * gamma * actualpointx * actualpointy;

if ((actualpointy > 1) && (convection_y < 0))
  // In this case, we use the upwind scheme whenever the flow
  // in the v-direction is towards v=V. (i.e. alpha(beta - v) < 0).
  {
    // MatrixTotal
    // x y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i,
                diffusion_x * xdeltai0
                + mixted * xbetai0 * ybetaj0
                + diffusion_y * ydeltaj0
                + convection_x * xbetai0
                + convection_y * yalphaj0
                + order_0);
    // x-1 y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i - 1,
                diffusion_x * xdeltaim1
                + mixted * xbetaim1 * ybetaj0
                + convection_x * xbetaim1);
    // x+1 y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i + 1,
                diffusion_x * xdeltaip1
                + mixted * xbetaip1 * ybetaj0
                + convection_x * xbetaip1);
    // x y-2 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 2) * (M + 1) + i,
                convection_y * yalphajm2);
    // x y-1 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i,
                mixted * xbetai0 * ybetajm1
                + diffusion_y * ydeltajm1
                + convection_y * yalphajm1);
    // x y+1 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i,
                mixted * xbetai0 * ybetajp1
                + diffusion_y * ydeltajp1);
    // x y+2 coefficient
```

```
    if (j < N - 1) // Coefficient does not exist if j=N-1.
      {
        pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 2) * (M + 1) +
      }
    // x-1 y-1 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i -
                mixted * xbetaim1 * ybetajm1);
    // x+1 y-1 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i +
                mixted * xbetaip1 * ybetajm1);
    // x-1 y+1 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i -
                mixted * xbetaim1 * ybetajp1);
    // x+1 y+1 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i +
                mixted * xbetaip1 * ybetajp1);
  }
else // centered scheme in volatility
  {
    // MatrixTotal
    // x y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i,
                diffusion_x * xdeltai0
                + mixted * xbetai0 * ybetaj0
                + diffusion_y * ydeltaj0
                + convection_x * xbetai0
                + convection_y * ybetaj0
                + order_0);
    // x-1 y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i - 1,
                diffusion_x * xdeltaim1
                + mixted * xbetaim1 * ybetaj0
                + convection_x * xbetaim1);
    // x+1 y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i + 1,
                diffusion_x * xdeltaip1
                + mixted * xbetaip1 * ybetaj0
                + convection_x * xbetaip1);
    // x y-2 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 2) * (M + 1) + i,
                0.0);
```

```
        // x y-1 coefficient
        pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i,
                    mixted * xbetai0 * ybetajm1
                    + diffusion_y * ydeltajm1
                    + convection_y * ybetajm1);
        // x y+1 coefficient
        pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i,
                    mixted * xbetai0 * ybetajp1
                    + diffusion_y * ydeltajp1
                    + convection_y * ybetajp1);
        // x y+2 coefficient
        if (j < N - 1) // Coefficient does not exist if j=N-1.
          {
            pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 2) * (M + 1) +
          }
        // x-1 y-1 coefficient
        pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i -
                    mixted * xbetaim1 * ybetajm1);
        // x+1 y-1 coefficient
        pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i +
                    mixted * xbetaip1 * ybetajm1);
        // x-1 y+1 coefficient
        pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i -
                    mixted * xbetaim1 * ybetajp1);
        // x+1 y+1 coefficient
        pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i +
                    mixted * xbetaip1 * ybetajp1);
      }
  }

// For the minimal spot (i==0), there is a Dirichlet condition.
// So the matrix is null on this area. The condition is in the second memb
i = 0;
pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i, 0.0);
// Other coefficient are null by default.

// For the maximal spot (i==M), there is a Neumann condition modifying the
i = M;
Dxi = pointsx[i] - pointsx[i - 1]; // Only coefficient used in x-direction
Dyjm1 = pointsy[j - 1] - pointsy[j - 2]; // Used
Dyj = pointsy[j] - pointsy[j - 1]; // Used
```

```
Dyjp1 = pointsy[j + 1] - pointsy[j]; // Used
//Dyjp2 = pointsy[j+2]-pointsy[j+1]; // Not used

// In the x-direction, all coefficient have been simplified.
yalphajm2 = Dyj / (Dyjm1 * (Dyjm1 + Dyj)); // Used in Total : upwind schem
yalphajm1 = -(Dyjm1 + Dyj) / (Dyjm1 * Dyj); // Used in Total : upwind sche
yalphaj0 = (Dyjm1 + 2.0 * Dyj) / (Dyj * (Dyjm1 + Dyj)); // Used in Total :
ybetajm1 = -Dyjp1 / (Dyj * (Dyj + Dyjp1)); // Used in Mixted
ybetaj0 = (Dyjp1 - Dyj) / (Dyj * Dyjp1); // Used in Mixted
ybetajp1 = Dyj / (Dyjp1 * (Dyj + Dyjp1)); // Used in Mixted
//ygammaj0 = -(2.0*Dyjp1+Dyjp2)/(Dyjp1*(Dyjp1+Dyjp2)); // Not used
//ygammajp1 = (Dyjp1+Dyjp2)/(Dyjp1*Dyjp2); // Not used
//ygammajp2 = -Dyjp1/(Dyjp2*(Dyjp1+Dyjp2)); // Not used
ydeltajm1 = 2.0 / (Dyj * (Dyj + Dyjp1)); // Used in Total : centered schem
ydeltaj0 = -2.0 / (Dyj * Dyjp1); // Used in Total : centered scheme
ydeltajp1 = 2.0 / (Dyjp1 * (Dyj + Dyjp1)); // Used in Total : centered sch

actualpointy = pointsy[j];
actualpointx = pointsx[i];

order_0 = -r;
convection_x = (r - divid) * actualpointx; // It will be in the coefficien
convection_y = alpha * (beta - actualpointy);
diffusion_x = actualpointx * actualpointx * actualpointy / 2.0;
diffusion_y = gamma * gamma * actualpointy / 2.0;
mixted = rho * gamma * actualpointx * actualpointy;

if ((actualpointy > 1) && (convection_y < 0))
  // In this case, we use the upwind scheme whenever the flow
  // in the v-direction is towards v=V. (i.e. alpha(beta - v) < 0).
  {
    // MatrixTotal
    // x y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i,
                //diffusion_x * xdeltai0 // Diffusion term is modified :
                diffusion_x * (- 1.0 / (Dxi * Dxi))
                //+ mixted * xbetai0 * ybetaj0 // No mixed term
                + diffusion_y * ydeltaj0
                //+ convection_x * xbetai0 // No convection_x term. It is
                + convection_y * yalphaj0
                + order_0);
```

```
    // x-1 y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i - 1,
                //diffusion_x * xdeltaim1 // Diffusion term is modified :
                diffusion_x * (1.0 / (Dxi * Dxi)));
    //+ mixted * xbetaim1 * ybetaj0 // No mixed term
    //+ convection_x * xbetaim1); // No convection_x term. It is in the co
    // x+1 y coefficient
    // pnl_mat_set(MatrixTotal, j*(M+1)+i, 2, 0.0); // Does not exist !!!
    //diffusion_x * xdeltaip1 // Diffusion term is modified. This term bec
    //+ mixted * xbetaip1 * ybetaj0 // No mixed term
    //convection_x * xbetaip1);// No convection_x term. It is in the coeff
    // x y-2 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 2) * (M + 1) + i,
                convection_y * yalphajm2);
    // x y-1 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i,
                //mixted * xbetai0 * ybetajm1 // No mixed term
                diffusion_y * ydeltajm1
                + convection_y * yalphajm1);
    // x y+1 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i,
                //mixted * xbetai0 * ybetajp1 // No mixed term
                diffusion_y * ydeltajp1);
    // x y+2 coefficient
    if (j < N - 1) // Coefficient does not exist if j=N-1.
      {
        pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 2) * (M + 1) + i, 0
      }

    // No corner coefficients since there is no mixed term.
  }
else // centered scheme in volatility
  {
    // MatrixTotal
    // x y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i,
                //diffusion_x * xdeltai0 // Diffusion term is modified :
                diffusion_x * (- 1.0 / (Dxi * Dxi))
                //+ mixted * xbetai0 * ybetaj0 // No mixed term
                + diffusion_y * ydeltaj0
                //+ convection_x * xbetai0 // No convection_x term. It is
```

```
                + convection_y * ybetaj0
                + order_0);
        // x-1 y coefficient
        pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i - 1,
                //diffusion_x * xdeltaim1 // Diffusion term is modified :
                diffusion_x * (1.0 / (Dxi * Dxi)));
        //+ mixted * xbetaim1 * ybetaj0 // No mixed term
        //+ convection_x * xbetaim1);// No convection_x term. It is in the coe
        // x+1 y coefficient
        // pnl_mat_set(MatrixTotal, j*(M+1)+i, j*(M+1)+i+1, 0.0); // Does not
        //diffusion_x * xdeltaip1 // Diffusion term is modified. This term bec
        //+ mixted * xbetaip1 * ybetaj0 // No mixed term
        //convection_x * xbetaip1); // No convection_x term. It is in the coef
        // x y-2 coefficient
        pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 2) * (M + 1) + i,
                0.0);
        // x y-1 coefficient
        pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i,
                //mixted * xbetai0 * ybetajm1 // No mixed term
                diffusion_y * ydeltajm1
                + convection_y * ybetajm1);
        // x y+1 coefficient
        pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i,
                //mixted * xbetai0 * ybetajp1 // No mixed term
                diffusion_y * ydeltajp1
                + convection_y * ybetajp1);
        // x y+2 coefficient
        if (j < N - 1) // Coefficient does not exist if j=N-1.
          {
            pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 2) * (M + 1) + i, 0
          }

        // No corner coefficients since there is no mixed term.
      }
  }
// Second line of volatility (i.e. j=1) : Scheme is necessarily centered in vo
{
  j = 1;
  for (i = 1; i <= M - 1; i++)
    {
      //Dxim1 = pointsx[i-1]-pointsx[i-2]; // Not used
```

```
Dxi = pointsx[i] - pointsx[i - 1]; // Used
Dxip1 = pointsx[i + 1] - pointsx[i]; // Used
//Dxip2 = pointsx[i+2]-pointsx[i+1]; // Not used
//Dyjm1 = pointsy[j-1]-pointsy[j-2]; // Not used
Dyj = pointsy[j] - pointsy[j - 1]; // Used
Dyjp1 = pointsy[j + 1] - pointsy[j]; // Used
//Dyjp2 = pointsy[j+2]-pointsy[j+1]; // Not used

//xalphaim2 = Dxi/(Dxim1*(Dxim1+Dxi)); // Not used
//xalphaim1 = -(Dxim1+Dxi)/(Dxim1*Dxi); // Not used
//xalphai0 = (Dxim1+2.0*Dxi)/(Dxi*(Dxim1+Dxi)); // Not used
xbetaim1 = -Dxip1 / (Dxi * (Dxi + Dxip1)); // Used in Mixted
xbetai0 = (Dxip1 - Dxi) / (Dxi * Dxip1); // Used in Mixted
xbetaip1 = Dxi / (Dxip1 * (Dxi + Dxip1)); // Used in Mixted
//xgammai0 = -(2.0*Dxip1+Dxip2)/(Dxip1*(Dxip1+Dxip2)); // Not used
//xgammaip1 = (Dxip1+Dxip2)/(Dxip1*Dxip2); // Not used
//xgammaip2 = -Dxip1/(Dxip2*(Dxip1+Dxip2)); // Not used
xdeltaim1 = 2.0 / (Dxi * (Dxi + Dxip1)); // Used in Total : centered sch
xdeltai0 = -2.0 / (Dxi * Dxip1); // Used in Total : centered scheme
xdeltaip1 = 2.0 / (Dxip1 * (Dxi + Dxip1)); // Used in Total : centered s

//yalphajm2 = Dyj/(Dyjm1*(Dyjm1+Dyj)); // Not used
//yalphajm1 = -(Dyjm1+Dyj)/(Dyjm1*Dyj); // Not used
//yalphaj0 = (Dyjm1+2.0*Dyj)/(Dyj*(Dyjm1+Dyj)); // Not used
ybetajm1 = -Dyjp1 / (Dyj * (Dyj + Dyjp1)); // Used in Mixted
ybetaj0 = (Dyjp1 - Dyj) / (Dyj * Dyjp1); // Used in Mixted
ybetajp1 = Dyj / (Dyjp1 * (Dyj + Dyjp1)); // Used in Mixted
//ygammaj0 = -(2.0*Dyjp1+Dyjp2)/(Dyjp1*(Dyjp1+Dyjp2)); // Not used
//ygammajp1 = (Dyjp1+Dyjp2)/(Dyjp1*Dyjp2); // Not used
//ygammajp2 = -Dyjp1/(Dyjp2*(Dyjp1+Dyjp2)); // Not used
ydeltajm1 = 2.0 / (Dyj * (Dyj + Dyjp1)); // Used in Total : centered sch
ydeltaj0 = -2.0 / (Dyj * Dyjp1); // Used in Total : centered scheme
ydeltajp1 = 2.0 / (Dyjp1 * (Dyj + Dyjp1)); // Used in Total : centered s

actualpointy = pointsy[j];
actualpointx = pointsx[i];

order_0 = -r;
convection_x = r * actualpointx;
convection_y = alpha * (beta - actualpointy);
diffusion_x = actualpointx * actualpointx * actualpointy / 2.0;
```

```
diffusion_y = gamma * gamma * actualpointy / 2.0;
mixted = rho * gamma * actualpointx * actualpointy;

// MatrixTotal
{
  // x y coefficient
  pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i,
              diffusion_x * xdeltai0
              + mixted * xbetai0 * ybetaj0
              + diffusion_y * ydeltaj0
              + convection_x * xbetai0
              + convection_y * ybetaj0
              + order_0);
  // x-1 y coefficient
  pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i - 1,
              diffusion_x * xdeltaim1
              + mixted * xbetaim1 * ybetaj0
              + convection_x * xbetaim1);
  // x+1 y coefficient
  pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i + 1,
              diffusion_x * xdeltaip1
              + mixted * xbetaip1 * ybetaj0
              + convection_x * xbetaip1);
  // x y-2 coefficient
  //pnl_mat_set(MatrixTotal, j*(M+1)+i, (j-2)*(M+1)+i, 0.0); // Does not
  // x y-1 coefficient
  pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i,
              mixted * xbetai0 * ybetajm1
              + diffusion_y * ydeltajm1
              + convection_y * ybetajm1);
  // x y+1 coefficient
  pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i,
              mixted * xbetai0 * ybetajp1
              + diffusion_y * ydeltajp1
              + convection_y * ybetajp1);
  // x y+2 coefficient
  pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 2) * (M + 1) + i,
              0.0);
  // x-1 y-1 coefficient
  pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i - 1,
              mixted * xbetaim1 * ybetajm1);
```

```
      // x+1 y-1 coefficient
      pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i + 1,
                  mixted * xbetaip1 * ybetajm1);
      // x-1 y+1 coefficient
      pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i - 1,
                  mixted * xbetaim1 * ybetajp1);
      // x+1 y+1 coefficient
      pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i + 1,
                  mixted * xbetaip1 * ybetajp1);
    }
  }

// For the minimal spot (i==0), there is a Dirichlet condition.
// So the matrix is null on this area. The condition is in the second member
i = 0;
pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i, 0.0);
// Other coefficient are null by default.

// For the maximal spot (i==M), there is a Neumann condition modifying the p
i = M;
Dxi = pointsx[i] - pointsx[i - 1]; // Only coefficient used in x-direction
//Dyjm1 = pointsy[j-1]-pointsy[j-2]; // Not used
Dyj = pointsy[j] - pointsy[j - 1]; // Used
Dyjp1 = pointsy[j + 1] - pointsy[j]; // Used
//Dyjp2 = pointsy[j+2]-pointsy[j+1]; // Not used

// In the x-direction, all coefficient have been simplified.
//yalphajm2 = Dyj/(Dyjm1*(Dyjm1+Dyj)); // Not used
//yalphajm1 = -(Dyjm1+Dyj)/(Dyjm1*Dyj); // Not used
//yalphaj0 = (Dyjm1+2.0*Dyj)/(Dyj*(Dyjm1+Dyj)); // Not used
ybetajm1 = -Dyjp1 / (Dyj * (Dyj + Dyjp1)); // Used in Mixted
ybetaj0 = (Dyjp1 - Dyj) / (Dyj * Dyjp1); // Used in Mixted
ybetajp1 = Dyj / (Dyjp1 * (Dyj + Dyjp1)); // Used in Mixted
//ygammaj0 = -(2.0*Dyjp1+Dyjp2)/(Dyjp1*(Dyjp1+Dyjp2)); // Not used
//ygammajp1 = (Dyjp1+Dyjp2)/(Dyjp1*Dyjp2); // Not used
//ygammajp2 = -Dyjp1/(Dyjp2*(Dyjp1+Dyjp2)); // Not used
ydeltajm1 = 2.0 / (Dyj * (Dyj + Dyjp1)); // Used in Total : centered scheme
ydeltaj0 = -2.0 / (Dyj * Dyjp1); // Used in Total : centered scheme
ydeltajp1 = 2.0 / (Dyjp1 * (Dyj + Dyjp1)); // Used in Total : centered schem

actualpointy = pointsy[j];
```

```
actualpointx = pointsx[i];

order_0 = -r;
convection_x = (r - divid) * actualpointx; // It will be in the coefficient
convection_y = alpha * (beta - actualpointy);
diffusion_x = actualpointx * actualpointx * actualpointy / 2.0;
diffusion_y = gamma * gamma * actualpointy / 2.0;
mixted = rho * gamma * actualpointx * actualpointy;

// Matrix Total
{
  // x y coefficient
  pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i,
              //diffusion_x * xdeltai0 // Diffusion term is modified :
              diffusion_x * (- 1.0 / (Dxi * Dxi))
              //+ mixted * xbetai0 * ybetaj0 // No mixed term
              + diffusion_y * ydeltaj0
              //+ convection_x * xbetai0 // No convection_x term. It is in t
              + convection_y * ybetaj0
              + order_0);
  // x-1 y coefficient
  pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i - 1,
              //diffusion_x * xdeltaim1 // Diffusion term is modified :
              diffusion_x * (1.0 / (Dxi * Dxi)));
  //+ mixted * xbetaim1 * ybetaj0 // No mixed term
  //+ convection_x * xbetaim1);// No convection_x term. It is in the coeffic
  // x+1 y coefficient
  // pnl_mat_set(MatrixTotal, j*(M+1)+i, j*(M+1)+i+1, 0.0); // Does not exis
  //diffusion_x * xdeltaip1 // Diffusion term is modified. This term becomes
  //+ mixted * xbetaip1 * ybetaj0 // No mixed term
  //convection_x * xbetaip1); // No convection_x term. It is in the coeffici
  // x y-2 coefficient
  //pnl_mat_set(MatrixTotal, j*(M+1)+i, (j-2)*(M+1)+i, 0.0); // Does not exi
  // x y-1 coefficient
  pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j - 1) * (M + 1) + i,
              //mixted * xbetai0 * ybetajm1 // No mixed term
              diffusion_y * ydeltajm1
              + convection_y * ybetajm1);
  // x y+1 coefficient
  pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i,
              //mixted * xbetai0 * ybetajp1 // No mixed term
```

```
                diffusion_y * ydeltajp1
                + convection_y * ybetajp1);
    // x y+2 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 2) * (M + 1) + i,
                0.0);

    // No corner coefficients since there is no mixed term.
  }
}
// First line of volatility (i.e. j=0) : Scheme is forward in vol.
// Many coefficients of the pde are null.
{
  j = 0;
  for (i = 1; i <= M - 1; i++)
    {
      //Dxim1 = pointsx[i-1]-pointsx[i-2]; // Not used
      Dxi = pointsx[i] - pointsx[i - 1]; // Used
      Dxip1 = pointsx[i + 1] - pointsx[i]; // Used
      //Dxip2 = pointsx[i+2]-pointsx[i+1]; // Not used
      //Dyjm1 = pointsy[j-1]-pointsy[j-2]; // Not Used
      //Dyj = pointsy[j]-pointsy[j-1]; // Not Used
      Dyjp1 = pointsy[j + 1] - pointsy[j]; // Used
      Dyjp2 = pointsy[j + 2] - pointsy[j + 1]; // Used

      //xalphaim2 = Dxi/(Dxim1*(Dxim1+Dxi)); // Not used
      //xalphaim1 = -(Dxim1+Dxi)/(Dxim1*Dxi); // Not used
      //xalphai0 = (Dxim1+2.0*Dxi)/(Dxi*(Dxim1+Dxi)); // Not used
      xbetaim1 = -Dxip1 / (Dxi * (Dxi + Dxip1)); // Used in Mixted
      xbetai0 = (Dxip1 - Dxi) / (Dxi * Dxip1); // Used in Mixted
      xbetaip1 = Dxi / (Dxip1 * (Dxi + Dxip1)); // Used in Mixted
      //xgammai0 = -(2.0*Dxip1+Dxip2)/(Dxip1*(Dxip1+Dxip2)); // Not used
      //xgammaip1 = (Dxip1+Dxip2)/(Dxip1*Dxip2); // Not used
      //xgammaip2 = -Dxip1/(Dxip2*(Dxip1+Dxip2)); // Not used
      //xdeltaim1 = 2.0/(Dxi*(Dxi+Dxip1)); // Not used : Coefficient is nulll
      //xdeltai0 = -2.0/(Dxi*Dxip1); // Not used : Coefficient is nulll
      //xdeltaip1 = 2.0/(Dxip1*(Dxi+Dxip1)); // Not used : Coefficient is null

      //yalphajm2 = Dyj/(Dyjm1*(Dyjm1+Dyj)); // Not used
      //yalphajm1 = -(Dyjm1+Dyj)/(Dyjm1*Dyj); // Not used
      //yalphaj0 = (Dyjm1+2.0*Dyj)/(Dyj*(Dyjm1+Dyj)); // Not used
      //ybetajm1 = -Dyjp1/(Dyj*(Dyj+Dyjp1)); // Not used
```

```
        //ybetaj0 = (Dyjp1-Dyj)/(Dyj*Dyjp1); // Not used
        //ybetajp1 = Dyj/(Dyjp1*(Dyj+Dyjp1)); // Not used
        ygammaj0 = -(2.0 * Dyjp1 + Dyjp2) / (Dyjp1 * (Dyjp1 + Dyjp2));
// Used : Forward scheme
        ygammajp1 = (Dyjp1 + Dyjp2) / (Dyjp1 * Dyjp2); // Used : Forward scheme
        ygammajp2 = -Dyjp1 / (Dyjp2 * (Dyjp1 + Dyjp2)); // Used : Forward scheme
        //ydeltajm1 = 2.0/(Dyj*(Dyj+Dyjp1)); // Not used : Coefficient is nulll
        //ydeltaj0 = -2.0/(Dyj*Dyjp1); // Not used : Coefficient is nulll
        //ydeltajp1 = 2.0/(Dyjp1*(Dyj+Dyjp1)); // Not used : Coefficient is null

        actualpointy = pointsy[j]; // =0
        actualpointx = pointsx[i];

        order_0 = -r;
        convection_x = (r - divid) * actualpointx;
        convection_y = alpha * (beta - actualpointy);
        diffusion_x = actualpointx * actualpointx * actualpointy / 2.0; // =0
        diffusion_y = gamma * gamma * actualpointy / 2.0; // =0
        mixted = rho * gamma * actualpointx * actualpointy; // =0

        // MatrixTotal
        {
          // x y coefficient
          pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i,
                      convection_x * xbetai0
                      + convection_y * ygammaj0
                      + order_0);
          // x-1 y coefficient
          pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i - 1,
                      convection_x * xbetaim1);
          // x+1 y coefficient
          pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i + 1,
                      convection_x * xbetaip1);
          // x y-2 coefficient
          //pnl_mat_set(MatrixTotal, j*(M+1)+i, (j-2)*(M+1)+i, 0.0); // Does not
          // x y-1 coefficient
          //pnl_mat_set(MatrixTotal, j*(M+1)+i, (j-1)*(M+1)+i, 0.0); // Does not
          // x y+1 coefficient
          pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i,
                      convection_y * ygammajp1);
          // x y+2 coefficient
```

```
            pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 2) * (M + 1) + i,
                        convection_y * ygammajp2);

            // No corner coefficients since there is no mixed term.
          }
        }

      // For the minimal spot (i==0), there is a Dirichlet condition.
      // So the matrix is null on this area. The condition is in the second member
      i = 0;
      pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i, 0.0);
      // Other coefficient are null by default.

      // For the maximal spot (i==M), there is a Neumann condition modifying the p
      i = M;
      Dxi = pointsx[i] - pointsx[i - 1]; // Only coefficient used in x-direction
      //Dyjm1 = pointsy[j-1]-pointsy[j-2]; // Not Used
      //Dyj = pointsy[j]-pointsy[j-1]; // Not Used
      Dyjp1 = pointsy[j + 1] - pointsy[j]; // Used
      Dyjp2 = pointsy[j + 2] - pointsy[j + 1]; // Used

      // In the x-direction, all coefficient have been simplified.
      //yalphajm2 = Dyj/(Dyjm1*(Dyjm1+Dyj)); // Not used
      //yalphajm1 = -(Dyjm1+Dyj)/(Dyjm1*Dyj); // Not used
      //yalphaj0 = (Dyjm1+2.0*Dyj)/(Dyj*(Dyjm1+Dyj)); // Not used
      //ybetajm1 = -Dyjp1/(Dyj*(Dyj+Dyjp1)); // Not used
      //ybetaj0 = (Dyjp1-Dyj)/(Dyj*Dyjp1); // Not used
      //ybetajp1 = Dyj/(Dyjp1*(Dyj+Dyjp1)); // Not used
      ygammaj0 = -(2.0 * Dyjp1 + Dyjp2) / (Dyjp1 * (Dyjp1 + Dyjp2));
// Used : Forward scheme
      ygammajp1 = (Dyjp1 + Dyjp2) / (Dyjp1 * Dyjp2); // Used : Forward scheme
      ygammajp2 = -Dyjp1 / (Dyjp2 * (Dyjp1 + Dyjp2)); // Used : Forward scheme
      //ydeltajm1 = 2.0/(Dyj*(Dyj+Dyjp1)); // Not used : Coefficient is nulll
      //ydeltaj0 = -2.0/(Dyj*Dyjp1); // Not used : Coefficient is nulll
      //ydeltajp1 = 2.0/(Dyjp1*(Dyj+Dyjp1)); // Not used : Coefficient is nulll

      actualpointy = pointsy[j]; // =0
      actualpointx = pointsx[i];

      order_0 = -r;
      convection_x = (r - divid) * actualpointx; // It will be in the coefficient
```

```
  convection_y = alpha * (beta - actualpointy);
  diffusion_x = actualpointx * actualpointx * actualpointy / 2.0; // = 0
  diffusion_y = gamma * gamma * actualpointy / 2.0; // =0
  mixted = rho * gamma * actualpointx * actualpointy; // =0

  // MatrixTotal
  {
    // x y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i,
                // convection_x * xbetai0 // No convection_x term. It is in th
                + convection_y * ygammaj0
                + order_0);
    // x-1 y coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i - 1, 0.0);
    //+ convection_x * xbetaim1); // No convection_x term. It is in the coeffi
    // x+1 y coefficient
    //pnl_mat_set(MatrixTotal, j*(M+1)+i, j*(M+1)+i+1, 0.0); // Does not exist
    //diffusion_x * xdeltaip1 // Diffusion term is modified. This term becomes
    //+ mixted * xbetaip1 * ybetaj0 // No mixed term
    //convection_x * xbetaip1); // No convection_x term. It is in the coeffici
    // x y-2 coefficient
    //pnl_mat_set(MatrixTotal, j*(M+1)+i, (j-2)*(M+1)+i, 0.0); // Does not exi
    // x y-1 coefficient
    //pnl_mat_set(MatrixTotal, j*(M+1)+i, (j-1)*(M+1)+i, 0.0); // Does not exi
    // x y+1 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 1) * (M + 1) + i,
                convection_y * ygammajp1);
    // x y+2 coefficient
    pnl_mat_set(MatrixTotal, j * (M + 1) + i, (j + 2) * (M + 1) + i,
                convection_y * ygammajp2);

    // No corner coefficients since there is no mixed term.
  }
}
// Last line of volatility (i.e. j=N) : Dirichlet condition
{
  j = N;
  for (i = 0; i <= M; i++)
    {
      // For all spot, there is a Dirichlet condition.
      // So the matrix is null on this area. The condition is in the second me
```

```
      pnl_mat_set(MatrixTotal, j * (M + 1) + i, j * (M + 1) + i, 0.0);
      // Other coefficient are null by default.
    }
}

// Compute the solution "Sortie" of
// Sortie = VectTwo - coeff * (MatrixTotal * Sortie + Boundary conditions)
// which is also the solution of
// (Id + coeff * MatrixTotal) Sortie = VectTwo - coeff * Boundary conditions

// We do TWO backward Euler scheme iteration, so we need to divide coeff by 2.
// Sortie = VectTwo - coeff/2.0 * (MatrixTotal * Sortie + Boundary conditions)
// which is also the solution of
// (Id + coeff/2.0 * MatrixTotal) Sortie = VectTwo - coeff/2.0 * Boundary cond
// Sortie2 = Sortie - coeff/2.0 * (MatrixTotal * Sortie2 + Boundary conditions
// which is also the solution of
// (Id + coeff/2.0 * MatrixTotal) Sortie2 = Sortie - coeff/2.0 * Boundary cond

coeff = coeff / 2.0;

// Creation of identity matrix.
Ones = pnl_vect_create_from_double((M + 1) * (N + 1), 1.0);
SecondMember = pnl_vect_create((M + 1) * (N + 1));
Working_Matrix = pnl_mat_create_diag(Ones); // Identity matrix.
// Multiplication by coeff.
pnl_mat_mult_double(MatrixTotal, coeff);
// Sum with the identity matrix. Result is in the first argument.
pnl_mat_plus_mat(Working_Matrix, MatrixTotal);

// Computation with LU computation
permutation = pnl_permutation_create((M + 1) * (N + 1));
pnl_mat_lu(Working_Matrix, permutation);

// Add the boundary conditions.
pnl_vect_clone(SecondMember, VectTwo);
modify_solution_with_neumann_boundary_condition(
  r, divid, time / 2.0, call_or_put,
  M, pointsx, N, pointsy, coeff, SecondMember, SecondMember);
// First implicit Euler iteration
pnl_mat_lu_syslin(Ones, Working_Matrix, permutation, SecondMember);
// Add the boundary conditions.
```

```
  modify_solution_with_neumann_boundary_condition(
    r, divid, time, call_or_put,
    M, pointsx, N, pointsy, coeff, Ones, Ones);
  // Second implicit Euler iteration
  pnl_mat_lu_syslin(Sortie, Working_Matrix, permutation, Ones);

  // Free memory
  pnl_vect_int_free(&permutation);
  pnl_mat_free(&MatrixTotal);
  pnl_mat_free(&Working_Matrix);
  pnl_vect_free(&SecondMember);
  pnl_vect_free(&Ones);
}

static void construction_splitting_matrix_neumann(double coeff, double deltat,
    double *lower_d, double *diagonal, double *upper_d,
    int size_Matrix, PnlTridiagMat *Matrix)
{
  PnlTridiagMat *Identity_Matrix;
  PnlTridiagMat *Working_Matrix;
  Identity_Matrix = pnl_tridiag_mat_create_from_two_double(size_Matrix, 1.0, 0.0
  Working_Matrix = pnl_tridiag_mat_create_from_ptr(size_Matrix, lower_d, diagona
// Multiplication by coeff*deltat.
  pnl_tridiag_mat_mult_double(Working_Matrix, coeff * deltat);
// Sum with the identity matrix. Result is in the first argument.
  pnl_tridiag_mat_plus_tridiag_mat(Working_Matrix, Identity_Matrix);
  pnl_tridiag_mat_clone(Matrix, Working_Matrix);
// Boundary conditions.
  if (coeff < 0) // Explicit part
    {
      // Diagonal
      pnl_tridiag_mat_set(Matrix, size_Matrix - 1, 0, 0.0);
      // Lower diagonal
      pnl_tridiag_mat_set(Matrix, size_Matrix - 1, -1, 0.0);
    }
  if (coeff > 0) // Implicit part
    {
      // Diagonal
      pnl_tridiag_mat_set(Matrix, size_Matrix - 1, 0, 1.0);
      // Lower diagonal
      pnl_tridiag_mat_set(Matrix, size_Matrix - 1, -1, -1.0);
```

```
    }

  pnl_tridiag_mat_free(&Identity_Matrix);
  pnl_tridiag_mat_free(&Working_Matrix);
}

static void construction_splitting_matrix_dirichlet(double coeff, double deltat,
    double *lower_d, double *diagonal, double *upper_d,
    int size_Matrix, PnlTridiagMat *Matrix)
{
  PnlTridiagMat *Identity_Matrix;
  PnlTridiagMat *Working_Matrix;

  Identity_Matrix = pnl_tridiag_mat_create_from_two_double(size_Matrix, 1.0, 0.0
  Working_Matrix = pnl_tridiag_mat_create_from_ptr(size_Matrix, lower_d, diagona
// Multiplication by coeff*deltat.
  pnl_tridiag_mat_mult_double(Working_Matrix, coeff * deltat);
// Sum with the identity matrix. Result is in the first argument.
  pnl_tridiag_mat_plus_tridiag_mat(Working_Matrix, Identity_Matrix);
  pnl_tridiag_mat_clone(Matrix, Working_Matrix);
// Boundary conditions.
  if (coeff < 0) // Explicit part
    {
      // Diagonal
      pnl_tridiag_mat_set(Matrix, size_Matrix - 1, 0, 0.0);
      // Lower diagonal
      pnl_tridiag_mat_set(Matrix, size_Matrix - 1, -1, 0.0);
    }
  if (coeff > 0) // Implicit part
    {
      // Diagonal
      pnl_tridiag_mat_set(Matrix, size_Matrix - 1, 0, 1.0);
      // Lower diagonal
      pnl_tridiag_mat_set(Matrix, size_Matrix - 1, -1, 0.0);
    }

  pnl_tridiag_mat_free(&Identity_Matrix);
  pnl_tridiag_mat_free(&Working_Matrix);
}

static void construction_splitting_matrix_diagonal_neumann(double coeff, double
```

```
    double *lower_d, double *diagonal, double *upper_d,
    int size_Matrix, PnlTridiagMat *Matrix)
{
  PnlTridiagMat *Identity_Matrix;
  PnlTridiagMat *Working_Matrix;
  Identity_Matrix = pnl_tridiag_mat_create_from_two_double(size_Matrix, 1.0, 0.0
  Working_Matrix = pnl_tridiag_mat_create_from_ptr(size_Matrix, lower_d, diagona
// Multiplication by coeff*deltat.
  pnl_tridiag_mat_mult_double(Working_Matrix, coeff * deltat);
// Sum with the identity matrix. Result is in the first argument.
  pnl_tridiag_mat_plus_tridiag_mat(Working_Matrix, Identity_Matrix);
  pnl_tridiag_mat_clone(Matrix, Working_Matrix);
// Boundary conditions.
// If size_Matrix==1 then we are in a corner of the domain.
  if (size_Matrix > 1)
    {
      if (coeff < 0) // Explicit part
        {
          // Diagonal
          pnl_tridiag_mat_set(Matrix, size_Matrix - 1, 0, 0.0);
          // Lower diagonal
          pnl_tridiag_mat_set(Matrix, size_Matrix - 1, -1, 0.0);
        }
      if (coeff > 0) // Implicit part
        {
          // Diagonal
          pnl_tridiag_mat_set(Matrix, size_Matrix - 1, 0, 1.0);
          // Lower diagonal
          // Be careful... This does not look like Neumann, because
          // the coefficient for Neumann conditions is in another diagonal !!!
          // So the Neumann condition is treated later in the program.
          pnl_tridiag_mat_set(Matrix, size_Matrix - 1, -1, 0.0);
        }
    }
  else // size_Matrix==1 i.e. Matrix is just one coefficient.
    {
      // Diagonal
      pnl_tridiag_mat_set(Matrix, size_Matrix - 1, 0, 1.0);
    }

  pnl_tridiag_mat_free(&Identity_Matrix);
```

```
  pnl_tridiag_mat_free(&Working_Matrix);
}

static void construction_splitting_matrix_diagonal_dirichlet(double coeff, doubl
    double *lower_d, double *diagonal, double *upper_d,
    int size_Matrix, PnlTridiagMat *Matrix)
{
  PnlTridiagMat *Identity_Matrix;
  PnlTridiagMat *Working_Matrix;
  Identity_Matrix = pnl_tridiag_mat_create_from_two_double(size_Matrix, 1.0, 0.0
  Working_Matrix = pnl_tridiag_mat_create_from_ptr(size_Matrix, lower_d, diagona
// Multiplication by coeff*deltat.
  pnl_tridiag_mat_mult_double(Working_Matrix, coeff * deltat);
// Sum with the identity matrix. Result is in the first argument.
  pnl_tridiag_mat_plus_tridiag_mat(Working_Matrix, Identity_Matrix);
  pnl_tridiag_mat_clone(Matrix, Working_Matrix);
// Boundary conditions.
// If size_Matrix==1 then we are in a corner of the domain.
  if (size_Matrix > 1)
    {
      if (coeff < 0) // Explicit part
        {
          // Diagonal
          pnl_tridiag_mat_set(Matrix, size_Matrix - 1, 0, 0.0);
          // Lower diagonal
          pnl_tridiag_mat_set(Matrix, size_Matrix - 1, -1, 0.0);
        }
      if (coeff > 0) // Implicit part
        {
          // Diagonal
          pnl_tridiag_mat_set(Matrix, size_Matrix - 1, 0, 1.0);
          // Lower diagonal
          pnl_tridiag_mat_set(Matrix, size_Matrix - 1, -1, 0.0);
        }
    }
  else // size_Matrix==1 i.e. Matrix is just one coefficient.
    {
      // Diagonal
      pnl_tridiag_mat_set(Matrix, size_Matrix - 1, 0, 1.0);
    }
```

```
  pnl_tridiag_mat_free(&Identity_Matrix);
  pnl_tridiag_mat_free(&Working_Matrix);
}

static void brennan_schwartz_tridiag_call(PnlTridiagMat *Matrix, int size_Matrix
    PnlVect *SecondMember, PnlVect *Maximum, PnlVect *Sortie)
{
  int i;
  PnlVect *VectTemp;
  PnlTridiagMat *U_Matrix;
  PnlTridiagMat *L_Matrix;

  U_Matrix = pnl_tridiag_mat_create_from_two_double(size_Matrix, 7.7, 0.0);
  L_Matrix = pnl_tridiag_mat_create_from_two_double(size_Matrix, 1.0, 0.0);
//pnl_tridiag_mat_clone(U_Matrix, Matrix);
//pnl_tridiag_mat_clone(L_Matrix, Matrix);
  VectTemp = pnl_vect_create(size_Matrix);

  pnl_tridiag_mat_set(U_Matrix, 1 - 1, 0, pnl_tridiag_mat_get(Matrix, 1 - 1, 0))
  pnl_vect_set(VectTemp, 1 - 1, pnl_vect_get(SecondMember, 1 - 1));

  for (i = 1; i < size_Matrix; i++)
    {
      pnl_tridiag_mat_set(L_Matrix, i, -1,
                          pnl_tridiag_mat_get(Matrix, i, -1) / pnl_tridiag_mat_g

      pnl_tridiag_mat_set(U_Matrix, i - 1, 1,
                          pnl_tridiag_mat_get(Matrix, i - 1, 1));

      pnl_tridiag_mat_set(U_Matrix, i, 0,
                          pnl_tridiag_mat_get(Matrix, i, 0) - pnl_tridiag_mat_ge
                          * pnl_tridiag_mat_get(U_Matrix, i - 1, 1));

      pnl_vect_set(VectTemp, i, pnl_vect_get(SecondMember, i) -
                   pnl_tridiag_mat_get(L_Matrix, i, -1) * pnl_vect_get(VectTemp,
    }

  pnl_vect_set(Sortie, size_Matrix - 1,
               MAX(
                 pnl_vect_get(VectTemp, size_Matrix - 1) / pnl_tridiag_mat_get(U
                 pnl_vect_get(Maximum, size_Matrix - 1))
```

```
            );

  for (i = size_Matrix - 2; i >= 0; i--)
    {
      pnl_vect_set(Sortie, i,
                   MAX(
                     (
                       pnl_vect_get(VectTemp, i) -
                       pnl_tridiag_mat_get(U_Matrix, i, 1) * pnl_vect_get(Sortie
                     ) / pnl_tridiag_mat_get(U_Matrix, i, 0),
                     pnl_vect_get(Maximum, i))
                  );
    }

  pnl_vect_free(&VectTemp);
  pnl_tridiag_mat_free(&L_Matrix);
  pnl_tridiag_mat_free(&U_Matrix);

}

static void brennan_schwartz_tridiag_put(PnlTridiagMat *MatrixOld, int size_Matr
    PnlVect *SecondMemberOld, PnlVect *MaximumOld, PnlVect *Sortie)
{
  int i;
  PnlVect *SecondMember;
  PnlVect *Maximum;
  PnlVect *VectFlip;
  PnlTridiagMat *MatrixFlip;
  PnlTridiagMat *Matrix;

  SecondMember = pnl_vect_create(size_Matrix);
  Maximum = pnl_vect_create(size_Matrix);

// First, we should flip all the argument of the function.
  for (i = 0; i < size_Matrix; i++)
    {
      pnl_vect_set(SecondMember, i, pnl_vect_get(SecondMemberOld, size_Matrix -
    }
  for (i = 0; i < size_Matrix; i++)
    {
      pnl_vect_set(Maximum, i, pnl_vect_get(MaximumOld, size_Matrix - i - 1));
```

```
    }

  Matrix = pnl_tridiag_mat_create(size_Matrix);
  MatrixFlip = pnl_tridiag_mat_create(size_Matrix);

  for (i = 0; i < size_Matrix; i++)
    {
      pnl_tridiag_mat_set(MatrixFlip, i, 0, pnl_tridiag_mat_get(MatrixOld, size_
    }
  for (i = 0; i < size_Matrix - 1; i++)
    {
      pnl_tridiag_mat_set(MatrixFlip, i, 1, pnl_tridiag_mat_get(MatrixOld, size_
    }
  for (i = 1; i < size_Matrix; i++)
    {
      pnl_tridiag_mat_set(MatrixFlip, i, -1, pnl_tridiag_mat_get(MatrixOld, size
    }
  pnl_tridiag_mat_clone(Matrix, MatrixFlip);

  brennan_schwartz_tridiag_call(Matrix, size_Matrix, SecondMember, Maximum, Sort

// Finally, we should flip the solution Sortie.
  VectFlip = pnl_vect_create(size_Matrix);
  for (i = 0; i < size_Matrix; i++)
    {
      pnl_vect_set(VectFlip, i, pnl_vect_get(Sortie, size_Matrix - i - 1));
    }
  for (i = 0; i < size_Matrix; i++)
    {
      pnl_vect_set(Sortie, i, pnl_vect_get(VectFlip, i));
    }

  pnl_vect_free(&SecondMember);
  pnl_vect_free(&Maximum);
  pnl_vect_free(&VectFlip);
  pnl_tridiag_mat_free(&MatrixFlip);
  pnl_tridiag_mat_free(&Matrix);

}

static int ComSplitEu(double x, double y,
```

```
                        double t, double r, double divid,
                        double alpha, double beta, double gamma, double rho, doubl
                        double X, double Y, int mt, int N, int L,
                        double kappa, double coeff_for_x_1,
                        double omega_global, int boundary_conditions_method,
                        int call_or_put,
                        double *ptprice, double *ptdelta)
{
  int TimeIndex, i, j, k, l, cumul_index, index_central, M;
  int IndexX, IndexY, IndexUnknown;
  int size_Unknowns, number_coeff_on_diagonal;
  double deltat;
  double *diagonal_Ax, *upper_d_Ax, *lower_d_Ax;
  double *diagonal_Ay, *upper_d_Ay, *lower_d_Ay;
  double *diagonal_Axy, *upper_d_Axy, *lower_d_Axy;
  double *Unknowns;
  double *pointsx;
  double *pointsy;
  double penultimate_coeff_on_boundary, central_penultimate_coeff_on_boundary;
  double price_dl, price_dr, price_ul, price_ur;
  double h = Y / ((double)N);

  PnlVect *Unknowns_vect;
  PnlVect *Unknowns_old;
  PnlVect *Unknowns_h_line;
  PnlVect *Unknowns_h_line_new;
  PnlVect *Unknowns_v_line;
  PnlVect *Unknowns_v_line_new;
  PnlVect *Unknowns_d_line;
  PnlVect *Unknowns_d_line_new;
  PnlTridiagMat *Mat_Bx;
  PnlTridiagMat *Mat_Cx;
  PnlTridiagMat *Mat_Bxy;
  PnlTridiagMat *Mat_Cxy;
  PnlTridiagMat *Mat_By;
  PnlTridiagMat *Mat_Cy;

  // Y Space Step // Uniform grid
  pointsy = (double *)malloc((N + 1) * sizeof(double));
  grid_generation_uniform(pointsy, Y, y, N);
```

```
// X Space Step with IT's nonuniform grid.
M = size_grid_generation_IT(X, E, mt, kappa, rho, gamma, h, h / coeff_for_x_1)
size_Unknowns = (M + 1) * (N + 1);
pointsx = (double *)malloc((M + 1) * sizeof(double));
grid_generation_IT(pointsx, X, E, mt, kappa, rho, gamma, h, h / coeff_for_x_1)

// Time Step
deltat = t / ((double)L);

// Memory Allocation
Unknowns = (double *)malloc(size_Unknowns * sizeof(double));
upper_d_Ax = (double *)malloc(size_Unknowns * sizeof(double));
diagonal_Ax = (double *)malloc(size_Unknowns * sizeof(double));
lower_d_Ax = (double *)malloc(size_Unknowns * sizeof(double));
upper_d_Ay = (double *)malloc(size_Unknowns * sizeof(double));
diagonal_Ay = (double *)malloc(size_Unknowns * sizeof(double));
lower_d_Ay = (double *)malloc(size_Unknowns * sizeof(double));
upper_d_Axy = (double *)malloc(size_Unknowns * sizeof(double));
diagonal_Axy = (double *)malloc(size_Unknowns * sizeof(double));
lower_d_Axy = (double *)malloc(size_Unknowns * sizeof(double));

for (j = 0; j < N + 1; j++)
  {
    for (i = 0; i < M + 1; i++)
      {
        if (call_or_put == 1) //Call
          {
            Unknowns[i + j * (M + 1)] = MAX(pointsx[i] - E, 0.0);
          }
        if (call_or_put == -1) //Put
          {
            Unknowns[i + j * (M + 1)] = MAX(E - pointsx[i], 0.0);
          }
      }
  }

Unknowns_vect = pnl_vect_create_from_ptr(size_Unknowns, Unknowns);
Unknowns_old = pnl_vect_create_from_ptr(size_Unknowns, Unknowns);

Mat_Bx = pnl_tridiag_mat_create(M);
Mat_Cx = pnl_tridiag_mat_create(M);
```

```
Mat_By = pnl_tridiag_mat_create(N);
Mat_Cy = pnl_tridiag_mat_create(N);
Mat_Bxy = pnl_tridiag_mat_create((int)(MIN(M, N)));
Mat_Cxy = pnl_tridiag_mat_create((int)(MIN(M, N)));


// Finite Difference Cycle.
for (TimeIndex = 0; TimeIndex < L; TimeIndex++)
  {
    if (TimeIndex < 0)
      {
        // We can do Rannacher iterations i.e. two iterations with deltat/2.0.
        do_rannacher_iteration(r, divid, deltat * TimeIndex, alpha, beta, gamm
                               M, pointsx, N, pointsy,
                               -deltat, Unknowns_vect, Unknowns_vect);
      }
    else
      {

        // Construction of the matrix Ax, Ay and Axy.
        // System solver.

        //  Resolution of N-1 problems among the x-direction.
        for (j = 1; j < N; j++)
          {
            // Extraction of a subvector of Unknowns.
            Unknowns_h_line = pnl_vect_create(M + 1);
            Unknowns_h_line_new = pnl_vect_create(M + 1);
            pnl_vect_extract_subvect(Unknowns_h_line, Unknowns_vect, j * (M +
            pnl_vect_clone(Unknowns_h_line_new, Unknowns_h_line);
            // Construction of the submatrix.
            // First coefficient is 0 in order to respect boundary condition a
            // For Neumann boundary condition at x=X, we have two choices :
            // boundary condition with first order scheme
            // u^l+1_M = u^l+1_M-1
            // or boundary condition with second order scheme
            //  hl/hr u^l+1_M - (hl/hr-hr/hl) u^l+1_M-1 - hr/hl u^l+1_M-2 = 0
            // i.e. u^l+1_M = (1-hr^2/hl^2) u^l+1_M-1 + hr^2/hl^2 u^l+1_M-2
            //actualpointy = j*h;
            construction_Ax_coefficients(
              r, divid, alpha, beta, gamma, rho,
```

```
  X, M, pointsx, 1,
  Y, N, pointsy, j,
  omega_global,
  lower_d_Ax, diagonal_Ax,  upper_d_Ax);
// Splitting method.
if (boundary_conditions_method == 0)
  {
    // Homogeneous Neumann.
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Ma
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx,
    // Modification of the last coefficient in order to take into
    // Here, we do nothing.
  }
if (boundary_conditions_method == 1)
  {
    // Constant Neumann (= 1.0)
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Ma
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx,
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_h_line_new, M, pointsx[M] - pointsx[M -
  }
if (boundary_conditions_method == 2)
  {
    // Constant Neumann (= 1.0)
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Ma
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx,
```

```
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_h_line_new, M, pointsx[M] - pointsx[M -
  }
if (boundary_conditions_method == 3)
  {
    // Constant Neumann (= 1.0)
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Ma
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx,
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_h_line_new, M, pointsx[M] - pointsx[M -
  }
if (boundary_conditions_method == 4)
  {
    // Dirichlet = max(Spot - E*exp(-rt))
    construction_splitting_matrix_dirichlet(
      0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat
    construction_splitting_matrix_dirichlet(
      -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Ma
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx,
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_h_line_new, M, MAX(pointsx[M] - E * exp(
  }
if (boundary_conditions_method == 5)
  {
    construction_splitting_matrix_dirichlet(
      1.0, 1.0, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_Bx
    // Dirichlet = BS
    construction_splitting_matrix_dirichlet(
      0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat
    construction_splitting_matrix_dirichlet(
      -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Ma
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx,
```

```
        // Modification of the last coefficient in order to take into
        pnl_vect_set(Unknowns_h_line_new, M,
                     pnl_bs_call(pointsx[M], E, deltat * TimeIndex, r,
      }
    // Implicit part.
    pnl_tridiag_mat_syslin_inplace(Mat_Bx, Unknowns_h_line_new);
    // Update of the unknowns.
    for (k = 0; k <= M; k++)
      {
        pnl_vect_set(Unknowns_vect, j * (M + 1) + k, pnl_vect_get(Unkn
      }
    //#########
    //Correction Neumann
    //#########
    //  pnl_vect_set(Unknowns_vect, j*(M+1) + M,
    //  pnl_vect_get(Unknowns_h_line_new,M-1)+pointsx[M]-pointsx[M-1])
    //#########
    //#########

    // Memory desallocation
    pnl_vect_free(&Unknowns_h_line_new);
    pnl_vect_free(&Unknowns_h_line);
  }

// The line where wolatility is null. Outflow boundary.
{
  // Extraction of a subvector of Unknowns.
  Unknowns_h_line = pnl_vect_create(M + 1);
  Unknowns_h_line_new = pnl_vect_create(M + 1);
  pnl_vect_extract_subvect(Unknowns_h_line, Unknowns_vect, 0, M + 1);
  pnl_vect_clone(Unknowns_h_line_new, Unknowns_h_line);
  // Construction of the submatrix.
  // First coefficient is 0 in order to respect boundary condition at
  // For Neumann boundary condition at x=X, we have two choices :
  // boundary condition with first order scheme
  // u^l+1_M = u^l+1_M-1
  // or boundary condition with second order scheme
  //  hl/hr u^l+1_M - (hl/hr-hr/hl) u^l+1_M-1 - hr/hl u^l+1_M-2 = 0
  // i.e. u^l+1_M = (1-hr^2/hl^2) u^l+1_M-1 + hr^2/hl^2 u^l+1_M-2
  //actualpointy = 0
  construction_Ax_coefficients(
```

```
  r, divid, alpha, beta, gamma, rho,
  X, M, pointsx, 1,
  Y, N, pointsy, 0,
  omega_global,
  lower_d_Ax, diagonal_Ax,  upper_d_Ax);
// Splitting method.
if (boundary_conditions_method == 0)
  {
    // Homogeneous Neumann.
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_B
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx, U
    // Modification of the last coefficient in order to take into ac
    // Here, we do nothing.
  }
if (boundary_conditions_method == 1)
  {
    // Constant Neumann (= 1.0)
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_B
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx, U
    // Modification of the last coefficient in order to take into ac
    pnl_vect_set(Unknowns_h_line_new, M, pointsx[M] - pointsx[M - 1]
  }
if (boundary_conditions_method == 2)
  {
    // Constant Neumann (= 1.0)
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_B
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_
    // Resolution of the subproblem.
    // Explicit part.
```

```
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx, U
    // Modification of the last coefficient in order to take into ac
    pnl_vect_set(Unknowns_h_line_new, M, pointsx[M] - pointsx[M - 1]
  }
if (boundary_conditions_method == 3)
  {
    // Constant Neumann (= 1.0)
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_B
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx, U
    // Modification of the last coefficient in order to take into ac
    pnl_vect_set(Unknowns_h_line_new, M, pointsx[M] - pointsx[M - 1]
  }
if (boundary_conditions_method == 4)
  {
    // Dirichlet = max(Spot - E*exp(-rt))
    construction_splitting_matrix_dirichlet(
      0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_B
    construction_splitting_matrix_dirichlet(
      -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx, U
    // Modification of the last coefficient in order to take into ac
    pnl_vect_set(Unknowns_h_line_new, M, MAX(pointsx[M] - E * exp(-r
  }
if (boundary_conditions_method == 5)
  {
    construction_splitting_matrix_dirichlet(
      1.0, 1.0, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_Bx);
    // Dirichlet = BS
    construction_splitting_matrix_dirichlet(
      0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_B
    construction_splitting_matrix_dirichlet(
      -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_
    // Resolution of the subproblem.
    // Explicit part.
```

```
      pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx, U
      // Modification of the last coefficient in order to take into ac
      pnl_vect_set(Unknowns_h_line_new, M,
                   pnl_bs_call(pointsx[M], E, deltat * TimeIndex, r, d
    }
  // Implicit part.
  pnl_tridiag_mat_syslin_inplace(Mat_Bx, Unknowns_h_line_new);
  // Update of the unknowns.
  for (k = 0; k <= M; k++)
    {
      pnl_vect_set(Unknowns_vect, k, pnl_vect_get(Unknowns_h_line_new,
    }
  //#########
  //Correction Neumann
  //#########
  //  pnl_vect_set(Unknowns_vect, M,
  //  pnl_vect_get(Unknowns_h_line_new,M-1)+pointsx[M]-pointsx[M-1]);
  //#########
  //#########

  // Memory desallocation
  pnl_vect_free(&Unknowns_h_line_new);
  pnl_vect_free(&Unknowns_h_line);
}

// The line where volatility is maximal. t'Hout & Foulon says Dirichle
// I&T wants Neumann in volatility.
{
  if (boundary_conditions_method == 0)
    {
      // Neumann outflow
      for (i = 0; i <= M; i++)
        {
          if (call_or_put == 1) //Call
            {
              pnl_vect_set(Unknowns_vect, N * (M + 1) + i, pnl_vect_ge
            }
          if (call_or_put == -1) //Put
            {
              pnl_vect_set(Unknowns_vect, N * (M + 1) + i, pnl_vect_ge
            }
```

```
      }
  }
if (boundary_conditions_method == 1)
  {
    // We have Dirichlet condition.
    for (i = 0; i <= M; i++)
      {
        if (call_or_put == 1)
          {
            pnl_vect_set(Unknowns_vect, N * (M + 1) + i, pointsx[i])
            // Or maybe Condition should be S-E and not just S ???
          }
        if (call_or_put == -1)
          {
            // Not the good condition for the put option !!!!
            pnl_vect_set(Unknowns_vect, N * (M + 1) + i, -pointsx[i]
          }
      }
  }
if (boundary_conditions_method == 2)
  {
    // We have Dirichlet condition.
    for (i = 0; i <= M; i++)
      {
        if (call_or_put == 1)
          {
            pnl_vect_set(Unknowns_vect, N * (M + 1) + i, pointsx[i])
          }
        if (call_or_put == -1)
          {
            // Not the good condition for the put option !!!!
            pnl_vect_set(Unknowns_vect, N * (M + 1) + i, -pointsx[i]
          }
      }
  }
if (boundary_conditions_method == 3)
  {
    // Neumann outflow homogeneous
    for (i = 0; i <= M; i++)
      {
        if (call_or_put == 1) //Call
```

```
            {
              pnl_vect_set(Unknowns_vect, N * (M + 1) + i, pnl_vect_ge
            }
          if (call_or_put == -1) //Put
            {
              pnl_vect_set(Unknowns_vect, N * (M + 1) + i, pnl_vect_ge
            }
        }
    }
  if (boundary_conditions_method == 4)
    {
      // Neumann outflow homogeneous
      for (i = 0; i <= M; i++)
        {
          if (call_or_put == 1) //Call
            {
              pnl_vect_set(Unknowns_vect, N * (M + 1) + i, pnl_vect_ge
            }
          if (call_or_put == -1) //Put
            {
              pnl_vect_set(Unknowns_vect, N * (M + 1) + i, pnl_vect_ge
            }
        }
    }
  if (boundary_conditions_method == 5)
    {
      // Neumann outflow homogeneous
      for (i = 0; i <= M; i++)
        {
          if (call_or_put == 1) //Call
            {
              pnl_vect_set(Unknowns_vect, N * (M + 1) + i, pnl_vect_ge
            }
          if (call_or_put == -1) //Put
            {
              pnl_vect_set(Unknowns_vect, N * (M + 1) + i, pnl_vect_ge
            }
        }
    }

}
```

```
// Resolution of various sized problems among the diagonal-direction
// and in two distinct directions.
reorder_unknowns_x_to_xy(Unknowns_vect, M, N, Unknowns_old);
pnl_vect_clone(Unknowns_vect, Unknowns_old);
pnl_tridiag_mat_clone(Mat_Bxy, Mat_Bx);
pnl_tridiag_mat_clone(Mat_Cxy, Mat_Cx);
// First, we should find the central index.
index_central = 0;
for (k = 0; k < M; k++)
  {
    index_central = index_central + MIN(MIN(k + 1, MIN(M, N) + 1), M +
  }
cumul_index = index_central;
// Then we solve the first diagonal (i.e. the principal diagonal).
{
  number_coeff_on_diagonal = (int) MIN(MIN(M + 1, MIN(M, N) + 1), N +
  // Extraction of a subvector of Unknowns.
  Unknowns_d_line = pnl_vect_create(number_coeff_on_diagonal);
  Unknowns_d_line_new = pnl_vect_create(number_coeff_on_diagonal);
  pnl_vect_extract_subvect(Unknowns_d_line, Unknowns_vect, cumul_index
  pnl_vect_clone(Unknowns_d_line_new, Unknowns_d_line);
  // Construction of the submatrix.
  // First coefficient is 0 in order to respect boundary condition at
  // Last coefficient is 0 for Neumann boundary condition.
  // If M>N then the first point y is 0
  // else the first point y is the (N-M) point in the y grid.
  // If N>M then the first point x is pointsx[0]
  // else the first point x is the (M-N) point in the x grid.
  construction_Axy_coefficients(r, divid, alpha, beta, gamma, rho,
                                X, M, pointsx, (int)MAX(M - N, 0),
                                Y, N, pointsy, (int)MAX(N - M, 0),
                                number_coeff_on_diagonal,
                                omega_global,
                                lower_d_Axy, diagonal_Axy, upper_d_Axy

  // Splitting method.
  if (boundary_conditions_method == 0)
    {
      // Homogeneous Neumann.
      // Be careful. For this diagonal (and only this diagonal), we us
      // method than for x-direction (or y-direction) problem.
```

```
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_co
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_c
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy,
    // Modification of the last coefficient in order to take into ac
    // Here, we do nothing.
  }
if (boundary_conditions_method == 1)
  {
    // Dirichlet (= pointsx)
    // Be careful. For this diagonal (and only this diagonal), we us
    // method than for x-direction (or y-direction) problem.
    construction_splitting_matrix_dirichlet(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_co
    construction_splitting_matrix_dirichlet(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_c
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy,
    // Modification of the last coefficient in order to take into ac
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1,
  }
if (boundary_conditions_method == 2)
  {
    // Dirichlet (= pointsx)
    // Be careful. For this diagonal (and only this diagonal), we us
    // method than for x-direction (or y-direction) problem.
    construction_splitting_matrix_dirichlet(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_co
    construction_splitting_matrix_dirichlet(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_c
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy,
    // Modification of the last coefficient in order to take into ac
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1,
  }
if (boundary_conditions_method == 3)
```

```
  {
    // Constant Neumann (= 1.0)
    // Be careful. For this diagonal (and only this diagonal), we us
    // method than for x-direction (or y-direction) problem.
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_co
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_c
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy,
    // Modification of the last coefficient in order to take into ac
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1,
  }
if (boundary_conditions_method == 4)
  {
    construction_splitting_matrix_dirichlet(
      1.0, 1.0, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_coeff
    // Dirichlet = max(Spot - E exp(-rt)
    // Be careful. For this diagonal (and only this diagonal), we us
    // method than for x-direction (or y-direction) problem.
    construction_splitting_matrix_dirichlet(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_co
    construction_splitting_matrix_dirichlet(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_c
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy,
    // Modification of the last coefficient in order to take into ac
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1,
                 MAX(pointsx[M] - E * exp(-r * deltat * TimeIndex),
  }
if (boundary_conditions_method == 5)
  {
    construction_splitting_matrix_dirichlet(
      1.0, 1.0, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_coeff
    // Dirichlet = BS
    // Be careful. For this diagonal (and only this diagonal), we us
    // method than for x-direction (or y-direction) problem.
    construction_splitting_matrix_dirichlet(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_co
```

```
      construction_splitting_matrix_dirichlet(
        -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_c
      // Resolution of the subproblem.
      // Explicit part.
      pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy,
      // Modification of the last coefficient in order to take into ac
      pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1,
                   pnl_bs_call(pointsx[M], E, deltat * TimeIndex, r, d
    }

  // Implicit part.
  pnl_tridiag_mat_syslin_inplace(Mat_Bxy, Unknowns_d_line_new);
  // Update of the unknowns.
  for (l = 0; l < number_coeff_on_diagonal; l++)
    {
      pnl_vect_set(Unknowns_vect, cumul_index + l , pnl_vect_get(Unkno
    }

  cumul_index = cumul_index + number_coeff_on_diagonal;
}

// Then we solve the lower-right diagonals. Decreasing in vol. Fixed i
penultimate_coeff_on_boundary = pnl_vect_get(Unknowns_d_line_new, numb
central_penultimate_coeff_on_boundary = penultimate_coeff_on_boundary;
// Libération mémoire
pnl_vect_free(&Unknowns_d_line_new);
pnl_vect_free(&Unknowns_d_line);
pnl_tridiag_mat_clone(Mat_Bxy, Mat_Bx);
pnl_tridiag_mat_clone(Mat_Cxy, Mat_Cx);

for (k = M + 1; k < M + N; k++)
  {
    number_coeff_on_diagonal = (int) MIN(MIN(k + 1, MIN(M, N) + 1), M
    // The size of this subproblem is MIN(k+1,MIN(M,N)+1) if k <= MAX(
    // If k > MAX(M,N), the size of this subproblem is M+N+1-k.
    // So the size in general case is MIN(MIN(k+1,MIN(M,N)+1),M+N+1-k)
    // Extraction of a subvector of Unknowns.
    Unknowns_d_line = pnl_vect_create(number_coeff_on_diagonal);
    Unknowns_d_line_new = pnl_vect_create(number_coeff_on_diagonal);
    pnl_vect_extract_subvect(Unknowns_d_line, Unknowns_vect, cumul_ind
    pnl_vect_clone(Unknowns_d_line_new, Unknowns_d_line);
```

```
// Construction of the submatrix.
// First coefficient is 0 in order to respect boundary condition a
// Last coefficient is 0... But why ?
// The boundary condition is not explicit in the I&T's article.
// So we use the first diagonal in order to specify the Neumann bo
// If k>N then the first point y is 0
// else the first point y is the (N-k) point in the y grid.
// If N>k then the first point x is pointsx[0]
// else the first point x is the (k-N) point in the x grid.
construction_Axy_coefficients(r, divid, alpha, beta, gamma, rho,
                              X, M, pointsx, (int)MAX(k - N, 0),
                              Y, N, pointsy, (int)MAX(N - k, 0),
                              number_coeff_on_diagonal,
                              omega_global,
                              lower_d_Axy, diagonal_Axy, upper_d_A
// Resize the matrix...
pnl_tridiag_mat_resize(Mat_Bxy, number_coeff_on_diagonal);
pnl_tridiag_mat_resize(Mat_Cxy, number_coeff_on_diagonal);
// Splitting method.
if (boundary_conditions_method == 0)
  {
    // Homogeneous Neumann.
    construction_splitting_matrix_diagonal_neumann(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_neumann(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
  }
if (boundary_conditions_method == 1)
  {
    // Constant Neumann (= 1.0)
    construction_splitting_matrix_diagonal_neumann(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_neumann(
```

```
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
                 penultimate_coeff_on_boundary + pointsx[M] - poin
  }
if (boundary_conditions_method == 2)
  {
    // Constant Neumann (= 1.0)
    construction_splitting_matrix_diagonal_neumann(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_neumann(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
                 penultimate_coeff_on_boundary + pointsx[M] - poin
  }
if (boundary_conditions_method == 3)
  {
    // Constant Neumann (= 1.0)
    construction_splitting_matrix_diagonal_neumann(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_neumann(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
                 penultimate_coeff_on_boundary + pointsx[M] - poin
  }
```

```
if (boundary_conditions_method == 4)
  {
    // Dirichlet = max(Spot - E exp(-rt)
    construction_splitting_matrix_diagonal_dirichlet(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_dirichlet(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
                 MAX(pointsx[M] - E * exp(-r * deltat * TimeIndex)
  }
if (boundary_conditions_method == 5)
  {
    construction_splitting_matrix_dirichlet(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_
    construction_splitting_matrix_dirichlet(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number
    // Dirichlet = BS
    construction_splitting_matrix_diagonal_dirichlet(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_dirichlet(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    // What is the var/vol at the last point
    // The diagonal begins at point (int)MAX(N-k,0) so it finishes
    // (int)MAX(N-k,0)+number_coeff_on_diagonal-1
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
                 pnl_bs_call(pointsx[M], E, deltat * TimeIndex, r,
                               (int)MAX(N - k, 0) + number_coeff_o
                             ])));
  }
```

```
    // Implicit part.
    pnl_tridiag_mat_syslin_inplace(Mat_Bxy, Unknowns_d_line_new);
    // Update of the unknowns.
    for (l = 0; l < number_coeff_on_diagonal; l++)
      {
        pnl_vect_set(Unknowns_vect, cumul_index + l , pnl_vect_get(Unk
      }
    cumul_index = cumul_index + number_coeff_on_diagonal;
    if (number_coeff_on_diagonal > 1)
      {
        // if number_coeff_on_diagonal==1 so this is the end of the lo
        // and the penultimate coefficient is useless (and nonsense).
        // And in fact, this case will never occur.
        penultimate_coeff_on_boundary = pnl_vect_get(Unknowns_d_line_n
      }
    // Memory desallocation
    pnl_vect_free(&Unknowns_d_line_new);
    pnl_vect_free(&Unknowns_d_line);
  }
// The unknowns in the lower-right corner is not modified...
// pnl_vect_set(Unknowns_vect, size_Unknowns-1, pnl_vect_get(Unknowns_
// Then we solve the upper-left diagonals. Decreasing in spot. Fixed i
penultimate_coeff_on_boundary = central_penultimate_coeff_on_boundary;
cumul_index = index_central;
pnl_tridiag_mat_clone(Mat_Bxy, Mat_Bx);
pnl_tridiag_mat_clone(Mat_Cxy, Mat_Cx);

for (k = M - 1; k > 0; k--)
  {
    number_coeff_on_diagonal = (int) MIN(MIN(k + 1, MIN(M, N) + 1), M
    // The size of this subproblem is MIN(k+1,MIN(M,N)+1) if k <= MAX(
    // If k > MAX(M,N), the size of this subproblem is M+N+1-k.
    // So the size in general case is MIN(MIN(k+1,MIN(M,N)+1),M+N+1-k)
    cumul_index = cumul_index - number_coeff_on_diagonal;
    // Extraction of a subvector of Unknowns.
    Unknowns_d_line = pnl_vect_create(number_coeff_on_diagonal);
    Unknowns_d_line_new = pnl_vect_create(number_coeff_on_diagonal);

    pnl_vect_extract_subvect(Unknowns_d_line, Unknowns_vect, cumul_ind
    pnl_vect_clone(Unknowns_d_line_new, Unknowns_d_line);
```

```
// Construction of the submatrix.
// First coefficient is 0 in order to respect boundary condition a
// Last coefficient is 0... But why ?
// The boundary condition is not explicit in the I&T's article.
// So we use the first diagonal in order to specify the Neumann bo
// If k>N then the first point y is 0
// else the first point y is the (N-k) point in the y grid.
// If N>k then the first point x is pointsx[0]
// else the first point x is the (k-N) point in the x grid.
construction_Axy_coefficients(r, divid, alpha, beta, gamma, rho,
                              X, M, pointsx, (int)MAX(k - N, 0),
                              Y, N, pointsy, (int)MAX(N - k, 0),
                              number_coeff_on_diagonal,
                              omega_global,
                              lower_d_Axy, diagonal_Axy, upper_d_A
// Resize the matrix...
pnl_tridiag_mat_resize(Mat_Bxy, number_coeff_on_diagonal);
pnl_tridiag_mat_resize(Mat_Cxy, number_coeff_on_diagonal);
// Splitting method.
if (boundary_conditions_method == 0)
  {
    // Homogeneous Neumann.
    construction_splitting_matrix_diagonal_neumann(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_neumann(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
  }
if (boundary_conditions_method == 1)
  {
    // Dirichlet (= pointsx)
    construction_splitting_matrix_diagonal_dirichlet(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_dirichlet(
```

```
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    // The pointsx index is max((k-N),0)+min(N+1,k+1)-1 = k.
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
  }
if (boundary_conditions_method == 2)
  {
    // Dirichlet (= pointsx)
    construction_splitting_matrix_diagonal_dirichlet(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_dirichlet(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    // The pointsx index is max((k-N),0)+min(N+1,k+1)-1 = k.
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
  }
if (boundary_conditions_method == 3)
  {
    // Homogeneous Neumann.
    construction_splitting_matrix_diagonal_neumann(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_neumann(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
  }
if (boundary_conditions_method == 4)
```

```
  {
    // Homogeneous Neumann.
    construction_splitting_matrix_diagonal_neumann(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_neumann(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
  }
if (boundary_conditions_method == 5)
  {
    construction_splitting_matrix_dirichlet(
      1.0, 1.0, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_coe
    // Homogeneous Neumann.
    construction_splitting_matrix_diagonal_neumann(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_neumann(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
  }

// Implicit part.
pnl_tridiag_mat_syslin_inplace(Mat_Bxy, Unknowns_d_line_new);
// Update of the unknowns.
for (l = 0; l < number_coeff_on_diagonal; l++)
  {
    pnl_vect_set(Unknowns_vect, cumul_index + l , pnl_vect_get(Unk

  }
if (number_coeff_on_diagonal > 1)
```

```
              {
                // if number_coeff_on_diagonal==1 so this is the end of the lo
                // and the penultimate coefficient is useless (and nonsense).
                // And in fact, this case will never occur.
                penultimate_coeff_on_boundary = pnl_vect_get(Unknowns_d_line_n
              }
            // Libération mémoire
            pnl_vect_free(&Unknowns_d_line_new);
            pnl_vect_free(&Unknowns_d_line);
          }
        // The unknowns in the upper-left corner is not modified...
        // pnl_vect_set(Unknowns_vect, 0, pnl_vect_get(Unknowns_vect, 0));

        reorder_unknowns_xy_to_y(Unknowns_vect, M, N, Unknowns_old);
        pnl_vect_clone(Unknowns_vect, Unknowns_old);

        //  Resolution of M-1 problems among the y-direction.
        for (i = 1; i < M; i++)
          {
            // Extraction of a subvector of Unknowns.
            Unknowns_v_line = pnl_vect_create(N + 1);
            Unknowns_v_line_new = pnl_vect_create(N + 1);
            pnl_vect_extract_subvect(Unknowns_v_line, Unknowns_vect, i * (N +
            pnl_vect_clone(Unknowns_v_line_new, Unknowns_v_line);
// Construction of the submatrix.
            // First coefficient is 0 in order to respect boundary condition a
            // For Neumann boundary condition at y=Y, we have two choices :
            // boundary condition with first order scheme
            // u^l+1_M = u^l+1_M-1
            // or boundary condition with second order scheme
            // u^l+1_M = u^l+1_M-2
            //actualpointx = pointsx[i];
            construction_Ay_coefficients(
              r, divid, alpha, beta, gamma, rho,
              X, M, pointsx, i,
              Y, N, pointsy, 1,
              omega_global,
              lower_d_Ay, diagonal_Ay,  upper_d_Ay);
            // Splitting method.
            if (boundary_conditions_method == 0)
              {
```

```
    // Homogeneous Neumann.
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Mat
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Ma
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_v_line_new, Mat_Cy,
    // Modification of the last coefficient in order to take into
    // Here we do nothing.
  }
if (boundary_conditions_method == 1)
  {
    // Dirichlet (= pointsx)
    construction_splitting_matrix_dirichlet(
      0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Mat
    construction_splitting_matrix_dirichlet(
      -0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Ma
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_v_line_new, Mat_Cy,
    // Modification of the last coefficient in order to take into
    // The pointsx index is i.
    pnl_vect_set(Unknowns_v_line_new, N, pointsx[i]);
  }
if (boundary_conditions_method == 2)
  {
    // Dirichlet (= pointsx)
    construction_splitting_matrix_dirichlet(
      0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Mat
    construction_splitting_matrix_dirichlet(
      -0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Ma
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_v_line_new, Mat_Cy,
    // Modification of the last coefficient in order to take into
    // The pointsx index is i.
    pnl_vect_set(Unknowns_v_line_new, N, pointsx[i]);
  }
if (boundary_conditions_method == 3)
  {
```

```
    // Homogeneous Neumann.
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Mat
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Ma
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_v_line_new, Mat_Cy,
    // Modification of the last coefficient in order to take into
    // Here we do nothing.
  }
if (boundary_conditions_method == 4)
  {
    // Homogeneous Neumann.
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Mat
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Ma
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_v_line_new, Mat_Cy,
    // Modification of the last coefficient in order to take into
    // Here we do nothing.
  }
if (boundary_conditions_method == 5)
  {
    construction_splitting_matrix_neumann(
      1.0, 1.0, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Mat_By
    // Homogeneous Neumann.
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Mat
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Ma
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_v_line_new, Mat_Cy,
    // Modification of the last coefficient in order to take into
    // Here we do nothing.
  }

// Implicit part.
```

```
    pnl_tridiag_mat_syslin_inplace(Mat_By, Unknowns_v_line_new);

    // Update of the unknowns.
    for (k = 0; k <= N; k++)
      {
        pnl_vect_set(Unknowns_vect, i * (N + 1) + k, pnl_vect_get(Unkn
      }
    // Libération mémoire
    pnl_vect_free(&Unknowns_v_line_new);
    pnl_vect_free(&Unknowns_v_line);
  }

// The line where spot is 0. Dirichlet homogeneous
{
  if (boundary_conditions_method == 0)
    {
      // Dirichlet homogeneous
      for (k = 0; k <= N; k++)
        {
          if (call_or_put == 1) //Call
            {
              pnl_vect_set(Unknowns_vect, k, 0.0);
            }
          if (call_or_put == -1) //Put
            {
              pnl_vect_set(Unknowns_vect, k, E * exp(-r * TimeIndex *
            }
        }
    }
  if (boundary_conditions_method == 1)
    {
      // Dirichlet homogeneous
      for (k = 0; k <= N; k++)
        {
          if (call_or_put == 1) //Call
            {
              pnl_vect_set(Unknowns_vect, k, 0.0);
            }
          if (call_or_put == -1) //Put
            {
              pnl_vect_set(Unknowns_vect, k, E * exp(-r * TimeIndex *
```

```
          }
      }
  }
if (boundary_conditions_method == 2)
  {
    // Dirichlet homogeneous
    for (k = 0; k <= N; k++)
      {
        if (call_or_put == 1) //Call
          {
            pnl_vect_set(Unknowns_vect, k, 0.0);
          }
        if (call_or_put == -1) //Put
          {
            pnl_vect_set(Unknowns_vect, k, E * exp(-r * TimeIndex *
          }
      }
  }
if (boundary_conditions_method == 3)
  {
    // Dirichlet homogeneous
    for (k = 0; k <= N; k++)
      {
        if (call_or_put == 1) //Call
          {
            pnl_vect_set(Unknowns_vect, k, 0.0);
          }
        if (call_or_put == -1) //Put
          {
            pnl_vect_set(Unknowns_vect, k, E * exp(-r * TimeIndex *
          }
      }
  }
if (boundary_conditions_method == 4)
  {
    // Dirichlet homogeneous
    for (k = 0; k <= N; k++)
      {
        if (call_or_put == 1) //Call
          {
            pnl_vect_set(Unknowns_vect, k, 0.0);
```

```
            }
          if (call_or_put == -1) //Put
            {
              pnl_vect_set(Unknowns_vect, k, E * exp(-r * TimeIndex *
            }
        }
    }
  if (boundary_conditions_method == 5)
    {
      // Dirichlet homogeneous
      for (k = 0; k <= N; k++)
        {
          if (call_or_put == 1) //Call
            {
              pnl_vect_set(Unknowns_vect, k, 0.0);
            }
          if (call_or_put == -1) //Put
            {
              pnl_vect_set(Unknowns_vect, k, E * exp(-r * TimeIndex *
            }
        }
    }
}

// The line where spot is maximal. t'Hout & Foulon recommande Neumann
// I&T recommande Neumann =0. Est-ce que la frontiere est outflow ou i
{
  if (boundary_conditions_method == 0)
    {
      // Neumann outflow homogeneous
      for (k = 0; k <= N; k++)
        {
          if (call_or_put == 1) //Call
            {
              pnl_vect_set(Unknowns_vect, M * (N + 1) + k, pnl_vect_ge
            }
          if (call_or_put == -1) //Put
            {
              pnl_vect_set(Unknowns_vect, M * (N + 1) + k, pnl_vect_ge
            }
        }
```

```
  }
if (boundary_conditions_method == 1)
  {
    // Neumann outflow = 1
    for (k = 0; k <= N; k++)
      {
        if (call_or_put == 1) //Call
          {
            pnl_vect_set(Unknowns_vect, M * (N + 1) + k, pnl_vect_ge
                         + pointsx[M] - pointsx[M - 1]);
          }
        if (call_or_put == -1) //Put (NOT THE RIGHT CONDITION HERE)
          {
            pnl_vect_set(Unknowns_vect, M * (N + 1) + k, pnl_vect_ge
                         + pointsx[M] - pointsx[M - 1]);
          }
      }
  }
if (boundary_conditions_method == 2)
  {
    // Neumann outflow = 1
    for (k = 0; k <= N; k++)
      {
        if (call_or_put == 1) //Call
          {
            pnl_vect_set(Unknowns_vect, M * (N + 1) + k, pnl_vect_ge
                         + pointsx[M] - pointsx[M - 1]);
          }
        if (call_or_put == -1) //Put (NOT THE RIGHT CONDITION HERE)
          {
            pnl_vect_set(Unknowns_vect, M * (N + 1) + k, pnl_vect_ge
                         + pointsx[M] - pointsx[M - 1]);
          }
      }
  }
if (boundary_conditions_method == 3)
  {
    // Neumann outflow = 1
    for (k = 0; k <= N; k++)
      {
        if (call_or_put == 1) //Call
```

```
          {
            pnl_vect_set(Unknowns_vect, M * (N + 1) + k, pnl_vect_ge
                         + pointsx[M] - pointsx[M - 1]);
          }
        if (call_or_put == -1) //Put (NOT THE RIGHT CONDITION HERE)
          {
            pnl_vect_set(Unknowns_vect, M * (N + 1) + k, pnl_vect_ge
                         + pointsx[M] - pointsx[M - 1]);
          }
      }
  }
if (boundary_conditions_method == 4)
  {
    // Dirichlet = max(Spot - E exp(-rt)
    for (k = 0; k <= N; k++)
      {
        if (call_or_put == 1) //Call
          {
            pnl_vect_set(Unknowns_vect, M * (N + 1) + k, MAX(pointsx
          }
        if (call_or_put == -1) //Put
          {
            pnl_vect_set(Unknowns_vect, M * (N + 1) + k, MAX(E * exp
          }
      }
  }
if (boundary_conditions_method == 5)
  {
    // Dirichlet = BS
    for (k = 0; k <= N; k++)
      {
        if (call_or_put == 1) //Call
          {
            pnl_vect_set(Unknowns_vect, M * (N + 1) + k,
                         pnl_bs_call(pointsx[M], E, deltat * TimeInd
          }
        if (call_or_put == -1) //Put
          {
            pnl_vect_set(Unknowns_vect, M * (N + 1) + k,
                         pnl_bs_put(pointsx[M], E, deltat * TimeInde
          }
```

```
                }
            }
        }
        reorder_unknowns_y_to_x(Unknowns_vect, M, N , Unknowns_old);
        pnl_vect_clone(Unknowns_vect, Unknowns_old);
      }
  }

// Index in the domain for the price.
// Find the index in pointsx corresponding to the price x of the asset.
IndexX = lower_index(pointsx, M + 1, x);
// Find the index in [0,Y] corresponding to the variance y of the asset.
//IndexY = (int)(y/h);
IndexY = lower_index(pointsy, N + 1, y);
// Compute the index in the vector of unknowns
IndexUnknown = IndexX + IndexY * (M + 1);

// Price.
// We do an interpolation.
price_ul = pnl_vect_get(Unknowns_vect, IndexUnknown + M + 1);
price_ur = pnl_vect_get(Unknowns_vect, IndexUnknown + 1 + M + 1);
price_dl = pnl_vect_get(Unknowns_vect, IndexUnknown);
price_dr = pnl_vect_get(Unknowns_vect, IndexUnknown + 1);

*ptprice =
  (
    (pointsy[IndexY + 1] - y) *
    (
      (x - pointsx[IndexX]) * price_dr
      +
      (pointsx[IndexX + 1] - x) * price_dl
    )
    / (pointsx[IndexX + 1] - pointsx[IndexX])
    +
    (y - pointsy[IndexY]) *
    (
      (x - pointsx[IndexX]) * price_ur
      +
      (pointsx[IndexX + 1] - x) * price_ul
    )
    / (pointsx[IndexX + 1] - pointsx[IndexX])
```

```
  )
  / (pointsy[IndexY + 1] - pointsy[IndexY]);

pnl_vect_get(Unknowns_vect, IndexUnknown);

// Delta.
if (x - pointsx[IndexX] > 0)
  {
    *ptdelta =
      (
        ((((pointsy[IndexY + 1] - y) * price_dr + (y - pointsy[IndexY]) * pric
          / (pointsy[IndexY + 1] - pointsy[IndexY])) - *ptprice)
        /
        (pointsx[IndexX + 1] - x)
        +
        (*ptprice - (((pointsy[IndexY + 1] - y) * price_dl + (y - pointsy[Inde
                     / (pointsy[IndexY + 1] - pointsy[IndexY])))
        /
        (x - pointsx[IndexX])
      ) / 2.0;
  }
else
  {
    *ptdelta =
      ((((pointsy[IndexY + 1] - y) * price_dr + (y - pointsy[IndexY]) * price_
        (pointsy[IndexY + 1] - pointsy[IndexY])) - *ptprice)
      /
      (pointsx[IndexX + 1] - x);

  }

// Memory desallocation.
free(pointsx);
free(pointsy);
free(Unknowns);
free(upper_d_Ax);
free(diagonal_Ax);
free(lower_d_Ax);
free(upper_d_Ay);
free(diagonal_Ay);
free(lower_d_Ay);
```

```
  free(upper_d_Axy);
  free(diagonal_Axy);
  free(lower_d_Axy);

  pnl_vect_free(&Unknowns_vect);
  pnl_vect_free(&Unknowns_old);

  pnl_tridiag_mat_free(&Mat_Bx);
  pnl_tridiag_mat_free(&Mat_Cx);
  pnl_tridiag_mat_free(&Mat_Bxy);
  pnl_tridiag_mat_free(&Mat_Cxy);
  pnl_tridiag_mat_free(&Mat_By);
  pnl_tridiag_mat_free(&Mat_Cy);

  return 0.;
}


static int ComSplitAm
(double x, double y,
 double t, double r, double divid,
 double alpha, double beta, double gamma, double rho, double E,
 double X, double Y, int mt, int N, int L,
 double kappa, double coeff_for_x_1,
 double omega_global, int boundary_conditions_method,
 int call_or_put,
 double *ptprice, double *ptdelta)
{
  int TimeIndex, i, j, k, l, cumul_index, index_central, M;
  int IndexX, IndexY, IndexUnknown;
  int size_Unknowns, number_coeff_on_diagonal;
  double deltat;
  double *diagonal_Ax, *upper_d_Ax, *lower_d_Ax;
  double *diagonal_Ay, *upper_d_Ay, *lower_d_Ay;
  double *diagonal_Axy, *upper_d_Axy, *lower_d_Axy;
  double *Unknowns;
  double *pointsx;
  double *pointsy;
  double penultimate_coeff_on_boundary, central_penultimate_coeff_on_boundary;
  double price_dl, price_dr, price_ul, price_ur;
  double h = Y / ((double)N);
```

```
  PnlVect *Unknowns_vect;
  PnlVect *Unknowns_old;
  PnlVect *Unknowns_h_line;
  PnlVect *Unknowns_h_line_new;
  PnlVect *Unknowns_v_line;
  PnlVect *Unknowns_v_line_new;
  PnlVect *Unknowns_d_line;
  PnlVect *Unknowns_d_line_new;
//##
  PnlVect *Unknowns_max;
//##
  PnlTridiagMat *Mat_Bx;
  PnlTridiagMat *Mat_Cx;
  PnlTridiagMat *Mat_Bxy;
  PnlTridiagMat *Mat_Cxy;
  PnlTridiagMat *Mat_By;
  PnlTridiagMat *Mat_Cy;

  // Y Space Step // Uniform grid
  pointsy = (double *)malloc((N + 1) * sizeof(double));
  grid_generation_uniform(pointsy, Y, y, N);

  // X Space Step with IT's nonuniform grid.
  M = size_grid_generation_IT(X, E, mt, kappa, rho, gamma, h, h / coeff_for_x_1)
  size_Unknowns = (M + 1) * (N + 1);
  pointsx = (double *)malloc((M + 1) * sizeof(double));
  grid_generation_IT(pointsx, X, E, mt, kappa, rho, gamma, h, h / coeff_for_x_1)

  // Time Step
  deltat = t / ((double)L);

  // Memory Allocation
  Unknowns = (double *)malloc(size_Unknowns * sizeof(double));
  upper_d_Ax = (double *)malloc(size_Unknowns * sizeof(double));
  diagonal_Ax = (double *)malloc(size_Unknowns * sizeof(double));
  lower_d_Ax = (double *)malloc(size_Unknowns * sizeof(double));
  upper_d_Ay = (double *)malloc(size_Unknowns * sizeof(double));
  diagonal_Ay = (double *)malloc(size_Unknowns * sizeof(double));
  lower_d_Ay = (double *)malloc(size_Unknowns * sizeof(double));
  upper_d_Axy = (double *)malloc(size_Unknowns * sizeof(double));
```

```
diagonal_Axy = (double *)malloc(size_Unknowns * sizeof(double));
lower_d_Axy = (double *)malloc(size_Unknowns * sizeof(double));

for (j = 0; j < N + 1; j++)
  {
    for (i = 0; i < M + 1; i++)
      {
        if (call_or_put == 1) //Call
          {
            Unknowns[i + j * (M + 1)] = MAX(pointsx[i] - E, 0.0);
          }
        if (call_or_put == -1) //Put
          {
            Unknowns[i + j * (M + 1)] = MAX(E - pointsx[i], 0.0);
          }
      }
  }

Unknowns_vect = pnl_vect_create_from_ptr(size_Unknowns, Unknowns);
Unknowns_old = pnl_vect_create_from_ptr(size_Unknowns, Unknowns);
Unknowns_max = pnl_vect_create(1);

Mat_Bx = pnl_tridiag_mat_create(M);
Mat_Cx = pnl_tridiag_mat_create(M);
Mat_By = pnl_tridiag_mat_create(N);
Mat_Cy = pnl_tridiag_mat_create(N);
Mat_Bxy = pnl_tridiag_mat_create((int)(MIN(M, N)));
Mat_Cxy = pnl_tridiag_mat_create((int)(MIN(M, N)));


// Finite Difference Cycle.
for (TimeIndex = 0; TimeIndex < L; TimeIndex++)
  {
    if (TimeIndex < 0)
      {
        // We do Rannacher iterations i.e. two iterations with deltat/2.0.
        do_rannacher_iteration(r, divid, deltat * TimeIndex, alpha, beta, gamm
                               M, pointsx, N, pointsy,
                               -deltat, Unknowns_vect, Unknowns_vect);
        // American condition = Unknowns >= payoff
        for (j = 0; j < N + 1; j++)
```

```
            {
              for (i = 0; i < M + 1; i++)
                {
                  if (call_or_put == 1) //Call
                    {
                      pnl_vect_set(Unknowns_vect, i + j * (M + 1),
                                   MAX(pnl_vect_get(Unknowns_vect, i + j * (M +
                    }
                  if (call_or_put == -1) //Put
                    {
                      pnl_vect_set(Unknowns_vect, i + j * (M + 1),
                                   MAX(pnl_vect_get(Unknowns_vect, i + j * (M +
                    }
                }
            }
        }
      else
        {
          // American condition = Unknowns >= payoff
          for (j = 0; j < N + 1; j++)
            {
              for (i = 0; i < M + 1; i++)
                {
                  if (call_or_put == 1) //Call
                    {
                      if (pnl_vect_get(Unknowns_vect, i + j * (M + 1)) < MAX(poi
                        {
//   std::cout << "Call ici i=" << i << " et j=" << j << std::endl;
                        }
                      pnl_vect_set(Unknowns_vect, i + j * (M + 1),
                                   MAX(pnl_vect_get(Unknowns_vect, i + j * (M +
                    }
                  if (call_or_put == -1) //Put
                    {
                      if (pnl_vect_get(Unknowns_vect, i + j * (M + 1)) < MAX(E -
                        {
//   std::cout << "Put ici i=" << i << " et j=" << j << std::endl;
                        }
                      pnl_vect_set(Unknowns_vect, i + j * (M + 1),
                                   MAX(pnl_vect_get(Unknowns_vect, i + j * (M +
                    }
```

```
      }
  }

// Construction of the matrix Ax, Ay and Axy.
// System solver.

//  Resolution of N-1 problems among the x-direction.
for (j = 1; j < N; j++)
  {
    // Extraction of a subvector of Unknowns.
    Unknowns_h_line = pnl_vect_create(M + 1);
    Unknowns_h_line_new = pnl_vect_create(M + 1);
    pnl_vect_extract_subvect(Unknowns_h_line, Unknowns_vect, j * (M +
    pnl_vect_clone(Unknowns_h_line_new, Unknowns_h_line);
    // Construction of the submatrix.
    // First coefficient is 0 in order to respect boundary condition a
    // For Neumann boundary condition at x=X, we have two choices :
    // boundary condition with first order scheme
    // u^l+1_M = u^l+1_M-1
    // or boundary condition with second order scheme
    //  hl/hr u^l+1_M - (hl/hr-hr/hl) u^l+1_M-1 - hr/hl u^l+1_M-2 = 0
    // i.e. u^l+1_M = (1-hr^2/hl^2) u^l+1_M-1 + hr^2/hl^2 u^l+1_M-2
    //actualpointy = j*h;
    construction_Ax_coefficients(
      r, divid, alpha, beta, gamma, rho,
      X, M, pointsx, 1,
      Y, N, pointsy, j,
      omega_global,
      lower_d_Ax, diagonal_Ax,  upper_d_Ax);
    // Splitting method.
    if (boundary_conditions_method == 0)
      {
        // Homogeneous Neumann.
        construction_splitting_matrix_neumann(
          0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat
        construction_splitting_matrix_neumann(
          -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Ma
        // Resolution of the subproblem.
        // Explicit part.
        pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx,
        // Modification of the last coefficient in order to take into
```

```
    // Here, we do nothing.
  }
if (boundary_conditions_method == 1)
  {
    // Constant Neumann (= 1.0)
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Ma
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx,
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_h_line_new, M, pointsx[M] - pointsx[M -
  }
if (boundary_conditions_method == 2)
  {
    // Constant Neumann (= 1.0)
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Ma
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx,
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_h_line_new, M, pointsx[M] - pointsx[M -
  }
if (boundary_conditions_method == 3)
  {
    // Constant Neumann (= 1.0)
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Ma
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx,
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_h_line_new, M, pointsx[M] - pointsx[M -
  }
```

```
              if (boundary_conditions_method == 4)
                {
                  // Dirichlet = max(Spot - E*exp(-rt))
                  construction_splitting_matrix_dirichlet(
                    0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat
                  construction_splitting_matrix_dirichlet(
                    -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Ma
                  // Resolution of the subproblem.
                  // Explicit part.
                  pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx,
                  // Modification of the last coefficient in order to take into
                  pnl_vect_set(Unknowns_h_line_new, M, MAX(pointsx[M] - E * exp(
                }
              if (boundary_conditions_method == 5)
                {
                  construction_splitting_matrix_dirichlet(
                    1.0, 1.0, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_Bx
                  // Dirichlet = BS
                  construction_splitting_matrix_dirichlet(
                    0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat
                  construction_splitting_matrix_dirichlet(
                    -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Ma
                  // Resolution of the subproblem.
                  // Explicit part.
                  pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx,
                  // Modification of the last coefficient in order to take into
                  pnl_vect_set(Unknowns_h_line_new, M,
                               pnl_bs_call(pointsx[M], E, deltat * TimeIndex, r,
                }

//##

              // Implicit part.
// pnl_tridiag_mat_syslin_inplace(Mat_Bx, Unknowns_h_line_new);
              // The implicit resolution is replaced by Brennan-Schwartz.
              // Definition of the payoff for the american condition.
              pnl_vect_resize(Unknowns_max, M + 1);
              if (call_or_put == 1) //Call
                {
                  for (l = 0; l < M + 1; l++)
                    {
```

```
                    pnl_vect_set(Unknowns_max, l, MAX(pointsx[l] - E, 0.0));
                  }
                brennan_schwartz_tridiag_call(Mat_Bx, M + 1, Unknowns_h_line_n
              }
            if (call_or_put == -1) //Put
              {
                for (l = 0; l < M + 1; l++)
                  {
                    pnl_vect_set(Unknowns_max, l, MAX(E - pointsx[l], 0.0));
                  }
                brennan_schwartz_tridiag_put(Mat_Bx, M + 1, Unknowns_h_line_ne
              }

//##

            // Update of the unknowns.
            for (k = 0; k <= M; k++)
              {
                pnl_vect_set(Unknowns_vect, j * (M + 1) + k, pnl_vect_get(Unkn
              }
            //#########
            //Correction Neumann
            //#########
            //  pnl_vect_set(Unknowns_vect, j*(M+1) + M,
            //  pnl_vect_get(Unknowns_h_line_new,M-1)+pointsx[M]-pointsx[M-1])
            //#########
            //#########

            // Memory desallocation
            pnl_vect_free(&Unknowns_h_line_new);
            pnl_vect_free(&Unknowns_h_line);
          }

        // The line where wolatility is null. Outflow boundary.
        {
          // Extraction of a subvector of Unknowns.
          Unknowns_h_line = pnl_vect_create(M + 1);
          Unknowns_h_line_new = pnl_vect_create(M + 1);
          pnl_vect_extract_subvect(Unknowns_h_line, Unknowns_vect, 0, M + 1);
          pnl_vect_clone(Unknowns_h_line_new, Unknowns_h_line);
          // Construction of the submatrix.
```

```
// First coefficient is 0 in order to respect boundary condition at
// For Neumann boundary condition at x=X, we have two choices :
// boundary condition with first order scheme
// u^l+1_M = u^l+1_M-1
// or boundary condition with second order scheme
//  hl/hr u^l+1_M - (hl/hr-hr/hl) u^l+1_M-1 - hr/hl u^l+1_M-2 = 0
// i.e. u^l+1_M = (1-hr^2/hl^2) u^l+1_M-1 + hr^2/hl^2 u^l+1_M-2
//actualpointy = 0
construction_Ax_coefficients(
  r, divid, alpha, beta, gamma, rho,
  X, M, pointsx, 1,
  Y, N, pointsy, 0,
  omega_global,
  lower_d_Ax, diagonal_Ax,  upper_d_Ax);
// Splitting method.
if (boundary_conditions_method == 0)
  {
    // Homogeneous Neumann.
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_B
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx, U
    // Modification of the last coefficient in order to take into ac
    // Here, we do nothing.
  }
if (boundary_conditions_method == 1)
  {
    // Constant Neumann (= 1.0)
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_B
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx, U
    // Modification of the last coefficient in order to take into ac
    pnl_vect_set(Unknowns_h_line_new, M, pointsx[M] - pointsx[M - 1]
  }
```

```
if (boundary_conditions_method == 2)
  {
    // Constant Neumann (= 1.0)
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_B
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx, U
    // Modification of the last coefficient in order to take into ac
    pnl_vect_set(Unknowns_h_line_new, M, pointsx[M] - pointsx[M - 1]
  }
if (boundary_conditions_method == 3)
  {
    // Constant Neumann (= 1.0)
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_B
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx, U
    // Modification of the last coefficient in order to take into ac
    pnl_vect_set(Unknowns_h_line_new, M, pointsx[M] - pointsx[M - 1]
  }
if (boundary_conditions_method == 4)
  {
    // Dirichlet = max(Spot - E*exp(-rt))
    construction_splitting_matrix_dirichlet(
      0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_B
    construction_splitting_matrix_dirichlet(
      -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx, U
    // Modification of the last coefficient in order to take into ac
    pnl_vect_set(Unknowns_h_line_new, M, MAX(pointsx[M] - E * exp(-r
  }
if (boundary_conditions_method == 5)
  {
```

```
                construction_splitting_matrix_dirichlet(
                  1.0, 1.0, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_Bx);
                // Dirichlet = BS
                construction_splitting_matrix_dirichlet(
                  0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_B
                construction_splitting_matrix_dirichlet(
                  -0.5, deltat, lower_d_Ax, diagonal_Ax, upper_d_Ax, M + 1, Mat_
                // Resolution of the subproblem.
                // Explicit part.
                pnl_tridiag_mat_mult_vect_inplace(Unknowns_h_line_new, Mat_Cx, U
                // Modification of the last coefficient in order to take into ac
                pnl_vect_set(Unknowns_h_line_new, M,
                             pnl_bs_call(pointsx[M], E, deltat * TimeIndex, r, d
              }

//##
            // Implicit part.
// pnl_tridiag_mat_syslin_inplace(Mat_Bx, Unknowns_h_line_new);

            // The implicit resolution is replaced by Brennan-Schwartz.
            // Definition of the payoff for the american condition.
            pnl_vect_resize(Unknowns_max, M + 1);
            if (call_or_put == 1) //Call
              {
                for (l = 0; l < M + 1; l++)
                  {
                    pnl_vect_set(Unknowns_max, l, MAX(pointsx[l] - E, 0.0));
                  }
                brennan_schwartz_tridiag_call(Mat_Bx, M + 1, Unknowns_h_line_new
              }
            if (call_or_put == -1) //Put
              {
                for (l = 0; l < M + 1; l++)
                  {
                    pnl_vect_set(Unknowns_max, l, MAX(E - pointsx[l], 0.0));
                  }
                brennan_schwartz_tridiag_put(Mat_Bx, M + 1, Unknowns_h_line_new,
              }

//##
```

```
  // Update of the unknowns.
  for (k = 0; k <= M; k++)
    {
      pnl_vect_set(Unknowns_vect, k, pnl_vect_get(Unknowns_h_line_new,
    }
  //#########
  //Correction Neumann
  //#########
  //  pnl_vect_set(Unknowns_vect, M,
  //  pnl_vect_get(Unknowns_h_line_new,M-1)+pointsx[M]-pointsx[M-1]);
  //#########
  //#########

  // Memory desallocation
  pnl_vect_free(&Unknowns_h_line_new);
  pnl_vect_free(&Unknowns_h_line);
}

// The line where volatility is maximal. t'Hout & Foulon says Dirichle
// I&T wants Neumann in volatility.
{
  if (boundary_conditions_method == 0)
    {
      // Neumann outflow
      for (i = 0; i <= M; i++)
        {
          if (call_or_put == 1) //Call
            {
              pnl_vect_set(Unknowns_vect, N * (M + 1) + i, pnl_vect_ge
            }
          if (call_or_put == -1) //Put
            {
              pnl_vect_set(Unknowns_vect, N * (M + 1) + i, pnl_vect_ge
            }
        }
    }
  if (boundary_conditions_method == 1)
    {
      // We have Dirichlet condition.
      for (i = 0; i <= M; i++)
        {
```

```
        if (call_or_put == 1)
          {
            pnl_vect_set(Unknowns_vect, N * (M + 1) + i, pointsx[i])
            // Or maybe Condition should be S-E and not just S ???
          }
        if (call_or_put == -1)
          {
            // Not the good condition for the put option !!!!
            pnl_vect_set(Unknowns_vect, N * (M + 1) + i, -pointsx[i]
          }
      }
  }
if (boundary_conditions_method == 2)
  {
    // We have Dirichlet condition.
    for (i = 0; i <= M; i++)
      {
        if (call_or_put == 1)
          {
            pnl_vect_set(Unknowns_vect, N * (M + 1) + i, pointsx[i])
          }
        if (call_or_put == -1)
          {
            // Not the good condition for the put option !!!!
            pnl_vect_set(Unknowns_vect, N * (M + 1) + i, -pointsx[i]
          }
      }
  }
if (boundary_conditions_method == 3)
  {
    // Neumann outflow homogeneous
    for (i = 0; i <= M; i++)
      {
        if (call_or_put == 1) //Call
          {
            pnl_vect_set(Unknowns_vect, N * (M + 1) + i, pnl_vect_ge
          }
        if (call_or_put == -1) //Put
          {
            pnl_vect_set(Unknowns_vect, N * (M + 1) + i, pnl_vect_ge
          }
```

```
        }
    }
  if (boundary_conditions_method == 4)
    {
      // Neumann outflow homogeneous
      for (i = 0; i <= M; i++)
        {
          if (call_or_put == 1) //Call
            {
              pnl_vect_set(Unknowns_vect, N * (M + 1) + i, pnl_vect_ge
            }
          if (call_or_put == -1) //Put
            {
              pnl_vect_set(Unknowns_vect, N * (M + 1) + i, pnl_vect_ge
            }
        }
    }
  if (boundary_conditions_method == 5)
    {
      // Neumann outflow homogeneous
      for (i = 0; i <= M; i++)
        {
          if (call_or_put == 1) //Call
            {
              pnl_vect_set(Unknowns_vect, N * (M + 1) + i, pnl_vect_ge
            }
          if (call_or_put == -1) //Put
            {
              pnl_vect_set(Unknowns_vect, N * (M + 1) + i, pnl_vect_ge
            }
        }
    }

}
// Resolution of various sized problems among the diagonal-direction
// and in two distinct directions.
reorder_unknowns_x_to_xy(Unknowns_vect, M, N, Unknowns_old);
pnl_vect_clone(Unknowns_vect, Unknowns_old);
pnl_tridiag_mat_clone(Mat_Bxy, Mat_Bx);
pnl_tridiag_mat_clone(Mat_Cxy, Mat_Cx);
// First, we should find the central index.
```

```
index_central = 0;
for (k = 0; k < M; k++)
  {
    index_central = index_central + MIN(MIN(k + 1, MIN(M, N) + 1), M +
  }
cumul_index = index_central;
// Then we solve the first diagonal (i.e. the principal diagonal).
{
  number_coeff_on_diagonal = (int) MIN(MIN(M + 1, MIN(M, N) + 1), N +
  // Extraction of a subvector of Unknowns.
  Unknowns_d_line = pnl_vect_create(number_coeff_on_diagonal);
  Unknowns_d_line_new = pnl_vect_create(number_coeff_on_diagonal);
  pnl_vect_extract_subvect(Unknowns_d_line, Unknowns_vect, cumul_index
  pnl_vect_clone(Unknowns_d_line_new, Unknowns_d_line);
  // Construction of the submatrix.
  // First coefficient is 0 in order to respect boundary condition at
  // Last coefficient is 0 for Neumann boundary condition.
  // If M>N then the first point y is 0
  // else the first point y is the (N-M) point in the y grid.
  // If N>M then the first point x is pointsx[0]
  // else the first point x is the (M-N) point in the x grid.
  construction_Axy_coefficients(r, divid, alpha, beta, gamma, rho,
                                X, M, pointsx, (int)MAX(M - N, 0),
                                Y, N, pointsy, (int)MAX(N - M, 0),
                                number_coeff_on_diagonal,
                                omega_global,
                                lower_d_Axy, diagonal_Axy, upper_d_Axy

  // Splitting method.
  if (boundary_conditions_method == 0)
    {
      // Homogeneous Neumann.
      // Be careful. For this diagonal (and only this diagonal), we us
      // method than for x-direction (or y-direction) problem.
      construction_splitting_matrix_neumann(
        0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_co
      construction_splitting_matrix_neumann(
        -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_c
      // Resolution of the subproblem.
      // Explicit part.
      pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy,
```

```
    // Modification of the last coefficient in order to take into ac
    // Here, we do nothing.
  }
if (boundary_conditions_method == 1)
  {
    // Dirichlet (= pointsx)
    // Be careful. For this diagonal (and only this diagonal), we us
    // method than for x-direction (or y-direction) problem.
    construction_splitting_matrix_dirichlet(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_co
    construction_splitting_matrix_dirichlet(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_c
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy,
    // Modification of the last coefficient in order to take into ac
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1,
  }
if (boundary_conditions_method == 2)
  {
    // Dirichlet (= pointsx)
    // Be careful. For this diagonal (and only this diagonal), we us
    // method than for x-direction (or y-direction) problem.
    construction_splitting_matrix_dirichlet(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_co
    construction_splitting_matrix_dirichlet(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_c
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy,
    // Modification of the last coefficient in order to take into ac
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1,
  }
if (boundary_conditions_method == 3)
  {
    // Constant Neumann (= 1.0)
    // Be careful. For this diagonal (and only this diagonal), we us
    // method than for x-direction (or y-direction) problem.
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_co
    construction_splitting_matrix_neumann(
```

```
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_c
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy,
    // Modification of the last coefficient in order to take into ac
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1,
  }
if (boundary_conditions_method == 4)
  {
    construction_splitting_matrix_dirichlet(
      1.0, 1.0, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_coeff
    // Dirichlet = max(Spot - E exp(-rt)
    // Be careful. For this diagonal (and only this diagonal), we us
    // method than for x-direction (or y-direction) problem.
    construction_splitting_matrix_dirichlet(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_co
    construction_splitting_matrix_dirichlet(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_c
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy,
    // Modification of the last coefficient in order to take into ac
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1,
                 MAX(pointsx[M] - E * exp(-r * deltat * TimeIndex),
  }
if (boundary_conditions_method == 5)
  {
    construction_splitting_matrix_dirichlet(
      1.0, 1.0, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_coeff
    // Dirichlet = BS
    // Be careful. For this diagonal (and only this diagonal), we us
    // method than for x-direction (or y-direction) problem.
    construction_splitting_matrix_dirichlet(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_co
    construction_splitting_matrix_dirichlet(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_c
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy,
    // Modification of the last coefficient in order to take into ac
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1,
```

```
                                pnl_bs_call(pointsx[M], E, deltat * TimeIndex, r, d
              }


//##

            // Implicit part.
            pnl_tridiag_mat_syslin_inplace(Mat_Bxy, Unknowns_d_line_new);
            /*
             // The implicit resolution is replaced by Brennan-Schwartz.
             // Definition of the payoff for the american condition.
             pnl_vect_resize(Unknowns_max,number_coeff_on_diagonal);
             if (call_or_put==1) //Call
             {
               for (l=0; l<number_coeff_on_diagonal; l++)
               {
             pnl_vect_set(Unknowns_max, l, MAX(pointsx[M-number_coeff_on_diagona
               }
               brennan_schwartz_tridiag_call(Mat_Bxy, number_coeff_on_diagonal,
             }
             if (call_or_put==-1) //Put
             {
               for (l=0; l<number_coeff_on_diagonal; l++)
               {
             pnl_vect_set(Unknowns_max, l, MAX(E - pointsx[M-number_coeff_on_dia
               }
               brennan_schwartz_tridiag_put(Mat_Bxy, number_coeff_on_diagonal, U
             }
            */
//##

            // Update of the unknowns.
            for (l = 0; l < number_coeff_on_diagonal; l++)
              {
                pnl_vect_set(Unknowns_vect, cumul_index + l , pnl_vect_get(Unkno
              }

            cumul_index = cumul_index + number_coeff_on_diagonal;
          }

          // Then we solve the lower-right diagonals. Decreasing in vol. Fixed i
```

```
penultimate_coeff_on_boundary = pnl_vect_get(Unknowns_d_line_new, numb
central_penultimate_coeff_on_boundary = penultimate_coeff_on_boundary;
// Libération mémoire
pnl_vect_free(&Unknowns_d_line_new);
pnl_vect_free(&Unknowns_d_line);
pnl_tridiag_mat_clone(Mat_Bxy, Mat_Bx);
pnl_tridiag_mat_clone(Mat_Cxy, Mat_Cx);

for (k = M + 1; k < M + N; k++)
  {
    number_coeff_on_diagonal = (int) MIN(MIN(k + 1, MIN(M, N) + 1), M
    // The size of this subproblem is MIN(k+1,MIN(M,N)+1) if k <= MAX(
    // If k > MAX(M,N), the size of this subproblem is M+N+1-k.
    // So the size in general case is MIN(MIN(k+1,MIN(M,N)+1),M+N+1-k)
    // Extraction of a subvector of Unknowns.
    Unknowns_d_line = pnl_vect_create(number_coeff_on_diagonal);
    Unknowns_d_line_new = pnl_vect_create(number_coeff_on_diagonal);
    pnl_vect_extract_subvect(Unknowns_d_line, Unknowns_vect, cumul_ind
    pnl_vect_clone(Unknowns_d_line_new, Unknowns_d_line);
    // Construction of the submatrix.
    // First coefficient is 0 in order to respect boundary condition a
    // Last coefficient is 0... But why ?
    // The boundary condition is not explicit in the I&T's article.
    // So we use the first diagonal in order to specify the Neumann bo
    // If k>N then the first point y is 0
    // else the first point y is the (N-k) point in the y grid.
    // If N>k then the first point x is pointsx[0]
    // else the first point x is the (k-N) point in the x grid.
    construction_Axy_coefficients(r, divid, alpha, beta, gamma, rho,
                                  X, M, pointsx, (int)MAX(k - N, 0),
                                  Y, N, pointsy, (int)MAX(N - k, 0),
                                  number_coeff_on_diagonal,
                                  omega_global,
                                  lower_d_Axy, diagonal_Axy, upper_d_A
    // Resize the matrix...
    pnl_tridiag_mat_resize(Mat_Bxy, number_coeff_on_diagonal);
    pnl_tridiag_mat_resize(Mat_Cxy, number_coeff_on_diagonal);
    // Splitting method.
    if (boundary_conditions_method == 0)
      {
        // Homogeneous Neumann.
```

```
    construction_splitting_matrix_diagonal_neumann(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_neumann(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
  }
if (boundary_conditions_method == 1)
  {
    // Constant Neumann (= 1.0)
    construction_splitting_matrix_diagonal_neumann(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_neumann(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
                 penultimate_coeff_on_boundary + pointsx[M] - poin
  }
if (boundary_conditions_method == 2)
  {
    // Constant Neumann (= 1.0)
    construction_splitting_matrix_diagonal_neumann(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_neumann(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
```

```
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
                 penultimate_coeff_on_boundary + pointsx[M] - poin
  }
if (boundary_conditions_method == 3)
  {
    // Constant Neumann (= 1.0)
    construction_splitting_matrix_diagonal_neumann(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_neumann(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
                 penultimate_coeff_on_boundary + pointsx[M] - poin
  }
if (boundary_conditions_method == 4)
  {
    // Dirichlet = max(Spot - E exp(-rt)
    construction_splitting_matrix_diagonal_dirichlet(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_dirichlet(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
                 MAX(pointsx[M] - E * exp(-r * deltat * TimeIndex)
  }
if (boundary_conditions_method == 5)
  {
    construction_splitting_matrix_dirichlet(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_
    construction_splitting_matrix_dirichlet(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy, number
```

```
    // Dirichlet = BS
    construction_splitting_matrix_diagonal_dirichlet(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_dirichlet(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    // What is the var/vol at the last point
    // The diagonal begins at point (int)MAX(N-k,0) so it finishes
    // (int)MAX(N-k,0)+number_coeff_on_diagonal-1
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
                 pnl_bs_call(pointsx[M], E, deltat * TimeIndex, r,
                             (int)MAX(N - k, 0) + number_coeff_o
                            ]));
  }


//##

                // Implicit part.
                pnl_tridiag_mat_syslin_inplace(Mat_Bxy, Unknowns_d_line_new);
                /* // The implicit resolution is replaced by Brennan-Schwartz.
                 // Definition of the payoff for the american condition.
                 pnl_vect_resize(Unknowns_max, number_coeff_on_diagonal);
                 if (call_or_put==1) //Call
                 {
                   for (l=0; l<number_coeff_on_diagonal; l++)
                   {
                 pnl_vect_set(Unknowns_max, l, MAX(pointsx[M-number_coeff_on_diago
                   }
                   brennan_schwartz_tridiag_call(Mat_Bxy, number_coeff_on_diagonal
                 }
                 if (call_or_put==-1) //Put
                 {
                   for (l=0; l<number_coeff_on_diagonal; l++)
                   {
                 pnl_vect_set(Unknowns_max, l, MAX(E - pointsx[M-number_coeff_on_d
```

```
                }
                brennan_schwartz_tridiag_put(Mat_Bxy, number_coeff_on_diagonal,
              }
             */

//##


             // Update of the unknowns.
             for (l = 0; l < number_coeff_on_diagonal; l++)
               {
                 pnl_vect_set(Unknowns_vect, cumul_index + l , pnl_vect_get(Unk
               }
             cumul_index = cumul_index + number_coeff_on_diagonal;
             if (number_coeff_on_diagonal > 1)
               {
                 // if number_coeff_on_diagonal==1 so this is the end of the lo
                 // and the penultimate coefficient is useless (and nonsense).
                 // And in fact, this case will never occur.
                 penultimate_coeff_on_boundary = pnl_vect_get(Unknowns_d_line_n
               }
             // Memory desallocation
             pnl_vect_free(&Unknowns_d_line_new);
             pnl_vect_free(&Unknowns_d_line);
           }
         // The unknowns in the lower-right corner is not modified...
         // pnl_vect_set(Unknowns_vect, size_Unknowns-1, pnl_vect_get(Unknowns_
         // Then we solve the upper-left diagonals. Decreasing in spot. Fixed i
         penultimate_coeff_on_boundary = central_penultimate_coeff_on_boundary;
         cumul_index = index_central;
         pnl_tridiag_mat_clone(Mat_Bxy, Mat_Bx);
         pnl_tridiag_mat_clone(Mat_Cxy, Mat_Cx);

         for (k = M - 1; k > 0; k--)
           {
             number_coeff_on_diagonal = (int) MIN(MIN(k + 1, MIN(M, N) + 1), M
             // The size of this subproblem is MIN(k+1,MIN(M,N)+1) if k <= MAX(
             // If k > MAX(M,N), the size of this subproblem is M+N+1-k.
             // So the size in general case is MIN(MIN(k+1,MIN(M,N)+1),M+N+1-k)
             cumul_index = cumul_index - number_coeff_on_diagonal;
             // Extraction of a subvector of Unknowns.
```

```
Unknowns_d_line = pnl_vect_create(number_coeff_on_diagonal);
Unknowns_d_line_new = pnl_vect_create(number_coeff_on_diagonal);

pnl_vect_extract_subvect(Unknowns_d_line, Unknowns_vect, cumul_ind
pnl_vect_clone(Unknowns_d_line_new, Unknowns_d_line);
// Construction of the submatrix.
// First coefficient is 0 in order to respect boundary condition a
// Last coefficient is 0... But why ?
// The boundary condition is not explicit in the I&T's article.
// So we use the first diagonal in order to specify the Neumann bo
// If k>N then the first point y is 0
// else the first point y is the (N-k) point in the y grid.
// If N>k then the first point x is pointsx[0]
// else the first point x is the (k-N) point in the x grid.
construction_Axy_coefficients(r, divid, alpha, beta, gamma, rho,
                              X, M, pointsx, (int)MAX(k - N, 0),
                              Y, N, pointsy, (int)MAX(N - k, 0),
                              number_coeff_on_diagonal,
                              omega_global,
                              lower_d_Axy, diagonal_Axy, upper_d_A
// Resize the matrix...
pnl_tridiag_mat_resize(Mat_Bxy, number_coeff_on_diagonal);
pnl_tridiag_mat_resize(Mat_Cxy, number_coeff_on_diagonal);
// Splitting method.
if (boundary_conditions_method == 0)
  {
    // Homogeneous Neumann.
    construction_splitting_matrix_diagonal_neumann(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_neumann(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
  }
if (boundary_conditions_method == 1)
  {
```

```
    // Dirichlet (= pointsx)
    construction_splitting_matrix_diagonal_dirichlet(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_dirichlet(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    // The pointsx index is max((k-N),0)+min(N+1,k+1)-1 = k.
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
  }
if (boundary_conditions_method == 2)
  {
    // Dirichlet (= pointsx)
    construction_splitting_matrix_diagonal_dirichlet(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_dirichlet(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    // The pointsx index is max((k-N),0)+min(N+1,k+1)-1 = k.
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
  }
if (boundary_conditions_method == 3)
  {
    // Homogeneous Neumann.
    construction_splitting_matrix_diagonal_neumann(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_neumann(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
```

```
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
  }
if (boundary_conditions_method == 4)
  {
    // Homogeneous Neumann.
    construction_splitting_matrix_diagonal_neumann(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_neumann(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
  }
if (boundary_conditions_method == 5)
  {
    construction_splitting_matrix_dirichlet(
      1.0, 1.0, lower_d_Axy, diagonal_Axy, upper_d_Axy, number_coe
    // Homogeneous Neumann.
    construction_splitting_matrix_diagonal_neumann(
      0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Bxy);
    construction_splitting_matrix_diagonal_neumann(
      -0.5, deltat, lower_d_Axy, diagonal_Axy, upper_d_Axy,
      number_coeff_on_diagonal, Mat_Cxy);
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_d_line_new, Mat_Cxy
    // Modification of the last coefficient in order to take into
    pnl_vect_set(Unknowns_d_line_new, number_coeff_on_diagonal - 1
  }


//##

// Implicit part.
```

```
pnl_tridiag_mat_syslin_inplace(Mat_Bxy, Unknowns_d_line_new);
/*     // If N>k then the first point x is pointsx[0]
 // so the last point is pointsx[number_coeff_on_diagonal-1]
     // else the first point x is the (k-N) point in the x grid
 // so the last point is pointsx[(k-N) + number_coeff_on_diagonal-

 // The implicit resolution is replaced by Brennan-Schwartz.
 // Definition of the payoff for the american condition.
 pnl_vect_resize(Unknowns_max,number_coeff_on_diagonal);
 if (call_or_put==1) //Call
 {
   for (l=0; l<number_coeff_on_diagonal; l++)
   {
     if (N>k)
 {
   pnl_vect_set(Unknowns_max, l, MAX(pointsx[l] - E,0.0));
 }
 else
 {
   pnl_vect_set(Unknowns_max, l, MAX(pointsx[(k-N)+l] - E,0.0));
 }
   }
   brennan_schwartz_tridiag_call(Mat_Bxy, number_coeff_on_diagonal
 }
 if (call_or_put==-1) //Put
 {
   for (l=0; l<number_coeff_on_diagonal; l++)
   {
     if (N>k)
 {
   pnl_vect_set(Unknowns_max, l, MAX(E - pointsx[l],0.0));
 }
     else
 {
   pnl_vect_set(Unknowns_max, l, MAX(E - pointsx[(k-N)+l],0.0));
 }
   }
   brennan_schwartz_tridiag_put(Mat_Bxy, number_coeff_on_diagonal,
 }

*/
```

```
//##

              // Update of the unknowns.
              for (l = 0; l < number_coeff_on_diagonal; l++)
                {
                  pnl_vect_set(Unknowns_vect, cumul_index + l , pnl_vect_get(Unk

                }
              if (number_coeff_on_diagonal > 1)
                {
                  // if number_coeff_on_diagonal==1 so this is the end of the lo
                  // and the penultimate coefficient is useless (and nonsense).
                  // And in fact, this case will never occur.
                  penultimate_coeff_on_boundary = pnl_vect_get(Unknowns_d_line_n
                }
              // Libération mémoire
              pnl_vect_free(&Unknowns_d_line_new);
              pnl_vect_free(&Unknowns_d_line);
            }
          // The unknowns in the upper-left corner is not modified...
          // pnl_vect_set(Unknowns_vect, 0, pnl_vect_get(Unknowns_vect, 0));

          reorder_unknowns_xy_to_y(Unknowns_vect, M, N, Unknowns_old);
          pnl_vect_clone(Unknowns_vect, Unknowns_old);

          //  Resolution of M-1 problems among the y-direction.
          for (i = 1; i < M; i++)
            {
              // Extraction of a subvector of Unknowns.
              Unknowns_v_line = pnl_vect_create(N + 1);
              Unknowns_v_line_new = pnl_vect_create(N + 1);
              pnl_vect_extract_subvect(Unknowns_v_line, Unknowns_vect, i * (N +
              pnl_vect_clone(Unknowns_v_line_new, Unknowns_v_line);
// Construction of the submatrix.
              // First coefficient is 0 in order to respect boundary condition a
              // For Neumann boundary condition at y=Y, we have two choices :
              // boundary condition with first order scheme
              // u^l+1_M = u^l+1_M-1
              // or boundary condition with second order scheme
              // u^l+1_M = u^l+1_M-2
              //actualpointx = pointsx[i];
```

```
construction_Ay_coefficients(
  r, divid, alpha, beta, gamma, rho,
  X, M, pointsx, i,
  Y, N, pointsy, 1,
  omega_global,
  lower_d_Ay, diagonal_Ay,  upper_d_Ay);
// Splitting method.
if (boundary_conditions_method == 0)
  {
    // Homogeneous Neumann.
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Mat
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Ma
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_v_line_new, Mat_Cy,
    // Modification of the last coefficient in order to take into
    // Here we do nothing.
  }
if (boundary_conditions_method == 1)
  {
    // Dirichlet (= pointsx)
    construction_splitting_matrix_dirichlet(
      0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Mat
    construction_splitting_matrix_dirichlet(
      -0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Ma
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_v_line_new, Mat_Cy,
    // Modification of the last coefficient in order to take into
    // The pointsx index is i.
    pnl_vect_set(Unknowns_v_line_new, N, pointsx[i]);
  }
if (boundary_conditions_method == 2)
  {
    // Dirichlet (= pointsx)
    construction_splitting_matrix_dirichlet(
      0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Mat
    construction_splitting_matrix_dirichlet(
      -0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Ma
```

```
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_v_line_new, Mat_Cy,
    // Modification of the last coefficient in order to take into
    // The pointsx index is i.
    pnl_vect_set(Unknowns_v_line_new, N, pointsx[i]);
  }
if (boundary_conditions_method == 3)
  {
    // Homogeneous Neumann.
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Mat
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Ma
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_v_line_new, Mat_Cy,
    // Modification of the last coefficient in order to take into
    // Here we do nothing.
  }
if (boundary_conditions_method == 4)
  {
    // Homogeneous Neumann.
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Mat
    construction_splitting_matrix_neumann(
      -0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Ma
    // Resolution of the subproblem.
    // Explicit part.
    pnl_tridiag_mat_mult_vect_inplace(Unknowns_v_line_new, Mat_Cy,
    // Modification of the last coefficient in order to take into
    // Here we do nothing.
  }
if (boundary_conditions_method == 5)
  {
    construction_splitting_matrix_neumann(
      1.0, 1.0, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Mat_By
    // Homogeneous Neumann.
    construction_splitting_matrix_neumann(
      0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Mat
    construction_splitting_matrix_neumann(
```

```
                  -0.5, deltat, lower_d_Ay, diagonal_Ay, upper_d_Ay, N + 1, Ma
                // Resolution of the subproblem.
                // Explicit part.
                pnl_tridiag_mat_mult_vect_inplace(Unknowns_v_line_new, Mat_Cy,
                // Modification of the last coefficient in order to take into
                // Here we do nothing.
              }

//##

              // Implicit part.
// pnl_tridiag_mat_syslin_inplace(Mat_By, Unknowns_v_line_new);
              // The implicit resolution is replaced by Brennan-Schwartz.
              // Definition of the payoff for the american condition.
              pnl_vect_resize(Unknowns_max, N + 1);
              if (call_or_put == 1) //Call
                {
                  for (l = 0; l < N + 1; l++)
                    {
                      pnl_vect_set(Unknowns_max, l, MAX(pointsx[i] - E, 0.0));
                    }
                  brennan_schwartz_tridiag_call(Mat_By, N + 1, Unknowns_v_line_n
                }
              if (call_or_put == -1) //Put
                {
                  for (l = 0; l < N + 1; l++)
                    {
                      pnl_vect_set(Unknowns_max, l, MAX(E - pointsx[i], 0.0));
                    }
                  brennan_schwartz_tridiag_put(Mat_By, N + 1, Unknowns_v_line_ne
                }


//##

              // Update of the unknowns.
              for (k = 0; k <= N; k++)
                {
                  pnl_vect_set(Unknowns_vect, i * (N + 1) + k, pnl_vect_get(Unkn
                }
              // Libération mémoire
```

```
    pnl_vect_free(&Unknowns_v_line_new);
    pnl_vect_free(&Unknowns_v_line);
  }

// The line where spot is 0.
{
  if (boundary_conditions_method == 0)
    {
      // Dirichlet homogeneous
      for (k = 0; k <= N; k++)
        {
          if (call_or_put == 1) //Call
            {
              pnl_vect_set(Unknowns_vect, k, 0.0);
            }
          if (call_or_put == -1) //Put
            {
              pnl_vect_set(Unknowns_vect, k, E * exp(-r * TimeIndex *
            }
        }
    }
  if (boundary_conditions_method == 1)
    {
      // Dirichlet homogeneous
      for (k = 0; k <= N; k++)
        {
          if (call_or_put == 1) //Call
            {
              pnl_vect_set(Unknowns_vect, k, 0.0);
            }
          if (call_or_put == -1) //Put
            {
              pnl_vect_set(Unknowns_vect, k, E * exp(-r * TimeIndex *
            }
        }
    }
  if (boundary_conditions_method == 2)
    {
      // Dirichlet homogeneous
      for (k = 0; k <= N; k++)
        {
```

```
        if (call_or_put == 1) //Call
          {
            pnl_vect_set(Unknowns_vect, k, 0.0);
          }
        if (call_or_put == -1) //Put
          {
            pnl_vect_set(Unknowns_vect, k, E * exp(-r * TimeIndex *
          }
      }
  }
if (boundary_conditions_method == 3)
  {
    // Dirichlet homogeneous
    for (k = 0; k <= N; k++)
      {
        if (call_or_put == 1) //Call
          {
            pnl_vect_set(Unknowns_vect, k, 0.0);
          }
        if (call_or_put == -1) //Put
          {
            pnl_vect_set(Unknowns_vect, k, E * exp(-r * TimeIndex *
          }
      }
  }
if (boundary_conditions_method == 4)
  {
    // Dirichlet homogeneous
    for (k = 0; k <= N; k++)
      {
        if (call_or_put == 1) //Call
          {
            pnl_vect_set(Unknowns_vect, k, 0.0);
          }
        if (call_or_put == -1) //Put
          {
            pnl_vect_set(Unknowns_vect, k, E * exp(-r * TimeIndex *
          }
      }
  }
if (boundary_conditions_method == 5)
```

```
    {
      for (k = 0; k <= N; k++)
        {
          if (call_or_put == 1) //Call
            {
              pnl_vect_set(Unknowns_vect, k, 0.0);
            }
          if (call_or_put == -1) //Put
            {
              pnl_vect_set(Unknowns_vect, k, E * exp(-r * TimeIndex *
            }
        }
    }
}

// The line where spot is maximal. t'Hout & Foulon recommande Neumann
// I&T recommande Neumann =0. Est-ce que la frontiere est outflow ou i
{
  if (boundary_conditions_method == 0)
    {
      // Neumann outflow homogeneous
      for (k = 0; k <= N; k++)
        {
          if (call_or_put == 1) //Call
            {
              pnl_vect_set(Unknowns_vect, M * (N + 1) + k, pnl_vect_ge
            }
          if (call_or_put == -1) //Put
            {
              pnl_vect_set(Unknowns_vect, M * (N + 1) + k, pnl_vect_ge
            }
        }
    }
  if (boundary_conditions_method == 1)
    {
      // Neumann outflow = 1
      for (k = 0; k <= N; k++)
        {
          if (call_or_put == 1) //Call
            {
              pnl_vect_set(Unknowns_vect, M * (N + 1) + k, pnl_vect_ge
```

```
                        + pointsx[M] - pointsx[M - 1]);
            }
          if (call_or_put == -1) //Put (NOT THE RIGHT CONDITION HERE)
            {
              pnl_vect_set(Unknowns_vect, M * (N + 1) + k, pnl_vect_ge
                        + pointsx[M] - pointsx[M - 1]);
            }
        }
    }
  if (boundary_conditions_method == 2)
    {
      // Neumann outflow = 1
      for (k = 0; k <= N; k++)
        {
          if (call_or_put == 1) //Call
            {
              pnl_vect_set(Unknowns_vect, M * (N + 1) + k, pnl_vect_ge
                        + pointsx[M] - pointsx[M - 1]);
            }
          if (call_or_put == -1) //Put (NOT THE RIGHT CONDITION HERE)
            {
              pnl_vect_set(Unknowns_vect, M * (N + 1) + k, pnl_vect_ge
                        + pointsx[M] - pointsx[M - 1]);
            }
        }
    }
  if (boundary_conditions_method == 3)
    {
      // Neumann outflow = 1
      for (k = 0; k <= N; k++)
        {
          if (call_or_put == 1) //Call
            {
              pnl_vect_set(Unknowns_vect, M * (N + 1) + k, pnl_vect_ge
                        + pointsx[M] - pointsx[M - 1]);
            }
          if (call_or_put == -1) //Put (NOT THE RIGHT CONDITION HERE)
            {
              pnl_vect_set(Unknowns_vect, M * (N + 1) + k, pnl_vect_ge
                        + pointsx[M] - pointsx[M - 1]);
            }
```

```
            }
          }
        if (boundary_conditions_method == 4)
          {
            // Dirichlet = max(Spot - E exp(-rt)
            for (k = 0; k <= N; k++)
              {
                if (call_or_put == 1) //Call
                  {
                    pnl_vect_set(Unknowns_vect, M * (N + 1) + k, MAX(pointsx
                  }
                if (call_or_put == -1) //Put
                  {
                    pnl_vect_set(Unknowns_vect, M * (N + 1) + k, MAX(E * exp
                  }
              }
          }
        if (boundary_conditions_method == 5)
          {
            // Dirichlet = BS
            for (k = 0; k <= N; k++)
              {
                if (call_or_put == 1) //Call
                  {
                    pnl_vect_set(Unknowns_vect, M * (N + 1) + k,
                                 pnl_bs_call(pointsx[M], E, deltat * TimeInd
                  }
                if (call_or_put == -1) //Put
                  {
                    pnl_vect_set(Unknowns_vect, M * (N + 1) + k,
                                 pnl_bs_put(pointsx[M], E, deltat * TimeInde
                  }
              }
          }
      }

      reorder_unknowns_y_to_x(Unknowns_vect, M, N , Unknowns_old);
      pnl_vect_clone(Unknowns_vect, Unknowns_old);

    }
}
```

```
// Index in the domain for the price.
// Find the index in pointsx corresponding to the price x of the asset.
IndexX = lower_index(pointsx, M + 1, x);
// Find the index in [0,Y] corresponding to the variance y of the asset.
//IndexY = (int)(y/h);
IndexY = lower_index(pointsy, N + 1, y);
// Compute the index in the vector of unknowns
IndexUnknown = IndexX + IndexY * (M + 1);

// Price.
// We do an interpolation.
price_ul = pnl_vect_get(Unknowns_vect, IndexUnknown + M + 1);
price_ur = pnl_vect_get(Unknowns_vect, IndexUnknown + 1 + M + 1);
price_dl = pnl_vect_get(Unknowns_vect, IndexUnknown);
price_dr = pnl_vect_get(Unknowns_vect, IndexUnknown + 1);

*ptprice =
  (
    (pointsy[IndexY + 1] - y) *
    (
      (x - pointsx[IndexX]) * price_dr
      +
      (pointsx[IndexX + 1] - x) * price_dl
    )
    / (pointsx[IndexX + 1] - pointsx[IndexX])
    +
    (y - pointsy[IndexY]) *
    (
      (x - pointsx[IndexX]) * price_ur
      +
      (pointsx[IndexX + 1] - x) * price_ul
    )
    / (pointsx[IndexX + 1] - pointsx[IndexX])
  )
  / (pointsy[IndexY + 1] - pointsy[IndexY]);

pnl_vect_get(Unknowns_vect, IndexUnknown);

// Delta.
if (x - pointsx[IndexX] > 0)
```

```
  {
    *ptdelta =
      (
        ((((pointsy[IndexY + 1] - y) * price_dr + (y - pointsy[IndexY]) * pric
          / (pointsy[IndexY + 1] - pointsy[IndexY])) - *ptprice)
        /
        (pointsx[IndexX + 1] - x)
        +
        (*ptprice - (((pointsy[IndexY + 1] - y) * price_dl + (y - pointsy[Inde
                     / (pointsy[IndexY + 1] - pointsy[IndexY])))
        /
        (x - pointsx[IndexX])
      ) / 2.0;
  }
else
  {
    *ptdelta =
      ((((pointsy[IndexY + 1] - y) * price_dr + (y - pointsy[IndexY]) * price_
        (pointsy[IndexY + 1] - pointsy[IndexY])) - *ptprice)
      /
      (pointsx[IndexX + 1] - x);

  }

// Memory desallocation.
free(pointsx);
free(pointsy);
free(Unknowns);
free(upper_d_Ax);
free(diagonal_Ax);
free(lower_d_Ax);
free(upper_d_Ay);
free(diagonal_Ay);
free(lower_d_Ay);
free(upper_d_Axy);
free(diagonal_Axy);
free(lower_d_Axy);

pnl_vect_free(&Unknowns_vect);
pnl_vect_free(&Unknowns_old);
pnl_vect_free(&Unknowns_max);
```

```
  pnl_tridiag_mat_free(&Mat_Bx);
  pnl_tridiag_mat_free(&Mat_Cx);
  pnl_tridiag_mat_free(&Mat_Bxy);
  pnl_tridiag_mat_free(&Mat_Cxy);
  pnl_tridiag_mat_free(&Mat_By);
  pnl_tridiag_mat_free(&Mat_Cy);

  return 0.;
}

static int FDIkonenToivanen_Heston(int am, double S0, double V0, NumFunc_1  *p,
{
  double K;
  int call_or_put;

  int S_max = 800;
  int v_max = 1;
  //Mesh parameters
  double k_mesh = 5.;
  double coeff_for_x_1 = 10;
  double omega = 0.5;
  int boundary_conditions_method = 5;


  K = p->Par[0].Val.V_DOUBLE;
  if ((p->Compute) == &Call)
    call_or_put = 1;
  else
    call_or_put = -1;


  /* 0 = Ikonen Toivanen = Dirichlet en vol = 0 est étrange. Cela devrait dépend
   Neumann = homogeneous
   ------------------------
  Dirichlet |                        | Neumann
  = |                       | = homogeneous
  Call |                          |
  ->homogeneous |                        |
   |_______________________|
   Dirichlet = Call_T = max(Spot-E)
```

```
*/
/* 1 = In t Hout Foulon = Outflow boundary n'est pas la bonne solution. En fai
en vol = 0 est donnée par la solution de l'EDP en pluggant directement vol=0 d
 Dirichlet = s

 ________________________
Dirichlet |                        | Neumann
= |                         | = 1
Call |                          |
->homogeneous |                          |/
 |________________________|
 Outflow boundary
*/
/* 2 = On suit In t Hout et Foulon sur les bords lointains, mais avec une cond
en vol = 0 qui est presque cohérente avec l'EDP où on a mis brutalement la vol
 Dirichlet = s

 ________________________
Dirichlet |                        | Neumann
= |                         | = 1
Call |                          |
->homogeneous |                          |
 |________________________|
 Dirichlet = Call_t = max(Spot - E exp(-rt))
*/
/* 3 = On suit Ikonen et Toivanen sur les bords lointains, mais avec une condi
en vol = 0 qui est presque cohérente avec l'EDP où on a mis brutalement la vol
 Neumann = homogeneous in vol (or impose neuman = 1 in strike so it is affine

 ________________________
Dirichlet |                        | Neumann
= |                         | = 1
Call |                          |
->homogeneous |                          |
 |________________________|
 Dirichlet = Call_t = max(Spot - E exp(-rt))
*/
/* 4 = Mélange des différents conditions aux limites.
 Neumann (Vol max) = homogeneous in vol

 ________________________
Dirichlet |                        |
= |                         | Dirichlet (Spot max) = max(Spot - E exp(-rt))
Call |                          |
->homogeneous |                          |
```

```
   |_____________________|
   Outflow boundary
  */


  //mt Spot grid numbers
  //N Volatility grid numbers
  //L Time Step Numbers
  if (am == 0)
    ComSplitEu(S0, V0, T, r, divid, kappa, theta, sigma, rho, K, S_max, v_max, m
               ptprice, ptdelta);

  else
    {
      boundary_conditions_method = 5;
      ComSplitAm(S0, V0, T, r, divid, kappa, theta, sigma, rho, K, S_max, v_max,
                 ptprice, ptdelta);
    }

  return OK;
}

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

  if (ptMod->Sigma.Val.V_PDOUBLE == 0.0)
    {
      Fprintf(TOSCREEN, "BLACK-SCHOLES MODEL\ n\ n\ n");
      return WRONG;
    }
  else
    {
      r = log(1. + ptMod->R.Val.V_DOUBLE / 100.);
      divid = log(1. + ptMod->Divid.Val.V_DOUBLE / 100.);

      return FDIkonenToivanen_Heston(ptOpt->EuOrAm.Val.V_BOOL, ptMod->S0.Val.V_P
                                     ptMod->Sigma0.Val.V_PDOUBLE,
                                     ptOpt->PayOff.Val.V_NUMFUNC_1,
```

```
                                        ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V
                                        r,
                                        divid, ptMod->Sigma.Val.V_PDOUBLE,
                                        ptMod->Rho.Val.V_PDOUBLE,
                                        ptMod->MeanReversion.Val.V_PDOUBLE,
                                        ptMod->LongRunVariance.Val.V_PDOUBLE,
                                        Met->Par[0].Val.V_INT, Met->Par[1].Val.V_IN
                                        &(Met->Res[0].Val.V_DOUBLE),
                                        &(Met->Res[1].Val.V_DOUBLE)
                                       );
    }
}


static int CHK_OPT(FD_IkonenToivanen_Heston)(void *Opt, void *Mod)
{
  if ((strcmp(((Option *)Opt)->Name, "CallEuro") == 0) || (strcmp(((Option *)Opt
      (strcmp(((Option *)Opt)->Name, "PutAmer") == 0))
    return OK;

  return WRONG;
}

#endif //PremiaCurrentVersion
static int MET(Init)(PricingMethod *Met, Option *Opt)
{
  if (Met->init == 0)
    {
      Met->init = 1;
      Met->HelpFilenameHint = "fd_ikonentoivanen_heston";

      Met->Par[0].Val.V_INT2 = 1000;
      Met->Par[1].Val.V_INT2 = 101;
      Met->Par[2].Val.V_INT2 = 50;
    }

  return OK;
}

PricingMethod MET(FD_IkonenToivanen_Heston) =
{
```

```
  "FD_IkonenToivanen_Heston",
  { {"Time Step", INT2, {100}, ALLOW}, {"SpaceStepNumber S", INT2, {100}, ALLOW}
    , {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(FD_IkonenToivanen_Heston),
  { {"Price", DOUBLE, {100}, FORBID},
    {"Delta", DOUBLE, {100}, FORBID} ,
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(FD_IkonenToivanen_Heston),
  CHK_ok,
  MET(Init)
};
```