Help

```
#include  "
href../../../mod/doublehes1d/doublehes1d_std/doublehes1d_std_h_src.pdfhes1d_std.

#include "pnl/pnl_mathtools.h"
#include "pnl/pnl_finance.h"
#include "pnl/pnl_tridiag_matrix.h"
#include "pnl/pnl_band_matrix.h"

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

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

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

#else


static double asinh1(double value)
{
  double returned;
  if (value > 0)
    returned = log(value + sqrt(value * value + 1));
  else
    returned = -log(-value + sqrt(value * value + 1));
  return (returned);
}

static void grid_generation_HF_spot(double *pointsx, double Smax, double K, int
{
  // Tests on parameters would be useful.
  int i;
  double temp = asinh1(-K / c);
  double deltaxi = (asinh1((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 void grid_generation_HF_variance(double *pointsy, double Vmax, int m2, do
{
  // Tests on parameters would be useful.
  int j;
  double deltaeta = (asinh1(Vmax / d)) / m2;

  // Definition of uniform grid eta.
  for (j = 0; j <= m2; j++)
    {
      pointsy[j] = j *  deltaeta;
    }

  // Definition of the volatility grid with the uniform grid eta.
  pointsy[0] = 0;
  for (j = 1; j <= m2; j++)
    {
      pointsy[j] = d * sinh(pointsy[j]);
    }
}

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 double interpolation(double griddown, double valuedown,
                            double gridup, double valueup, double gridunknown)
{
  return (valueup - valuedown) / (gridup - griddown) * (gridunknown - griddown)
}

static int it_exists_stencil(int i, int M, int j, int N, int stencil)
{
  // We use i from 1 to M && j from 0 to N-1
  // We use points i-1 -> i+1 && j-2 -> j+2

  //    *                   6
  //  * * * 9 5 10
  //  * * *     <=>1 0 2
  //  * * * 7 4 8
  //    *                   3

  if ((i > 1) && (i < M) && (j > 1) && (j < N - 2))
    {
      return TRUE;
    }

  if (i == 1)
```

```
    if ((stencil == 1) || (stencil == 7) || (stencil == 9))
      return FALSE;

  if (i == M)
    if ((stencil == 2) || (stencil == 8) || (stencil == 10))
      return FALSE;

  if (j == 0)
    if ((stencil == 3) || (stencil == 4) || (stencil == 7) || (stencil == 8))
      return FALSE;

  if (j == 1)
    if (stencil == 3)
      return FALSE;

  if (j == N - 2)
    if (stencil == 6)
      return FALSE;

  if (j == N - 1)
    if ((stencil == 5) || (stencil == 6) || (stencil == 9) || (stencil == 10))
      return FALSE;

  return TRUE;
}

static void point_of_stencil(int i, int j, int stencil, int *pi, int *pj)
{
  //    *                   6
  //  * * * 9 5 10
  //  * * *     <=>1 0 2
  //  * * * 7 4 8
  //    *                   3

  if (stencil == 0)
    {
      *pi = i;
      *pj = j;
    }
  if (stencil == 1)
    {
```

```
    *pi = i - 1;
    *pj = j;
  }
if (stencil == 2)
  {
    *pi = i + 1;
    *pj = j;
  }
if (stencil == 3)
  {
    *pi = i;
    *pj = j - 2;
  }
if (stencil == 4)
  {
    *pi = i;
    *pj = j - 1;
  }
if (stencil == 5)
  {
    *pi = i;
    *pj = j + 1;
  }
if (stencil == 6)
  {
    *pi = i;
    *pj = j + 2;
  }
if (stencil == 7)
  {
    *pi = i - 1;
    *pj = j - 1;
  }
if (stencil == 8)
  {
    *pi = i + 1;
    *pj = j - 1;
  }
if (stencil == 9)
  {
    *pi = i - 1;
```

```
      *pj = j + 1;
    }
  if (stencil == 10)
    {
      *pi = i + 1;
      *pj = j + 1;
    }
}

static void spatial_step_size(int i, int M, double *spoint,
                              int j, int N, double *vpoint,
                              double *pDsi, double *pDsip1,
                              double *pDvjm1, double *pDvj,
                              double *pDvjp1, double *pDvjp2)
{
  // For the s variable, there is only central scheme
  // both for diffusion && convection.
  // At point near S=Smin, there is a Dirichlet condition.
  // At point S=Smax, there is a Neumann condition.

  if (i == M) // Neumann boundary condition && Central scheme
    {
      *pDsi = spoint[i] - spoint[i - 1];
      *pDsip1 = *pDsi; //N
    }
  else // Central scheme
    {
      *pDsi = spoint[i] - spoint[i - 1];
      *pDsip1 = spoint[i + 1] - spoint[i];
    }

  // For the v variable, there is only central scheme for diffusion.
  // But there are multiple schemes for convection.
  // At point near V=Smin, the scheme is forward.
  // At point before long term variance beta, there is central scheme.
  // At point after long term variance beta, there is backward scheme.
  // At point V=Vmax, there is a Dirichlet condition.

  if (j == 0) // Forward scheme
    {
      *pDvjp2 = vpoint[j + 2] - vpoint[j + 1];
```

```
      *pDvjp1 = vpoint[j + 1] - vpoint[j];
      *pDvj = -1.; //!
      *pDvjm1 = -1.; //!
    }
  if (j == 1) // Central scheme
    {
      *pDvjp2 = vpoint[j + 2] - vpoint[j + 1];
      *pDvjp1 = vpoint[j + 1] - vpoint[j];
      *pDvj = vpoint[j] - vpoint[j - 1];
      *pDvjm1 = -1.; //!
    }
  if (j == N - 1) // Dirichlet boundary condition && Central or Backward scheme
    {
      *pDvjm1 = vpoint[j - 1] - vpoint[j - 2];
      *pDvj = vpoint[j] - vpoint[j - 1];
      *pDvjp1 = vpoint[j + 1] - vpoint[j];
      *pDvjp2 = -1.; //!
    }
  if ((j > 1) && (j < N - 1)) // Central or Backward scheme
    {
      *pDvjm1 = vpoint[j - 1] - vpoint[j - 2];
      *pDvj = vpoint[j] - vpoint[j - 1];
      *pDvjp1 = vpoint[j + 1] - vpoint[j];
      *pDvjp2 = -1.; //!
    }

}

static double Dirichlet_Smin(double *spoint, int i, double *vpoint, int j,
                             double r, double divid, double strike, double time,
{
  // Dirichlet boundary condition at point (s[i],v[j])
  if (call_or_put == 1)
    return 0.;
  else
    return strike * exp(-r * time);
}

static double Neumann_Smax(double *spoint, int i, double *vpoint, int j,
                           double r, double divid, double strike, double time, i
{
```

```
  // Neumann boundary condition at point (s[i],v[j])
  if (call_or_put == 1)
    return exp(-divid * time);
  else
    return 0.;
}

static double Dirichlet_Vmax(double *spoint, int i, double *vpoint, int j,
                             double r, double divid, double strike, double time,
{
  // Dirichlet boundary condition at point (s[i],v[j])
  if (call_or_put == 1)
    return spoint[i] * exp(-divid * time);
  else
    return strike * exp(-r * time);
}

static int value_of_boundary_condition(int i, int M, double *spoint,
                                       int j, int N, double *vpoint,
                                       double r, double divid, double strike, do
                                       int stencil, int call_or_put,
                                       double *pvalue_scmb, double *pvalue_coeff
{
  // This function returns the value of the second member in "value_scmb"
  // && the modification coefficient in "value_coeff".
  // The valued returned by the function is used to modify the matrix.
  // It is the stencil number representing the coefficient which
  // must be changed (often the diagonal coefficient).

  // We use i from 1 to M && j from 0 to N-1
  // We use points i-1 -> i+1 && j-2 -> j+2

  //    *                  6
  //  * * * 9 5 10
  //  * * *     <=>1 0 2
  //  * * * 7 4 8
  //    *                  3


  double Dsi, Dsip1, Dvjm1, Dvj, Dvjp1, Dvjp2;
  *pvalue_scmb = 0.;
```

```
*pvalue_coeff = 0.;


if ((i > 1) && (i < M) && (j > 1) && (j < N - 2))
  {
    *pvalue_scmb = 0; // No change in the second member.
    *pvalue_coeff = 0; // No change in the matrix.
    return 0;
  }
else
  {
    spatial_step_size(i, M, spoint, j, N, vpoint,
                      &Dsi, &Dsip1,
                      &Dvjm1, &Dvj, &Dvjp1, &Dvjp2);


    if (i == 1)
      {
        if (stencil == 1)
          {
            // S=Smin -> Dirichlet boundary condition
            *pvalue_scmb = Dirichlet_Smin(spoint, i - 1, vpoint, j, r, divid,
            *pvalue_coeff = 0.; // No change in the matrix.
            return 0;
          }
        if (stencil == 7)
          {
            // S=Smin -> Dirichlet boundary condition
            *pvalue_scmb = Dirichlet_Smin(spoint, i - 1, vpoint, j - 1,  r, di
            *pvalue_coeff = 0.; // No change in the matrix.
            return 0;
          }
        if (stencil == 9)
          {
            // S=Smin -> Dirichlet boundary condition
            *pvalue_scmb = Dirichlet_Smin(spoint, i - 1, vpoint, j + 1,  r, di
            *pvalue_coeff = 0.; // No change in the matrix.
            return 0;
            // Remark that if i==1 but j==N-1 there is a double Dirichlet
            // condition. So the necessary consistant condition:
            // Dirichlet(0,~N) = Dirichlet(~0,N)
```

```
        // where we say that the Dirichlet condition at point
        // S=Smin with V anything (but at V=Vmax)
        // should coincide with the Dirichlet condition at point
        // V=Vmax with S anything (but at S=Smin).
      }
  }

if (i == M)
  {
    if (stencil == 2)
      {
        // S=Smax -> Neumann boundary condition
        //std::cout << "Neumann_Smax=" << Neumann_Smax(spoint, i+1, vpoint
        *pvalue_scmb = Neumann_Smax(spoint, i + 1, vpoint, j,  r, divid, s
                       * (Dsip1);
        *pvalue_coeff = 1.; // Change in the matrix.
        //std::cout << "pvalue_scmb=" << *pvalue_scmb << std::endl;
        return 0; // Diagonal coefficient.
      }
    if (stencil == 8)
      {
        // S=Smax -> Neumann boundary condition
        // But in this nonparallel direction, the point u(-1,+1)
        // is interpolated by u(-1,0)+ Dsp1 N(-1)
        // So it does not modified the diagonal coefficient
        // but it modifies the (-1,0) coefficient (stencil 4)
        *pvalue_scmb = Neumann_Smax(spoint, i - 1, vpoint, j - 1,  r, divi
                       * (Dsip1);
        *pvalue_coeff = 1.; // Change in the matrix.
        return 4;
      }
    if (stencil == 10)
      {
        // S=Smax -> Neumann boundary condition
        // But in this nonparallel direction, the point u(+1,+1)
        // is interpolated by u(+1,0)+ Dsp1 N(+1)
        // So it does not modified the diagonal coefficient
        // but it modifies the (+1,0) coefficient (stencil 5)
        *pvalue_scmb = Neumann_Smax(spoint, i + 1, vpoint, j + 1,  r, divi
                       * (Dsip1);
        *pvalue_coeff = 1.; // Change in the matrix.
```

```
        return 5;
        // Remark that if i==M but j==N-1 we should used a Dirichlet
        // condition at point j=N. So the necessary consistant condition:
        // U(M+1,N) = Dirichlet(M+1,N) = U(M,N) + Dsp1(M) Neumann(M+1)
        // = Dirichlet(M,N) + Dsp1(M) Neumann(M+1)
        // Thus (Dirichlet(M+1,N)-Dirichlet(M,N))/Dsp1(M) = Neumann(M+1)
      }
  }

if (j == 0)
  {
    // The stencil 3 is never used at point j==0 since we
    // use the central or forward scheme.
    // However the stencils 4, 7 && 8 are used by the mixted derivative.
    // But the value of coefficient in the PDE vanishes, so all
    // the coefficients in the matrix vanish && we do not care about.

  }

if (j == 1)
  {
    // The stencil 3 is never used at point j==1 since we
    // use the central scheme.
    // The other stencils are valid points except 1, 7, 9, 2, 8 && 10
    // when i==1 or i==M but these cases are already treated.
  }

if (j == N - 2)
  {
    // The stencil 6 is never used at point j==N-2 since we
    // use the backward scheme.
    // The other stencils are valid points except 1, 7, 9, 2, 8 && 10
    // when i==1 or i==M but these cases are already treated.
  }

if (j == N - 1)
  {
    // The stencil 6 is never used since we use the backward scheme.
    if (stencil == 5)
      {
        // V=Vmax -> Dirichlet boundary condition
```

```
              *pvalue_scmb = Dirichlet_Vmax(spoint, i, vpoint, j + 1,  r, divid,
              *pvalue_coeff = 0.; // No change in the matrix.
              return 0; // Diagonal coefficient.
            }
          if (stencil == 9)
            {
              // V=Vmax -> Dirichlet boundary condition
              *pvalue_scmb = Dirichlet_Vmax(spoint, i - 1, vpoint, j + 1,  r, di
              *pvalue_coeff = 0.; // No change in the matrix.
              return 0;
              // Remark that if i==N-1 but i==1 there is a double Dirichlet
              // condition. So the necessary consistant condition:
              // Dirichlet(0,~N) = Dirichlet(~0,N)
              // where we say that the Dirichlet condition at point
              // S=Smin with V anything (but at V=Vmax)
              // should coincide with the Dirichlet condition at point
              // V=Vmax with S anything (but at S=Smin).
            }
          if (stencil == 10)
            {
              // V=Vmax -> Dirichlet boundary condition
              *pvalue_scmb = Dirichlet_Vmax(spoint, i + 1, vpoint, j + 1,  r, di
              *pvalue_coeff = 0.; // No change in the matrix.
              return 0;
              // Remark that if j==N-1 but i==M we should used a Dirichlet
              // condition at point j=N. So the necessary consistant condition:
              // U(M+1,N) = Dirichlet(M+1,N) = U(M,N) + Dsp1(M) Neumann(M+1)
              // = Dirichlet(M,N) + Dsp1(M) Neumann(M+1)
              // Thus (Dirichlet(M+1,N)-Dirichlet(M,N))/Dsp1(M) = Neumann(M+1)
            }
        }
    }
  return -1;
}

static void build_matrix_extra_large
(double r, double divid, double strike, double time,
 double alpha, double beta, double gamma, double rho,
 int M, double *spoint, int N, double *vpoint,
 double ***MatrixA0, double ***MatrixA1, double ***MatrixA2)
{
```

```
// We use i from 1 to M && j from 0 to N-1
// We use points i-1 -> i+1 && j-2 -> j+2

// STENCIL
//    *                  6
//  * * * 9 5 10
//  * * *     <=>1 0 2
//  * * * 7 4 8
//    *                  3

double actualspoint;
double actualvpoint;

double Dsi, Dsip1;
double Dvjm1, Dvj, Dvjp1, Dvjp2;

double convection_s, diffusion_s;
double convection_v, diffusion_v;
double order_0, mixted_sv;

double *cs;
double *ds;
double *cv;
double *dv;
double *mx;

// Coefficients
//    double salpha_im2, salpha_im1, salpha_i0;
double sbeta_im1, sbeta_i0, sbeta_ip1;
//    double sgamma_i0, sgamma_ip1, sgamma_ip2;
//    double sdelta_im1, sdelta_i0, sdelta_ip1;

//    double valpha_jm2, valpha_jm1, valpha_j0;
double vbeta_jm1, vbeta_j0, vbeta_jp1;
//    double vgamma_j0, vgamma_jp1, vgamma_jp2;
//    double vdelta_jm1, vdelta_j0, vdelta_jp1;

int i, j, st;

cs = (double *)malloc(11 * sizeof(double));
```

```
ds = (double *)malloc(11 * sizeof(double));
cv = (double *)malloc(11 * sizeof(double));
dv = (double *)malloc(11 * sizeof(double));
mx = (double *)malloc(11 * sizeof(double));


for (j = 0; j <= N - 1; j++)
  {
    for (i = 1; i <= M; i++)
      {
        for (st = 0; st <= 10; st++)
          {
            cs[st] = 0.;
            ds[st] = 0.;
            cv[st] = 0.;
            dv[st] = 0.;
            mx[st] = 0.;
          }

        actualspoint = spoint[i];
        actualvpoint = vpoint[j];

        convection_s = (r - divid) * actualspoint;
        diffusion_s = actualspoint * actualspoint * actualvpoint / 2.0;
        convection_v = alpha * (beta - actualvpoint);
        diffusion_v = gamma * gamma * actualvpoint / 2.0;
        order_0 = -r;
        mixted_sv = rho * gamma * actualspoint * actualvpoint;


        spatial_step_size(i, M, spoint, j, N, vpoint,
                          &Dsi, &Dsip1,
                          &Dvjm1, &Dvj, &Dvjp1, &Dvjp2);

        // Diffusion S
        {
          ds[0] = -2.0 / (Dsi * Dsip1);
          ds[1] = 2.0 / (Dsi * (Dsi + Dsip1));
          ds[2] = 2.0 / (Dsip1 * (Dsi + Dsip1));
        }
        // Diffusion V
```

```
{
  dv[0] = -2.0 / (Dvj * Dvjp1);
  dv[4] = 2.0 / (Dvj * (Dvj + Dvjp1));
  dv[5] = 2.0 / (Dvjp1 * (Dvj + Dvjp1));
}
// Mixted SV
{
  sbeta_im1 = -Dsip1 / (Dsi * (Dsi + Dsip1));
  sbeta_i0 = (Dsip1 - Dsi) / (Dsi * Dsip1);
  sbeta_ip1 = Dsi / (Dsip1 * (Dsi + Dsip1));
  vbeta_jm1 = -Dvjp1 / (Dvj * (Dvj + Dvjp1));
  vbeta_j0 = (Dvjp1 - Dvj) / (Dvj * Dvjp1);
  vbeta_jp1 = Dvj / (Dvjp1 * (Dvj + Dvjp1));

  mx[0] = sbeta_i0 * vbeta_j0;
  mx[1] = sbeta_im1 * vbeta_j0;
  mx[2] = sbeta_ip1 * vbeta_j0;
  mx[4] = sbeta_i0 * vbeta_jm1;
  mx[5] = sbeta_i0 * vbeta_jp1;
  mx[7] = sbeta_im1 * vbeta_jm1;
  mx[8] = sbeta_ip1 * vbeta_jm1;
  mx[9] = sbeta_im1 * vbeta_jp1;
  mx[10] = sbeta_ip1 * vbeta_jp1;
}
// Convection S
{
  cs[0] = (Dsip1 - Dsi) / (Dsi * Dsip1);
  cs[1] = -Dsip1 / (Dsi * (Dsi + Dsip1));
  cs[2] = Dsi / (Dsip1 * (Dsi + Dsip1));
}
// Convection V
{
  if (j == 0) // V=Vmin -> Forward
    {
      cv[0] = -(2.0 * Dvjp1 + Dvjp2) / (Dvjp1 * (Dvjp1 + Dvjp2));
      cv[5] = (Dvjp1 + Dvjp2) / (Dvjp1 * Dvjp2);
      cv[6] = -Dvjp1 / (Dvjp2 * (Dvjp1 + Dvjp2));
    }
  if (j == 1) // Central scheme
    {
      cv[0] = (Dvjp1 - Dvj) / (Dvj * Dvjp1);
```

```
            cv[4] = -Dvjp1 / (Dvj * (Dvj + Dvjp1));
            cv[5] = Dvj / (Dvjp1 * (Dvj + Dvjp1));
          }
        if (j == N - 1) // Backward scheme
          {
            cv[0] = (Dvjm1 + 2.0 * Dvj) / (Dvj * (Dvjm1 + Dvj));
            cv[4] = -(Dvjm1 + Dvj) / (Dvjm1 * Dvj);
            cv[3] = Dvj / (Dvjm1 * (Dvjm1 + Dvj));
          }
        if ((j > 1) && (j < N - 1)) // Central or Backward scheme
          {
            if (convection_v < 0) // vol>beta -> Backward
              {
                cv[0] = (Dvjm1 + 2.0 * Dvj) / (Dvj * (Dvjm1 + Dvj));
                cv[4] = -(Dvjm1 + Dvj) / (Dvjm1 * Dvj);
                cv[3] = Dvj / (Dvjm1 * (Dvjm1 + Dvj));
              }
            else // vol<= beta -> Central
              {
                cv[0] = (Dvjp1 - Dvj) / (Dvj * Dvjp1);
                cv[4] = -Dvjp1 / (Dvj * (Dvj + Dvjp1));
                cv[5] = Dvj / (Dvjp1 * (Dvj + Dvjp1));
              }
          }
      }

      for (st = 0; st <= 10; st++) // 11 points in the stencil
        {
          MatrixA0[i][j][st] = mixted_sv * mx[st];
          MatrixA1[i][j][st] = convection_s * cs[st] + diffusion_s * ds[st];
          MatrixA2[i][j][st] = convection_v * cv[st] + diffusion_v * dv[st];
        }
      // Central point with order_0
      MatrixA1[i][j][0] += order_0 / 2.;
      MatrixA2[i][j][0] += order_0 / 2.;

    }
  }

free(mx);
free(dv);
```

```
  free(cv);
  free(ds);
  free(cs);
}


static void change_matrix_for_boundary_condition
(double r, double divid, double strike, double time,
 double alpha, double beta, double gamma, double rho, int call_or_put,
 int M, double *spoint, int N, double *vpoint,
 double ***MatrixA0, double ***MatrixA1, double ***MatrixA2,
 double **G0, double **G1, double **G2)
{

  // We use i from 1 to M && j from 0 to N-1
  // We use points i-1 -> i+1 && j-2 -> j+2

  // STENCIL
  //    *                  6
  //  * * * 9 5 10
  //  * * *     <=>1 0 2
  //  * * * 7 4 8
  //    *                  3


  double value_scmb, value_coeff;
  int stencil_changed;
  int i, j, st;

  double *MatrixA0tmp;
  double *MatrixA1tmp;
  double *MatrixA2tmp;

  // Memory Allocation
  MatrixA0tmp = (double *) malloc(11 * sizeof(double));
  MatrixA1tmp = (double *) malloc(11 * sizeof(double));
  MatrixA2tmp = (double *) malloc(11 * sizeof(double));

  for (j = 0; j < N + 1; j++)
    {
      for (i = 0; i < M + 1; i++)
```

```
      {
        G0[i][j] = 0.;
        G1[i][j] = 0.;
        G2[i][j] = 0.;
      }
  }

for (j = 0; j <= N - 1; j++)
  {
    for (i = 1; i <= M; i++)
      {
        // Save the matrix in temporary contener.
        for (st = 0; st < 11; st++)
          {
            MatrixA0tmp[st] = MatrixA0[i][j][st];
            MatrixA1tmp[st] = MatrixA1[i][j][st];
            MatrixA2tmp[st] = MatrixA2[i][j][st];
          }
        for (st = 0; st < 11; st++)
          {
            if (it_exists_stencil(i, M, j, N, st)) // The point is valid
              {
                // There is no change in the matrix
              }
            else // The point is not valid. There is a boundary condition
              {
                stencil_changed =
                  value_of_boundary_condition(i, M, spoint,
                                              j, N, vpoint,
                                              r, divid, strike, time,
                                              st, call_or_put,
                                              &value_scmb, &value_coeff);

                if (stencil_changed >= 0) // The stencil_changed is concerned.
                  {
                    {
                      G0[i][j] += MatrixA0[i][j][st] * value_scmb;
                      MatrixA0tmp[stencil_changed] += value_coeff * MatrixA0[i

                      G1[i][j] += MatrixA1[i][j][st] * value_scmb;
                      MatrixA1tmp[stencil_changed] += value_coeff * MatrixA1[i
```

```
                        G2[i][j] += MatrixA2[i][j][st] * value_scmb;
                        MatrixA2tmp[stencil_changed] += value_coeff * MatrixA2[i
                      }
                  }
                else
                  {
                    // The point is not valid but not used by the scheme or us
                  }


              }
          }
        // Update the matrix from temporary contener.
        for (st = 0; st < 11; st++)
          {
            MatrixA0[i][j][st] = MatrixA0tmp[st];
            MatrixA1[i][j][st] = MatrixA1tmp[st];
            MatrixA2[i][j][st] = MatrixA2tmp[st];
          }

      }
  }

  // Memory desallocation
  free(MatrixA0tmp);
  free(MatrixA1tmp);
  free(MatrixA2tmp);
}

static void computation_explicit_syslin_all_matrix(double coeff,
    int M, double *spoint, int N, double *vpoint,
    double ***MatrixA0, double ***MatrixA1, double ***MatrixA2,
    double **G0nm1, double **G1nm1, double **G2nm1,
    double **Unm1, double **Y0)
{
  int i, j, st;
  double val = 0;
  int istencil, jstencil;

  for (i = 1; i < M + 1; i++)
```

```
    {
      for (j = 0; j < N; j++)
        {
          val = Unm1[i][j];
          for (st = 0; st < 11; st++)
            {
              if (it_exists_stencil(i, M, j, N, st))
                {
                  // If the point exists.
                  point_of_stencil(i, j, st, &istencil, &jstencil);
                  val += coeff * MatrixA0[i][j][st] * Unm1[istencil][jstencil];
                  val += coeff * MatrixA1[i][j][st] * Unm1[istencil][jstencil];
                  val += coeff * MatrixA2[i][j][st] * Unm1[istencil][jstencil];
                }
            }
          val += coeff * G0nm1[i][j];
          val += coeff * G1nm1[i][j];
          val += coeff * G2nm1[i][j];
          Y0[i][j] = val;
        }
    }
}

static void computation_explicit_syslin_spot_matrix(double coeff,
    int M, double *spoint, int N, double *vpoint,
    double ***MatrixA0, double ***MatrixA1, double ***MatrixA2,
    double **G0nm1, double **G1nm1, double **G2nm1,
    double **Unm1, double **Y0, double **Sortie)
{
  int i, j, st;
  double val = 0;
  int istencil, jstencil;

  for (j = 0; j < N; j++)
    {
      for (i = 1; i < M + 1; i++)
        {
          val = Y0[i][j];
          for (st = 0; st < 11; st++)
            {
              if (it_exists_stencil(i, M, j, N, st))
```

```
                {
                  // If the point exists.
                  point_of_stencil(i, j, st, &istencil, &jstencil);
                  val += -coeff * MatrixA1[i][j][st] * Unm1[istencil][jstencil];
                }
            }
          val += -coeff * G1nm1[i][j];
          Sortie[i][j] = val;

        }
    }
}

static void computation_implicit_syslin_spot_matrix(double coeff,
    int M, double *spoint, int N, double *vpoint,
    double ***MatrixA0, double ***MatrixA1, double ***MatrixA2,
    double **G0, double **G1, double **G2,
    double **rhs, double **lhs)
{
  int i, j;

  // Only points from i=1 to i=M.
  // Only points from j=0 to j=N-1.
  PnlTridiagMat *Identity_Matrix;
  PnlTridiagMat *Working_Matrix;
  PnlVect *Entree;
  PnlVect *Sortie;

  Identity_Matrix = pnl_tridiag_mat_create_from_two_double(M, 1.0, 0.0); // Iden
  Working_Matrix = pnl_tridiag_mat_create(M);
  Entree = pnl_vect_create(M);
  Sortie = pnl_vect_create(M);

  for (j = 0; j < N; j++)
    {
      // Build the matrix

      //pnl_tridiag_mat_set (Working_Matrix,0, -1, MatrixA1[1][j][1]);
      pnl_tridiag_mat_set(Working_Matrix, 0, 0, MatrixA1[1][j][0]);
      pnl_tridiag_mat_set(Working_Matrix, 0, 1, MatrixA1[1][j][2]);
      for (i = 1; i < M - 1; i++)
```

```
        {
          pnl_tridiag_mat_set(Working_Matrix, i, -1, MatrixA1[i + 1][j][1]);
          pnl_tridiag_mat_set(Working_Matrix, i, 0, MatrixA1[i + 1][j][0]);
          pnl_tridiag_mat_set(Working_Matrix, i, 1, MatrixA1[i + 1][j][2]);
        }
      pnl_tridiag_mat_set(Working_Matrix, M - 1, 0, MatrixA1[M][j][0]);
      pnl_tridiag_mat_set(Working_Matrix, M - 1, -1, MatrixA1[M][j][1]);
      //pnl_tridiag_mat_set (Working_Matrix,M-1, 1, MatrixA1[M][j][2]);
      // Multiplication by -coeff.
      pnl_tridiag_mat_mult_double(Working_Matrix, -coeff);
      // Sum with the identity matrix. Result is in the first argument.
      pnl_tridiag_mat_plus_tridiag_mat(Working_Matrix, Identity_Matrix);

      // Build the vectors.
      for (i = 0; i < M; i++)
        {
          pnl_vect_set(Entree, i, rhs[i + 1][j] + coeff * G1[i + 1][j]);
        }

      pnl_tridiag_mat_syslin(Sortie, Working_Matrix, Entree);

      for (i = 0; i < M; i++)
        {
          lhs[i + 1][j] = pnl_vect_get(Sortie, i);
        }

    }

  pnl_vect_free(&Sortie);
  pnl_vect_free(&Entree);
  pnl_tridiag_mat_free(&Working_Matrix);
  pnl_tridiag_mat_free(&Identity_Matrix);
}

static void computation_explicit_syslin_var_matrix(double coeff,
    int M, double *spoint, int N, double *vpoint,
    double ***MatrixA0, double ***MatrixA1, double ***MatrixA2,
    double **G0nm1, double **G1nm1, double **G2nm1,
    double **Unm1, double **Y1, double **Sortie)
{
  int i, j, st;
```

```
  double val = 0;
  int istencil, jstencil;

  for (i = 1; i < M + 1; i++)
    {
      for (j = 0; j < N; j++)
        {
          val = Y1[i][j];
          for (st = 0; st < 11; st++)
            {
              if (it_exists_stencil(i, M, j, N, st))
                {
                  // If the point exists.
                  point_of_stencil(i, j, st, &istencil, &jstencil);
                  val += -coeff * MatrixA2[i][j][st] * Unm1[istencil][jstencil];
                }
            }
          val += -coeff * G2nm1[i][j];
          Sortie[i][j] = val;
        }
    }
}

static void computation_implicit_syslin_var_matrix(double coeff,
    int M, double *spoint, int N, double *vpoint,
    double ***MatrixA0, double ***MatrixA1, double ***MatrixA2,
    double **G0, double **G1, double **G2,
    double **rhs, double **lhs)
{
  int i, j;

  // Only points from i=1 to i=M.
  // Only points from j=0 to j=N-1.
  // Pentadiagonal matrix.
  PnlBandMat *Working_Matrix;
  PnlVect *Entree;
  PnlVect *Sortie;

  Entree = pnl_vect_create(N);
  Sortie = pnl_vect_create(N);
```

```
for (i = 1; i < M + 1; i++)
  {
    Working_Matrix = pnl_band_mat_create(N, N, 2, 2);
    // Build the matrix

    //pnl_band_mat_set (Working_Matrix, 0, 0-2, MatrixA2[i][0][3]);
    //pnl_band_mat_set (Working_Matrix, 0, 0-1, MatrixA2[i][0][4]);
    pnl_band_mat_set(Working_Matrix, 0, 0 + 0, MatrixA2[i][0][0]);
    pnl_band_mat_set(Working_Matrix, 0, 0 + 1, MatrixA2[i][0][5]);
    pnl_band_mat_set(Working_Matrix, 0, 0 + 2, MatrixA2[i][0][6]);
    //pnl_band_mat_set (Working_Matrix, 1, 1-2, MatrixA2[i][1][3]);
    pnl_band_mat_set(Working_Matrix, 1, 1 - 1, MatrixA2[i][1][4]);
    pnl_band_mat_set(Working_Matrix, 1, 1 + 0, MatrixA2[i][1][0]);
    pnl_band_mat_set(Working_Matrix, 1, 1 + 1, MatrixA2[i][1][5]);
    pnl_band_mat_set(Working_Matrix, 1, 1 + 2, MatrixA2[i][1][6]);
    for (j = 2; j < N - 2; j++)
      {
        pnl_band_mat_set(Working_Matrix, j, j - 2, MatrixA2[i][j][3]);
        pnl_band_mat_set(Working_Matrix, j, j - 1, MatrixA2[i][j][4]);
        pnl_band_mat_set(Working_Matrix, j, j + 0, MatrixA2[i][j][0]);
        pnl_band_mat_set(Working_Matrix, j, j + 1, MatrixA2[i][j][5]);
        pnl_band_mat_set(Working_Matrix, j, j + 2, MatrixA2[i][j][6]);
      }
    pnl_band_mat_set(Working_Matrix, N - 2, N - 2 - 2, MatrixA2[i][N - 2][3]);
    pnl_band_mat_set(Working_Matrix, N - 2, N - 2 - 1, MatrixA2[i][N - 2][4]);
    pnl_band_mat_set(Working_Matrix, N - 2, N - 2 + 0, MatrixA2[i][N - 2][0]);
    pnl_band_mat_set(Working_Matrix, N - 2, N - 2 + 1, MatrixA2[i][N - 2][5]);
    //pnl_band_mat_set (Working_Matrix, N-2, N-2+2, MatrixA2[i][N-2][6]);
    pnl_band_mat_set(Working_Matrix, N - 1, N - 1 - 2, MatrixA2[i][N - 1][3]);
    pnl_band_mat_set(Working_Matrix, N - 1, N - 1 - 1, MatrixA2[i][N - 1][4]);
    pnl_band_mat_set(Working_Matrix, N - 1, N - 1 + 0, MatrixA2[i][N - 1][0]);
    //pnl_band_mat_set (Working_Matrix, N-1, N-1+1, MatrixA2[i][N-1][5]);
    //pnl_band_mat_set (Working_Matrix, N-1, N-1+2, MatrixA2[i][N-1][6]);

    // Multiplication by -coeff.
    pnl_band_mat_mult_double(Working_Matrix, -coeff);
    // Sum with the identity matrix.
    for (j = 0; j < N; j++)
      {
        pnl_band_mat_set(Working_Matrix, j, j, 1. + pnl_band_mat_get(Working_M
```

```
        }

      // Build the vectors.
      for (j = 0; j < N; j++)
        {
          pnl_vect_set(Entree, j, rhs[i][j] + coeff * G2[i][j]);
        }

      pnl_band_mat_syslin(Sortie, Working_Matrix, Entree);

      for (j = 0; j < N; j++)
        {
          lhs[i][j] = pnl_vect_get(Sortie, j);
        }
      pnl_band_mat_free(&Working_Matrix);
    }

  pnl_vect_free(&Sortie);
  pnl_vect_free(&Entree);
}


static int AdiHoutFoulon
(double S, double V,
 double t, double r, double divid, double alpha, double beta, double gamma, doub
 double Smax, double Vmax, int M, int N, int L,
 double theta, int computation_lu, int scheme, int call_or_put, int am,
 double *ptprice, double *ptdelta)
{

  int TimeIndex, i, j, st;
  int IndexS, IndexV;
  double valxdw;
  double valxup;

  double deltat;

  // 1-vectors
  double *spoint;
  double *vpoint;
```

```
// 2-vectors
double **Un;
double **Utmp;
double **Y0;
double **Y1;
//    double **Y2;
double **G0nm1;
double **G1nm1;
double **G2nm1;
double **G0n;
double **G1n;
double **G2n;

// Matrix (=3-vectors)
double ***MatrixA0;
double ***MatrixA1;
double ***MatrixA2;



// Memory allocation of 1-vectors.
spoint = (double *)malloc((M + 1) * sizeof(double));
vpoint = (double *)malloc((N + 1) * sizeof(double));

grid_generation_HF_spot(spoint, Smax, strike, M, strike / 5.0);
grid_generation_HF_variance(vpoint, Vmax, N, Vmax / 500.0);

// Memory allocation of 2-vectors.
Un = (double **) malloc((M + 1) * sizeof(double *));
Utmp = (double **) malloc((M + 1) * sizeof(double *));
Y0 = (double **) malloc((M + 1) * sizeof(double *));
Y1 = (double **) malloc((M + 1) * sizeof(double *));
//    Y2 = (double**) malloc((M+1) * sizeof(double*));
G0nm1 = (double **) malloc((M + 1) * sizeof(double *));
G1nm1 = (double **) malloc((M + 1) * sizeof(double *));
G2nm1 = (double **) malloc((M + 1) * sizeof(double *));
G0n = (double **) malloc((M + 1) * sizeof(double *));
G1n = (double **) malloc((M + 1) * sizeof(double *));
G2n = (double **) malloc((M + 1) * sizeof(double *));
for (i = 0; i < M + 1; i++)
  {
```

```
    Un[i] = (double *) malloc((N + 1) * sizeof(double));
    Utmp[i] = (double *) malloc((N + 1) * sizeof(double));
    Y0[i] = (double *) malloc((N + 1) * sizeof(double));
    Y1[i] = (double *) malloc((N + 1) * sizeof(double));
    //        Y2[i] = (double*) malloc((N+1) * sizeof(double));
    G0nm1[i] = (double *) malloc((N + 1) * sizeof(double));
    G1nm1[i] = (double *) malloc((N + 1) * sizeof(double));
    G2nm1[i] = (double *) malloc((N + 1) * sizeof(double));
    G0n[i] = (double *) malloc((N + 1) * sizeof(double));
    G1n[i] = (double *) malloc((N + 1) * sizeof(double));
    G2n[i] = (double *) malloc((N + 1) * sizeof(double));
  }

// Memory allocation of matrix (=3-vectors).
MatrixA0 = (double ** *) malloc((M + 1) * sizeof(double *));
MatrixA1 = (double ** *) malloc((M + 1) * sizeof(double *));
MatrixA2 = (double ** *) malloc((M + 1) * sizeof(double *));
for (i = 0; i < M + 1; i++)
  {
    MatrixA0[i] = (double **) malloc((N + 1) * sizeof(double));
    MatrixA1[i] = (double **) malloc((N + 1) * sizeof(double));
    MatrixA2[i] = (double **) malloc((N + 1) * sizeof(double));
    for (j = 0; j < N + 1; j++)
      {
        MatrixA0[i][j] = (double *) malloc(11 * sizeof(double));
        MatrixA1[i][j] = (double *) malloc(11 * sizeof(double));
        MatrixA2[i][j] = (double *) malloc(11 * sizeof(double));
      }
  }

// Initialization

for (i = 0; i < M + 1; i++)
  {
    for (j = 0; j < N + 1; j++)
      {
        Un[i][j] = MAX(call_or_put * (spoint[i] - strike), 0.0);
        Utmp[i][j] = MAX(call_or_put * (spoint[i] - strike), 0.0);
        Y0[i][j] = 0.;
        Y1[i][j] = 0.;
        //Y2[i][j] = 0.;
```

```
        G0nm1[i][j] = 0.;
        G1nm1[i][j] = 0.;
        G2nm1[i][j] = 0.;
        G0n[i][j] = 0.;
        G1n[i][j] = 0.;
        G2n[i][j] = 0.;
      }
  }

for (st = 0; st < 11; st++)
  {
    for (j = 0; j < N + 1; j++)
      {
        for (i = 0; i < M + 1; i++)
          {
            MatrixA0[i][j][st] = 0.;
            MatrixA1[i][j][st] = 0.;
            MatrixA2[i][j][st] = 0.;
          }
      }
  }

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

// Build the matrix for the first iteration.
TimeIndex = 0;
build_matrix_extra_large(r, divid, strike, TimeIndex * deltat, alpha, beta, ga
                         M, spoint, N, vpoint, MatrixA0, MatrixA1, MatrixA2);

change_matrix_for_boundary_condition(r, divid, strike, TimeIndex * deltat, alp
                                     M, spoint, N, vpoint, MatrixA0, MatrixA1,
                                     G0nm1, G1nm1, G2nm1);




/*
 // First iteration of finite difference cycle with deltat/2 && theta=1.
 {
 // Douglas Scheme with theta = 1
```

```
TimeIndex = 0;
deltat = deltat/2.;

// Compute the elements Gnm1.
build_matrix_extra_large(r, divid, TimeIndex*deltat, alpha, beta, gamma, rho,
M, spoint, N, vpoint, MatrixA0, MatrixA1, MatrixA2);
change_matrix_for_boundary_condition(r, divid, TimeIndex*deltat, alpha, beta,
M, spoint, N, vpoint, MatrixA0, MatrixA1, MatrixA2,
G0nm1, G1nm1, G2nm1);


// Y0 = U[n-1] + Dt (A0+A1+A2)[n-1] * U[n-1] + Dt (G0+G1+G2)[n-1]
computation_explicit_syslin_all_matrix(deltat, M, spoint, N, vpoint,
MatrixA0, MatrixA1, MatrixA2,
G0nm1, G1nm1, G2nm1, Un, Y0);

// Utmp = Y0 - theta*Dt A1[n-1] * U[n-1] - theta*Dt G1[n-1]
computation_explicit_syslin_spot_matrix(1. * deltat, M, spoint, N, vpoint,
MatrixA0, MatrixA1, MatrixA2,
G0nm1, G1nm1, G2nm1,
Un, Y0, Utmp);

// Compute the new elements Gn.
build_matrix_extra_large(r, divid, (TimeIndex+1)*deltat, alpha, beta, gamma,
M, spoint, N, vpoint, MatrixA0, MatrixA1, MatrixA2);
change_matrix_for_boundary_condition(r, divid, (TimeIndex+1)*deltat, alpha, b
M, spoint, N, vpoint, MatrixA0, MatrixA1, MatrixA2,
G0n, G1n, G2n);
// Y1 = (Id - theta*Dt A1[n])^-1 ( Utmp + theta*Dt G1[n] )
computation_implicit_syslin_spot_matrix(1. * deltat, M, spoint,  N, vpoint,
MatrixA0, MatrixA1, MatrixA2,
G0n, G1n, G2n, Utmp, Y1);

// Utmp = Y1 - theta*Dt A2[n-1] * U[n-1] - theta*Dt G2[n-1]
computation_explicit_syslin_var_matrix(1. * deltat, M, spoint, N, vpoint,
MatrixA0, MatrixA1, MatrixA2,
G0nm1, G1nm1, G2nm1,
Un, Y1, Utmp);

// Y2 = (Id - theta*Dt A2[n])^-1 ( Utmp + theta*Dt G2[n] )
computation_implicit_syslin_var_matrix(1. * deltat, M, spoint,  N, vpoint,
```

```
MatrixA0, MatrixA1, MatrixA2,
G0n, G1n, G2n, Utmp, Un); // /!\  with Y2=Un


}
// Second iteration of finite difference cycle again with deltat/2 && theta=1
{
// Douglas Scheme with theta = 1
TimeIndex = 1;
//deltat = deltat/2.;

// Compute the elements Gnm1.
build_matrix_extra_large(r, divid, TimeIndex*deltat, alpha, beta, gamma, rho,
M, spoint, N, vpoint, MatrixA0, MatrixA1, MatrixA2);
change_matrix_for_boundary_condition(r, divid, TimeIndex*deltat, alpha, beta,
M, spoint, N, vpoint, MatrixA0, MatrixA1, MatrixA2,
G0nm1, G1nm1, G2nm1);


// Y0 = U[n-1] + Dt (A0+A1+A2)[n-1] * U[n-1] + Dt (G0+G1+G2)[n-1]
computation_explicit_syslin_all_matrix(deltat, M, spoint, N, vpoint,
MatrixA0, MatrixA1, MatrixA2,
G0nm1, G1nm1, G2nm1, Un, Y0);

// Utmp = Y0 - theta*Dt A1[n-1] * U[n-1] - theta*Dt G1[n-1]
computation_explicit_syslin_spot_matrix(1. * deltat, M, spoint, N, vpoint,
MatrixA0, MatrixA1, MatrixA2,
G0nm1, G1nm1, G2nm1,
Un, Y0, Utmp);

// Compute the new elements Gn.
build_matrix_extra_large(r, divid, (TimeIndex+1)*deltat, alpha, beta, gamma,
M, spoint, N, vpoint, MatrixA0, MatrixA1, MatrixA2);
change_matrix_for_boundary_condition(r, divid, (TimeIndex+1)*deltat, alpha, b
M, spoint, N, vpoint, MatrixA0, MatrixA1, MatrixA2,
G0n, G1n, G2n);
// Y1 = (Id - theta*Dt A1[n])^-1 ( Utmp + theta*Dt G1[n] )
computation_implicit_syslin_spot_matrix(1. * deltat, M, spoint,  N, vpoint,
MatrixA0, MatrixA1, MatrixA2,
G0n, G1n, G2n, Utmp, Y1);
```

```
 // Utmp = Y1 - theta*Dt A2[n-1] * U[n-1] - theta*Dt G2[n-1]
 computation_explicit_syslin_var_matrix(1. * deltat, M, spoint, N, vpoint,
 MatrixA0, MatrixA1, MatrixA2,
 G0nm1, G1nm1, G2nm1,
 Un, Y1, Utmp);

 // Y2 = (Id - theta*Dt A2[n])^-1 ( Utmp + theta*Dt G2[n] )
 computation_implicit_syslin_var_matrix(1. * deltat, M, spoint,  N, vpoint,
 MatrixA0, MatrixA1, MatrixA2,
 G0n, G1n, G2n, Utmp, Un); // /!\  with Y2=Un

 }
 */
// Other iterations of finite difference cycle starting at TimeIndex = 1
//deltat=2.* deltat;

// Finite Difference Cycle.
for (TimeIndex = 0; TimeIndex < L; TimeIndex++)
  {
    if (am == 1)
      {
        // American condition = Unknowns >= payoff
        for (j = 0; j < N + 1; j++)
          {
            for (i = 0; i < M + 1; i++)
              {
                if (call_or_put == 1) //Call
                  {
                    Un[i][j] = MAX(Un[i][j], MAX(spoint[i] - strike, 0.0));
                  }
                if (call_or_put == -1) //Put
                  {
                    Un[i][j] = MAX(Un[i][j], MAX(strike - spoint[i], 0.0));
                  }
              }
          }
      }

    if (scheme == 0) // Douglas scheme
      {
        // Compute the elements Gnm1.
```

```
build_matrix_extra_large(r, divid, strike, TimeIndex * deltat,
                         alpha, beta, gamma, rho,
                         M, spoint, N, vpoint,
                         MatrixA0, MatrixA1, MatrixA2);
change_matrix_for_boundary_condition(r, divid, strike, TimeIndex * del
                                     alpha, beta, gamma, rho, call_or_
                                     M, spoint, N, vpoint, MatrixA0, M
                                     G0nm1, G1nm1, G2nm1);

// Y0 = U[n-1] + Dt (A0+A1+A2)[n-1] * U[n-1] + Dt (G0+G1+G2)[n-1]
computation_explicit_syslin_all_matrix(deltat, M, spoint, N, vpoint,
                                       MatrixA0, MatrixA1, MatrixA2,
                                       G0nm1, G1nm1, G2nm1, Un, Y0);

// Utmp = Y0 - theta*Dt A1[n-1] * U[n-1] - theta*Dt G1[n-1]
computation_explicit_syslin_spot_matrix(theta * deltat, M, spoint, N,
                                        MatrixA0, MatrixA1, MatrixA2,
                                        G0nm1, G1nm1, G2nm1,
                                        Un, Y0, Utmp);

// Compute the new elements Gn.
build_matrix_extra_large(r, divid, strike, (TimeIndex + 1)*deltat,
                         alpha, beta, gamma, rho,
                         M, spoint, N, vpoint, MatrixA0, MatrixA1, Mat
change_matrix_for_boundary_condition(r, divid, strike, (TimeIndex + 1)
                                     alpha, beta, gamma, rho, call_or_
                                     M, spoint, N, vpoint, MatrixA0, M
                                     G0n, G1n, G2n);
// Y1 = (Id - theta*Dt A1[n])^-1 ( Utmp + theta*Dt G1[n] )
computation_implicit_syslin_spot_matrix(theta * deltat, M, spoint,  N,
                                        MatrixA0, MatrixA1, MatrixA2,
                                        G0n, G1n, G2n, Utmp, Y1);

// Utmp = Y1 - theta*Dt A2[n-1] * U[n-1] - theta*Dt G2[n-1]
computation_explicit_syslin_var_matrix(theta * deltat, M, spoint, N, v
                                       MatrixA0, MatrixA1, MatrixA2,
                                       G0nm1, G1nm1, G2nm1,
                                       Un, Y1, Utmp);

// Y2 = (Id - theta*Dt A2[n])^-1 ( Utmp + theta*Dt G2[n] )
computation_implicit_syslin_var_matrix(theta * deltat, M, spoint,  N,
```

```
                                                    MatrixA0, MatrixA1, MatrixA2,
                                                    G0n, G1n, G2n, Utmp, Un); // /!

      }
    if (scheme == 1) // Craig-Sneyd scheme
      {

      }
    if (scheme == 2) // Modified Craig-Sneyd scheme
      {

      }
    if (scheme == 3) // Hundsdorfer-Verwer scheme
      {

      }


  }
// Index in the domain for the price.
// Find the index in spoint corresponding to the price S of the asset.
IndexS = lower_index(spoint, M + 1, S);
// Find the index in [0,Vmax] corresponding to the variance V of the asset.
//IndexV = (int)(V/h);
IndexV = lower_index(vpoint, N + 1, V);

// Compute first the delta (using *ptprice as temporary variable). We do an in
valxdw = interpolation(spoint[IndexS], Un[IndexS][IndexV],
                       spoint[IndexS + 1], Un[IndexS + 1][IndexV], spoint[Inde
valxup = interpolation(spoint[IndexS], Un[IndexS][IndexV + 1],
                       spoint[IndexS + 1], Un[IndexS + 1][IndexV + 1], spoint[
*ptprice = interpolation(vpoint[IndexV], valxdw, vpoint[IndexV + 1], valxup, V
valxdw = interpolation(spoint[IndexS], Un[IndexS][IndexV],
                       spoint[IndexS + 1], Un[IndexS + 1][IndexV], spoint[Inde
valxup = interpolation(spoint[IndexS], Un[IndexS][IndexV + 1],
                       spoint[IndexS + 1], Un[IndexS + 1][IndexV + 1], spoint[
*ptdelta = (interpolation(vpoint[IndexV], valxdw, vpoint[IndexV + 1], valxup,
           / (spoint[IndexS + 1] - spoint[IndexS]);
// Next compute the price. We do an interpolation.
valxdw = interpolation(spoint[IndexS], Un[IndexS][IndexV],
                       spoint[IndexS + 1], Un[IndexS + 1][IndexV], S);
```

```
valxup = interpolation(spoint[IndexS], Un[IndexS][IndexV + 1],
                       spoint[IndexS + 1], Un[IndexS + 1][IndexV + 1], S);
*ptprice = interpolation(vpoint[IndexV], valxdw, vpoint[IndexV + 1], valxup, V


// Memory desallocation of matrix (=3-vectors)

for (i = 0; i < M + 1; i++)
  {
    for (j = 0; j < N + 1; j++)
      {
        free(MatrixA0[i][j]);
        free(MatrixA1[i][j]);
        free(MatrixA2[i][j]);
      }
    free(MatrixA0[i]);
    free(MatrixA1[i]);
    free(MatrixA2[i]);
  }
free(MatrixA0);
free(MatrixA1);
free(MatrixA2);

// Memory desallocation of 2-vectors
// Memory Allocation
for (i = 0; i < M + 1; i++)
  {
    free(Un[i]);
    free(Utmp[i]);
    free(Y0[i]);
    free(Y1[i]);
    //free(Y2[i]);
    free(G0nm1[i]);
    free(G1nm1[i]);
    free(G2nm1[i]);
    free(G0n[i]);
    free(G1n[i]);
    free(G2n[i]);
  }

free(Un);
```

```
  free(Utmp);
  free(Y0);
  free(Y1);
  //free(Y2);
  free(G0nm1);
  free(G1nm1);
  free(G2nm1);
  free(G0n);
  free(G1n);
  free(G2n);

  // Memory desallocation of 1-vectors
  free(spoint);
  free(vpoint);


  return 0;
}



static int FDHout_Heston(int am, double S0, double V0, NumFunc_1  *p, double T,
{
  double K;
  int call_or_put;
  double omega = 0.5;  // Between 0 && 1.
  double X = 8.0; //  S_max
  double Y = 5.0; // V_max
  //h = Y/N < gamma/rho (i.e. Y rho < gamma N) && rho*rho < (1-Y*rho/Ngamma)
  int computation_lu = 1;
  int scheme = 0;
  X = X * S0;
  \
  // Basic splitting method = 0 or Strang symmetrized splitting method = 1.

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

```
  AdiHoutFoulon(S0, V0, T, r, divid, kappa, theta, sigma, rho, K, X, Y, M, N, L,
                omega, computation_lu, scheme, call_or_put, am,
                ptprice, ptdelta);
  return OK;
}

int CALC(FD_Hout_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 FDHout_Heston(ptOpt->EuOrAm.Val.V_BOOL, ptMod->S0.Val.V_PDOUBLE,
                           ptMod->Sigma0.Val.V_PDOUBLE,
                           ptOpt->PayOff.Val.V_NUMFUNC_1,
                           ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
                           r,
                           divid, ptMod->Sigma.Val.V_PDOUBLE,
                           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_INT, Met->Pa
                           &(Met->Res[0].Val.V_DOUBLE),
                           &(Met->Res[1].Val.V_DOUBLE)
                          );
    }
}


static int CHK_OPT(FD_Hout_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_hout_heston";

      Met->Par[0].Val.V_INT2 = 100;
      Met->Par[1].Val.V_INT2 = 80;
      Met->Par[2].Val.V_INT2 = 40;
    }

  return OK;
}

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