Help

```
#include <stdlib.h>
#include "
href../../../mod/hes1d_slv/hes1d_slv_std/hes1d_slv_std_h_src.pdfhes1d_slv_std.h"
#include "pnl/pnl_mathtools.h"
#include "pnl/pnl_tridiag_matrix.h"
#include "pnl/pnl_band_matrix.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2019+2) //The "#els
static int CHK_OPT(FD_HOUT_LVSV)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(FD_HOUT_LVSV)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else

// This file contains all the code for the calibration of stochastic
// local volatility models under the Heston model.
// It uses a finite difference scheme to simulate the associated PDE.
// Moreover an ADI-scheme is used to accelerate the computation.
// The model is described in Wyns and Hout's paper.

//////////
// (almost-)Booleans to modify behavior of the program
//////////
// Almost-booleans : BE CAREFUL. MUST BE 1.0 TO AVOID INTEGER DIVISION !!!
#define SPDE 1.0
#define VPDE 1.0
// Really booleans
#define V0_IN_GRID 1
#define S0_IN_GRID 1
#define VERBOSE 0
#define OPTI 1

//////////
// Functions to generate grids.
//////////
```

```
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 int lower_index(double *grid, int size, double value)
{
    double value_nearest;
    int index_nearest;
    int i;

    value_nearest = ABS(grid[0]-value);
    index_nearest = -1;

    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 grid_generation_spot(double *sgrid, double Sleft, double Sright, dou
{
    int i;
    double Ximin;
```

```
    double Xiint;
    double Ximax;
    double deltaxi;

    Ximin = asinh1(-Sleft/Coeff_s);
    Xiint = (Sright-Sleft)/Coeff_s;
    Ximax = Xiint + asinh1((Smax-Sright)/Coeff_s);
    deltaxi = (Ximax-Ximin)/Ns;

    // Definition of uniform grid Xi.
    for (i=0; i<=Ns; i++)
    {
        sgrid[i] = Ximin + i * deltaxi;
    }

    // Definition of the spot grid with the uniform grid Xi.
    sgrid[0] = 0;
    for (i=1; i<=Ns; i++)
    {
        if (sgrid[i]<0)
        {
            sgrid[i] = Sleft+Coeff_s*sinh(sgrid[i]);
        }
        else
        {
            if (sgrid[i]<=Xiint)
            {
                sgrid[i] = Sleft+Coeff_s*sgrid[i];
            }
            else
            {
                sgrid[i] = Sright+Coeff_s*sinh(sgrid[i]-Xiint);
            }
        }
    }
}

static void grid_generation_variance(double *vgrid, double Vmax, int Nv, double
{
    int j;
    double deltaeta;
```

```
    deltaeta = (asinh1(Vmax/Coeff_v))/Nv;
    // Definition of uniform grid eta.
    for (j=0; j<=Nv; j++)
    {
        vgrid[j] = j * deltaeta;
    }
    // Definition of the volatility grid with the uniform grid eta.
    vgrid[0] = 0;
    for (j=1; j<=Nv; j++)
    {
        vgrid[j] = Coeff_v * sinh(vgrid[j]);
    }

    if (V0_IN_GRID)
    {
        j = lower_index(vgrid, Nv, Center_v);
        vgrid[j] = Center_v;
    }
}

//////////
// Functions to manage stencil and interpolate values.
//////////
static int stencil(int i, int j)
{
    if ((i== 0) && (j== 0)) return 0;
    if ((i==-1) && (j== 0)) return 1;
    if ((i== 1) && (j== 0)) return 2;
    if ((i== 0) && (j==-2)) return 3;
    if ((i== 0) && (j==-1)) return 4;
    if ((i== 0) && (j== 1)) return 5;
    if ((i== 0) && (j== 2)) return 6;
    if ((i==-1) && (j==-1)) return 7;
    if ((i== 1) && (j==-1)) return 8;
    if ((i==-1) && (j== 1)) return 9;
    if ((i== 1) && (j== 1)) return 10;
    /*
     0, 0 -> 0
     -1, 0 -> 1
     1, 0 -> 2
```

```
     0,-2 -> 3
     0,-1 -> 4
     0, 1 -> 5
     0, 2 -> 6
     -1,-1 -> 7
     1,-1 -> 8
     -1, 1 -> 9
     1, 1 -> 10

     6
     9 5 10
     1 0 2
     7 4 8
     3

     */
    printf("Error in stencil %d %d\ n",i,j);
    return -1;
}

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

static double double_interpolation(double value_sd_vd, double value_sd_vu,
                                   double value_su_vd, double value_su_vu,
                                   double grid_sd, double grid_su,
                                   double grid_vd, double grid_vu,
                                   double s, double v)
{
    double value_sd,value_su;
    value_sd = interpolation (grid_vd, value_sd_vd, grid_vu, value_sd_vu, v);
    value_su = interpolation (grid_vd, value_su_vd, grid_vu, value_su_vu, v);
    return interpolation (grid_sd, value_sd, grid_su, value_su, s);
}

static int it_exists_stencil(int i, int Ns, int j, int Nv, int stencil)
{
    // We use i from 0 to Ns, j from 0 to Nv.
```

```
// We use points i-1 -> i+1 and j-2 -> j+2.

/*
 0, 0 -> 0
 -1, 0 -> 1
 1, 0 -> 2
 0,-2 -> 3
 0,-1 -> 4
 0, 1 -> 5
 0, 2 -> 6
 -1,-1 -> 7
 1,-1 -> 8
 -1, 1 -> 9
 1, 1 -> 10

 6
 9 5 10
 1 0 2
 7 4 8
 3

 */

if ((i>0) && (i<Ns)&& (j>1) && (j<Nv-1)) // Strict interior domain for all s
{
    return 1;
}

if (i==0)
    if ((stencil== 1) || (stencil== 7) || (stencil== 9))
        return 0;

if (i==Ns)
    if ((stencil== 2) || (stencil== 8) || (stencil==10))
        return 0;

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

if (j==1)
```

```
        if (stencil== 3)
            return 0;

    if (j==Nv-1)
        if (stencil== 6)
            return 0;

    if (j==Nv)
        if ((stencil== 5) || (stencil== 6) || (stencil==9) || (stencil== 10))
            return 0;

    return 1;
}

static void point_of_stencil(int i, int j, int stencil, int* pi, int* pj)
{
    *pi=i;     *pj=j;
    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;}
}

void positivizing(double **U, int Ns, int Nv)
{
    int i, j;

    for(j=0;j<Nv+1;j++)
    {
        for(i=0;i<Ns+1;i++)
        {
            U[i][j]=fabs(U[i][j]);
        }
    }
```

```
}

//////////
// Functions to print.
//////////
void print_array(double *U, int Ns)
{
    int i;

    printf("Sigma = [");
    for(i=0;i<Ns+1;i++)
    {
        for(i=0;i<Ns+1;i++)
        {
            printf("%f ", U[i]);

        }
        printf("]';\ n");
    }
}


void print_vector(double **U, int Ns, int Nv)
{
    int i, j;

    for(j=0;j<Nv+1;j++)
    {
        printf("U(:,%d) = [ ", j+1);
        for(i=0;i<Ns+1;i++)
        {
            printf("%.16f ", U[i][j]);

        }
        printf("]';\ n");
    }
}

void print_matrix(int Ns, int Nv, double ***M)
{
    int i, j, st;
```

```
    double sum;

    for(j=0;j<Nv+1;j++)
    {
        for(i=0;i<Ns+1;i++)
        {
            sum=0.;
            printf("M(%d,%d) = ", i, j);
            for (st=0; st<11 ; st++)
            {
                printf("%f ", M[i][j][st]);
                sum+=M[i][j][st];
            }
            printf("\ nsum over line = %f\ n",sum);
        }
        printf("\ n");
    }
}

//////////
// Functions to manage boundary conditions during classical backward diffusion.
//////////
static double bc_spot_min(double S0, double *sgrid, int Ns, int is,
                          double V0, double sigma_v, double alpha_v, double beta
                          double R0, double divid, double rho_sv,
                          double *tgrid, int Nt, int time_index, int call_or_put
{
    // dt U = sigma_v^2*V/2 dvv U + alpha_v (beta_v - V) dv U - r U
    // + 0 dss U + 0 ds U + 0 dsv U
    // Dirichlet = 0 or K.
    if (call_or_put==1)
        return 0.;
    else
        return strike * exp(-R0*tgrid[Nt-time_index]);
}

static double bc_spot_max(double S0, double *sgrid, int Ns, int is,
                          double V0, double sigma_v, double alpha_v, double beta
                          double R0, double divid, double rho_sv,
                          double *tgrid, int Nt, int time_index, int call_or_put
{
```

```
    // dt U = SSV/2 dss U + (r-a)S ds U + rho_sv sigma_v S V dsv U
    // + sigma_v^2*V/2 dvv U + alpha_v (beta_v - V) dv U - r U
    // Dirichlet = 0 or S.
    // Neuman exp(-dt) or 0

    if (call_or_put==1) // Call
        // Neumann = exp(-dt)
        return exp(-divid*tgrid[Nt-time_index]) * (sgrid[is]-sgrid[is-1]);
    else // (call_or_put==-1) // Put
        // Neumann = 0
        return 0.;
}

static double bc_var_max(double S0, double *sgrid, int Ns, int is,
                         double V0, double sigma_v, double alpha_v, double beta_
                         double R0, double divid, double rho_sv,
                         double *tgrid, int Nt, int time_index, int call_or_put,
{
    // Neumann = 0.
    return 0.;

}

static double bc_var_min(double S0, double *sgrid, int Ns, int is,
                         double V0, double sigma_v, double alpha_v, double beta_
                         double R0, double divid, double rho_sv,
                         double *tgrid, int Nt, int time_index, int call_or_put,
{
    // Neumann = 0.
    return 0.;

}

//////////
// Functions to manage boundary conditions during forward diffusion from Dirac i
//////////
static double bc_spot_min_forward(double S0, double *sgrid, int Ns, int is,
                          double V0, double sigma_v, double alpha_v, double beta
                          double R0, double divid, double rho_sv,
                          double *tgrid, int Nt, int time_index, int call_or_put
{
```

```
    // dt U = sigma_v^2*V/2 dvv U + alpha_v (beta_v - V) dv U - r U
    // + 0 dss U + 0 ds U + 0 dsv U
    // Neumann = 0.
    return 0.;
}

static double bc_spot_max_forward(double S0, double *sgrid, int Ns, int is,
                                  double V0, double sigma_v, double alpha_v, dou
                                  double R0, double divid, double rho_sv,
                                  double *tgrid, int Nt, int time_index, int cal
{
    // dt U = SSV/2 dss U + (r-a)S ds U + rho_sv sigma_v S V dsv U
    // + sigma_v^2*V/2 dvv U + alpha_v (beta_v - V) dv U - r U
    // Neumann = 0.

    return 0.;
}

static double bc_var_max_forward(double S0, double *sgrid, int Ns, int is,
                         double V0, double sigma_v, double alpha_v, double beta_
                         double R0, double divid, double rho_sv,
                         double *tgrid, int Nt, int time_index, int call_or_put,
{
    // Neumann = 0.
    return 0.;

}

static double bc_var_min_forward(double S0, double *sgrid, int Ns, int is,
                         double V0, double sigma_v, double alpha_v, double beta_
                         double R0, double divid, double rho_sv,
                         double *tgrid, int Nt, int time_index, int call_or_put,
{
    // Neumann = 0.
    return 0.;

}

//////////
// Functions to build matrix and solve systems.
//////////
```

```
static void build_all_matrix_forward(double S0, double *sgrid, int Ns,
                              double V0, double sigma_v, double alpha_v, double b
                              double R0, double divid, double rho_sv,
                              double *tgrid, int Nt, int time_index,
                                      int call_or_put, double strike,
                              double *VolSto,
                              double ***MatrixA0, double ***MatrixA1, double ***M
                              double **G0, double **G1, double **G2)
{
    // Matrix for the forward diffusion without actualization.
    double actualspoint;
    double actualvpoint;
    double actualrpoint; // To unify notations.
    double volsto_sm1, volsto_s, volsto_sp1;

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

    double convection_sm1, convection_s, convection_sp1;
    double diffusion_sm1, diffusion_s, diffusion_sp1;
    double convection_vm2, convection_vm1, convection_v, convection_vp1, convect
    double diffusion_vm1, diffusion_v, diffusion_vp1;
    double order_0, mixted_sv;

    double *cs;
    double *ds;
    double *cv;
    double *dv;
    double *msv;

    // 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));
msv=(double*)malloc(11*sizeof(double));

double G_cs, G_ds, G_cv, G_dv, G_msv;

for(j=0;j<Nv+1;j++)
{
    for(i=0;i<Ns+1;i++)
    {
        for(st=0;st<11;st++)
        {
            cs[st]=0.;
            ds[st]=0.;
            cv[st]=0.;
            dv[st]=0.;
            msv[st]=0.;
        }
        G_cs=0.;
        G_ds=0.;
        G_cv=0.;
        G_dv=0.;
        G_msv=0.;

        actualspoint = sgrid[i];
        actualvpoint = vgrid[j];
        actualrpoint = R0-divid;

        (i>0) ? (volsto_sm1 = VolSto[i-1]) : (volsto_sm1=0.);
        volsto_s = VolSto[i];
        (i<Ns) ? (volsto_sp1 = VolSto[i+1]) : (volsto_sp1=0.);


        (i>0) ? (convection_sm1 = -1.0 * actualrpoint * sgrid[i-1]) : (conve
        convection_s = -1.0 * actualrpoint * sgrid[i];
        (i<Ns) ? (convection_sp1 = -1.0 * actualrpoint * sgrid[i+1]) : (conv
```

```
(i>0) ? (diffusion_sm1 = sgrid[i-1] * sgrid[i-1] * actualvpoint/2.0)
diffusion_s = sgrid[i] * sgrid[i] * actualvpoint/2.0;
(i<Ns) ? (diffusion_sp1 = sgrid[i+1] * sgrid[i+1] * actualvpoint/2.0

diffusion_sm1 = diffusion_sm1 * volsto_sm1 * volsto_sm1;
diffusion_s = diffusion_s * volsto_s * volsto_s;
diffusion_sp1 = diffusion_sp1 * volsto_sp1 * volsto_sp1;

 //convection_v = -kappa*(theta(t) - v);
(j>1) ? (convection_vm2 = -1.0 * alpha_v*(beta_v - vgrid[j-2])) : (c
(j>0) ? (convection_vm1 = -1.0 * alpha_v*(beta_v - vgrid[j-1])) : (c
convection_v = -1.0 * alpha_v*(beta_v - vgrid[j]);
(j<Nv) ? (convection_vp1 = -1.0 * alpha_v*(beta_v - vgrid[j+1])) : (
(j<Nv-1) ? (convection_vp2 = -1.0 * alpha_v*(beta_v - vgrid[j+2])) :

(j>0) ? (diffusion_vm1 = sigma_v * sigma_v * vgrid[j-1]/2.0) : (diff
diffusion_v = sigma_v * sigma_v * vgrid[j]/2.0;
(j<Nv) ? (diffusion_vp1 = sigma_v * sigma_v * vgrid[j+1]/2.0) : (dif

order_0 = 0.;//-R0;

mixted_sv = 2. * rho_sv; //* sigma_v * actualspoint * actualvpoint;


if (j==2)
{

//printf("Diff[%d] %f %f %f\ n",i,diffusion_sm1,diffusion_s,diffusio

}

/*
convection_s = convection_s * SPDE;
diffusion_s = diffusion_s * SPDE;
convection_vm2 = convection_vm2 * VPDE;
convection_vm1 = convection_vm1 * VPDE;
convection_v = convection_v * VPDE;
convection_vp1 = convection_vp1 * VPDE;
convection_vp2 = convection_vp2 * VPDE;
diffusion_v = diffusion_v * VPDE;
mixted_sv = mixted_sv * SPDE * VPDE;
```

```
*/

// Method for boundary conditions.
// Compute all the ?grid_step for the finite difference scheme.
// Call bc_?grid_min/max function at the point concerned
// and put it in G * ?grid_step_concerned
// Suppress the bad coefficient in the matrix.
// Modify or not the diagonal in the matrix.

// Example 1 : Dirichlet for diffusion at minimal value on asset gri
// Compute Dsip1, copy it in Dsi.
// Call bc_spot_min at point i-1.
// Suppress ds[stencil(-1,0)]
// Do not modify the diagonal of the matrix.

// Example 2 : Neumann for diffusion at maximal value on variance gr
// Compute Dvj, copy it in Dvjp1.
// Call bc_var_max at point j.
// Suppress ds[stencil(0,1)]
// Modify the diagonal of the matrix with diffusion coefficient.



//////////////////////
// Diffusion and Convection S
//////////////////////
//printf("Diffusion and Convection S.\ n");
{
    if (i==0) // S=Smin -> Neumann
    {
        Dsip1 = sgrid[i+1]-sgrid[i];
        Dsi = Dsip1;
        //ds[stencil(-1,0)]=2.0/(Dsi*(Dsi+Dsip1)) * diffusion_sm1;
        ds[stencil(0,0)]=(-2.0/(Dsi*Dsip1) + 2.0/(Dsi*(Dsi+Dsip1)))
        ds[stencil(1,0)]=2.0/(Dsip1*(Dsi+Dsip1)) * diffusion_sp1;
        G_ds += 2.0/(Dsi*(Dsi+Dsip1)) * diffusion_sm1 *
        bc_spot_min_forward(S0, sgrid, Ns, i,
                    V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
                    R0, divid, rho_sv,
                    tgrid, Nt, time_index, call_or_put, strike);
```

```
    //cs[stencil(-1,0)]=-Dsip1/(Dsi*(Dsi+Dsip1)) * convection_sm
    cs[stencil(0,0)]=((Dsip1-Dsi)/(Dsi*Dsip1) - Dsip1/(Dsi*(Dsi+
    cs[stencil(1,0)]=Dsi/(Dsip1*(Dsi+Dsip1)) * convection_sp1;
    G_cs += -Dsip1/(Dsi*(Dsi+Dsip1)) * convection_sm1 *
    bc_spot_min_forward(S0, sgrid, Ns, i,
                V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
                R0, divid, rho_sv,
                tgrid, Nt, time_index, call_or_put, strike);
}
else
{
    if (i==Ns) // S=Smax -> Neumann
    {
        Dsi = sgrid[i]-sgrid[i-1];
        Dsip1 = Dsi;
        ds[stencil(-1,0)]=2.0/(Dsi*(Dsi+Dsip1)) * diffusion_sm1;
        ds[stencil(0,0)]=(-2.0/(Dsi*Dsip1) + 2.0/(Dsip1*(Dsi+Dsi
        //ds[stencil(1,0)]=2.0/(Dsip1*(Dsi+Dsip1)) * diffusion_s
        G_ds = 2.0/(Dsip1*(Dsi+Dsip1)) * diffusion_sp1 *
        bc_spot_max_forward(S0, sgrid, Ns, i,
                    V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
                    R0, divid, rho_sv,
                    tgrid, Nt, time_index, call_or_put, strike);

        cs[stencil(-1,0)]=-Dsip1/(Dsi*(Dsi+Dsip1)) * convection_
        cs[stencil(0,0)]=((Dsip1-Dsi)/(Dsi*Dsip1) + Dsi/(Dsip1*(
        //cs[stencil(1,0)]=Dsi/(Dsip1*(Dsi+Dsip1)) * convection_
        G_cs += Dsi/(Dsip1*(Dsi+Dsip1)) * convection_sp1 *
        bc_spot_max_forward(S0, sgrid, Ns, i,
                    V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
                    R0, divid, rho_sv,
                    tgrid, Nt, time_index, call_or_put, strike);
    }
    else
    {
        Dsi = sgrid[i]-sgrid[i-1];
        Dsip1 = sgrid[i+1]-sgrid[i];
        ds[stencil(-1,0)]=2.0/(Dsi*(Dsi+Dsip1)) * diffusion_sm1;
        ds[stencil(0,0)]=-2.0/(Dsi*Dsip1) * diffusion_s;
        ds[stencil(1,0)]=2.0/(Dsip1*(Dsi+Dsip1)) * diffusion_sp1
```

```
            cs[stencil(-1,0)]=-Dsip1/(Dsi*(Dsi+Dsip1)) * convection_
            cs[stencil(0,0)]=(Dsip1-Dsi)/(Dsi*Dsip1) * convection_s;
            cs[stencil(1,0)]=Dsi/(Dsip1*(Dsi+Dsip1)) *  convection_s
        }
    }
}
////////////////////
// Diffusion and Convection V
////////////////////
//printf("Diffusion and Convection V.\ n");
{
    if (j==0) // V=Vmin
    {
        // V=Vmin -> Diffusion = 0 and no boundary conditions.

        // V=Vmin -> Forward in convection.
        Dvjp1 = vgrid[j+1]-vgrid[j];
        Dvjp2 = vgrid[j+2]-vgrid[j+1];

        //cv[stencil(0,-1)]=-Dvjp1/(Dvj*(Dvj+Dvjp1)) * convection_vm
        //cv[stencil(0,0)]=((Dvjp1-Dvj)/(Dvj*Dvjp1) - Dvjp1/(Dvj*(Dv
        //cv[stencil(0,1)]=Dvj/(Dvjp1*(Dvj+Dvjp1)) * convection_vp1;
        //G_cv += -Dvjp1/(Dvj*(Dvj+Dvjp1)) * convection_vm1 *
        //bc_var_min(S0, sgrid, Ns, i,
        //           V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
        //           R0, divid, rho_sv,
        //           tgrid, Nt, time_index);

        cv[stencil(0,0)]=-(2.0*Dvjp1+Dvjp2)/(Dvjp1*(Dvjp1+Dvjp2)) *
        cv[stencil(0,1)]=(Dvjp1+Dvjp2)/(Dvjp1*Dvjp2) * convection_vp
        cv[stencil(0,2)]=-Dvjp1/(Dvjp2*(Dvjp1+Dvjp2)) * convection_v
    }
    else
    {
        if (j==Nv) // V=Vmax
        {
            // V=Vmax -> Neumann for diffusion.
            Dvjm1 = vgrid[j-1]-vgrid[j-2];
            Dvj = vgrid[j]-vgrid[j-1];
            Dvjp1 = Dvj;
            dv[stencil(0,-1)]=2.0/(Dvj*(Dvj+Dvjp1)) * diffusion_vm1;
```

```
    dv[stencil(0,0)]=(-2.0/(Dvj*Dvjp1) + 2.0/(Dvjp1*(Dvj+Dvj
    //dv[stencil(0,1)]=2.0/(Dvjp1*(Dvj+Dvjp1)) * diffusion_v
    G_dv += 2.0/(Dvjp1*(Dvj+Dvjp1)) * diffusion_vp1 *
    bc_var_max_forward(S0, sgrid, Ns, i,
               V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
               R0, divid, rho_sv,
               tgrid, Nt, time_index, call_or_put, strike);

    // V=Vmax -> Backward for convection.
    //cv[stencil(0,0)]=(Dvjm1+2.0*Dvj)/(Dvj*(Dvjm1+Dvj)) * c
    //cv[stencil(0,-1)]=-(Dvjm1+Dvj)/(Dvjm1*Dvj) * convectio
    //cv[stencil(0,-2)]=Dvj/(Dvjm1*(Dvjm1+Dvj)) * convection

    cv[stencil(0,-1)]=-Dvjp1/(Dvj*(Dvj+Dvjp1)) * convection_
    cv[stencil(0,0)]=((Dvjp1-Dvj)/(Dvj*Dvjp1) + Dvj/(Dvjp1*(
    //cv[stencil(0,1)]=Dvj/(Dvjp1*(Dvj+Dvjp1)) * convection_
    G_cv += Dvj/(Dvjp1*(Dvj+Dvjp1)) * convection_vp1 *
    bc_var_max_forward(S0, sgrid, Ns, i,
                V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
                R0, divid, rho_sv,
                tgrid, Nt, time_index, call_or_put, strike);

}
else
{
    // If 1 <= j <= Nv-1 -> use central scheme for diffusion
    Dvj = vgrid[j]-vgrid[j-1];
    Dvjp1 = vgrid[j+1]-vgrid[j];

    dv[stencil(0,-1)]=2.0/(Dvj*(Dvj+Dvjp1)) * diffusion_vm1;
    dv[stencil(0,0)]=-2.0/(Dvj*Dvjp1) * diffusion_v;
    dv[stencil(0,1)]=2.0/(Dvjp1*(Dvj+Dvjp1)) * diffusion_vp1

    // If 1 <= j <= Nv-1 -> scheme for convection depends on
    //if (j==1)
    //{
        //Dvjm1 = vgrid[j-1]-vgrid[j-2];

        //cv[stencil(0,-1)]=-Dvjp1/(Dvj*(Dvj+Dvjp1)) * conve
        //cv[stencil(0,0)]=((Dvjp1-Dvj)/(Dvj*Dvjp1) - Dvjp1/
        //cv[stencil(0,1)]=Dvj/(Dvjp1*(Dvj+Dvjp1)) * convect
```

```
                    //G_cv += -Dvjp1/(Dvj*(Dvj+Dvjp1)) * convection_vm1
                    //bc_var_min(S0, sgrid, Ns, i,
                    //            V0, sigma_v, alpha_v, beta_v, vgrid, N
                    //            R0, divid, rho_sv,
                    //            tgrid, Nt, time_index);
                //}
                //else // Centered scheme
                //{
                //if ((convection_v<0.)) // var > beta_v and j>1 -> Forw
                //{
                //    Dvjm1 = vgrid[j-1]-vgrid[j-2];
                //
                //    cv[stencil(0,-2)]=Dvj/(Dvjm1*(Dvjm1+Dvj)) * convec
                //    cv[stencil(0,-1)]=-(Dvjm1+Dvj)/(Dvjm1*Dvj) * conve
                //    cv[stencil(0,0)]=(Dvjm1+2.0*Dvj)/(Dvj*(Dvjm1+Dvj))
                //}
                //else // var <= beta_v or j==1 -> Central.
                //{
                    //Dvjm1 = vgrid[j]-vgrid[j-1];

                    cv[stencil(0,-1)]=-Dvjp1/(Dvj*(Dvj+Dvjp1)) * convect
                    cv[stencil(0,0)]=(Dvjp1-Dvj)/(Dvj*Dvjp1) * convectio
                    cv[stencil(0,1)]=Dvj/(Dvjp1*(Dvj+Dvjp1)) * convectio

            }
        }
    }

    // Mixted SV
    //printf("Mixted SV.\ n");
    {
        // If S=0 or V=0 ->> misted_sv = 0.

        // Boundary condition elsewhere :
        // if V=Vmax, we impose Neumann on V independent of S
        // ->> no problem at S=Smin since mixted_sv = 0.
        // ->> no problem at S=Smax since also Neumann on S independent
        // if S=Smax, we impose Neumann on S independent of V
        // ->> no problem at V=Vmin since mixted_sv = 0.
        // ->> no problem at V=Vmax since also Neumann on V independent
```

```
if ((0<i) && (i<Ns) && (0<j) && (j<Nv)) // Strict interior 0<i<N
{
    Dsi = sgrid[i]-sgrid[i-1];
    Dsip1 = sgrid[i+1]-sgrid[i];
    Dvj = vgrid[j]-vgrid[j-1];
    Dvjp1 = vgrid[j+1]-vgrid[j];

    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));

    msv[stencil(0,0)] = sbeta_i0 * vbeta_j0 * mixted_sv * sqrt(d
    msv[stencil(-1,0)] = sbeta_im1 * vbeta_j0 * mixted_sv * sqrt
    msv[stencil(1,0)] = sbeta_ip1 * vbeta_j0 * mixted_sv * sqrt(
    msv[stencil(0,-1)] = sbeta_i0 * vbeta_jm1 * mixted_sv * sqrt
    msv[stencil(0,1)] = sbeta_i0 * vbeta_jp1 * mixted_sv * sqrt(
    msv[stencil(-1,-1)] = sbeta_im1 * vbeta_jm1 * mixted_sv * sq
    msv[stencil(1,-1)] = sbeta_ip1 * vbeta_jm1 * mixted_sv * sqr
    msv[stencil(-1,1)] = sbeta_im1 * vbeta_jp1 * mixted_sv * sqr
    msv[stencil(1,1)] = sbeta_ip1 * vbeta_jp1 * mixted_sv * sqrt
}
else // i==0 or i==Ns or j==0 or j==Nv
{
    if (j==Nv) // V=Vmax
    {
        // Three cases :
        // i==0 -> mixted_sv = 0 -> Nothing to do.
        // 0<i<Ns -> Condition on V=Vmax.
        // i==Ns -> Condition on V=Vmax compatible with conditio
        if ((0<i) && (i<Ns))
        {
            Dsi = sgrid[i]-sgrid[i-1];
            Dsip1 = sgrid[i+1]-sgrid[i];
            Dvj = vgrid[j]-vgrid[j-1];
            Dvjp1 = Dvj;

            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));


            msv[stencil(-1,0)] = sbeta_im1 * vbeta_j0 * mixted_s
            msv[stencil(1,0)] = sbeta_ip1 * vbeta_j0 * mixted_sv
            msv[stencil(0,-1)] = sbeta_i0 * vbeta_jm1 * mixted_s
            //msv[stencil(0,1)] = sbeta_i0 * vbeta_jp1;
            G_msv += sbeta_i0 * vbeta_jp1 * mixted_sv * sqrt(dif
            bc_var_max_forward(S0, sgrid, Ns, i,
                                                    V0, sigma
                                                    R0, divid
                                                    tgrid, Nt
            msv[stencil(-1,-1)] = sbeta_im1 * vbeta_jm1 * mixted
            msv[stencil(1,-1)] = sbeta_ip1 * vbeta_jm1 * mixted_
            //msv[stencil(-1,1)] = sbeta_im1 * vbeta_jp1;
            G_msv += sbeta_im1 * vbeta_jp1 * mixted_sv * sqrt(di
            bc_var_max_forward(S0, sgrid, Ns, i-1,
                                                     V0, sigm
                                                     R0, divi
                                                     tgrid, N
            //msv[stencil(1,1)] = sbeta_ip1 * vbeta_jp1;
            G_msv += sbeta_ip1 * vbeta_jp1 * mixted_sv * sqrt(di
            bc_var_max_forward(S0, sgrid, Ns, i+1,
                                                     V0, sigm
                                                     R0, divi
                                                     tgrid, N

            msv[stencil(0,0)] =(
            sbeta_i0 * vbeta_j0
            + 1. * sbeta_im1 * vbeta_jp1 // Neumann
            + 1. * sbeta_i0 * vbeta_jp1 // Neumann
            + 1. * sbeta_ip1 * vbeta_jp1 // Neumann
                                ) * mixted_sv * sqrt(diffusion_s
        }
    } // End of V=Vmax -> we do not have done i==Ns in j==Nv.
    if (i==Ns) // S=Smax
    {
        // Three cases :
        // j==0 -> mixted_sv = 0 -> Nothing to do.
```

```
// 0<j<Nv -> Condition on S=Smax.
// j==Nv -> Condition on S=Smax compatible with conditio
if ((0<j) && (j<Nv))
{
    Dsi = sgrid[i]-sgrid[i-1];
    Dsip1 = Dsi;
    Dvj = vgrid[j]-vgrid[j-1];
    Dvjp1 = Dvj;

    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));


    msv[stencil(-1,0)] = sbeta_im1 * vbeta_j0 * mixted_s
    //msv[stencil(1,0)] = sbeta_ip1 * vbeta_j0 * mixted_
    G_msv += sbeta_ip1 * vbeta_j0 * mixted_sv * sqrt(dif
    bc_spot_max_forward(S0, sgrid, Ns, i,
                                                V0, sigm
                                                R0, divi
                                                tgrid, N
    msv[stencil(0,-1)] = sbeta_i0 * vbeta_jm1 * mixted_s
    msv[stencil(0,1)] = sbeta_i0 * vbeta_jp1 * mixted_sv
    msv[stencil(-1,-1)] = sbeta_im1 * vbeta_jm1 * mixted
    //msv[stencil(1,-1)] = sbeta_ip1 * vbeta_jm1 * mixte
    G_msv += sbeta_ip1 * vbeta_jm1 * mixted_sv * sqrt(di
    bc_spot_max_forward(S0, sgrid, Ns, i,
                                                 V0, sig
                                                 R0, div
                                                 tgrid,
    msv[stencil(-1,1)] = sbeta_im1 * vbeta_jp1 * mixted_
    //msv[stencil(1,1)] = sbeta_ip1 * vbeta_jp1 * mixted
    G_msv += sbeta_ip1 * vbeta_jp1 * mixted_sv * sqrt(di
    bc_spot_max_forward(S0, sgrid, Ns, i,
                                                V0, sig
                                                R0, div
                                                tgrid,

    msv[stencil(0,0)] = (
```

```
        sbeta_i0 * vbeta_j0
        + 1. * sbeta_ip1 * vbeta_jm1 // Neumann
        + 1. * sbeta_ip1 * vbeta_j0 // Neumann
        + 1. * sbeta_ip1 * vbeta_jp1 // Neumann
                              ) * mixted_sv * sqrt(diffusion_
    }
} // End of S=Smax -> we do not have done j==Nv in i==Ns. Do
if ((i==Ns) && (j==Nv)) // Corner ! The two Neumann conditio
{
    Dsi = sgrid[i]-sgrid[i-1];
    Dsip1 = Dsi;
    Dvj = vgrid[j]-vgrid[j-1];
    Dvjp1 = Dvj;

    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));


    msv[stencil(-1,0)] = sbeta_im1 * vbeta_j0 * mixted_sv *
    //msv[stencil(1,0)] = sbeta_ip1 * vbeta_j0 * mixted_sv *
    G_msv += sbeta_ip1 * vbeta_j0 * mixted_sv * sqrt(diffusi
    bc_spot_max_forward(S0, sgrid, Ns, i,
                                                V0, sigma_v,
                                                R0, divid, r
                                                tgrid, Nt, t
    msv[stencil(0,-1)] = sbeta_i0 * vbeta_jm1 * mixted_sv *
    //msv[stencil(0,1)] = sbeta_i0 * vbeta_jp1 * mixted_sv *
    G_msv += sbeta_i0 * vbeta_jp1 * mixted_sv * sqrt(diffusi
    bc_var_max_forward(S0, sgrid, Ns, i,
                                               V0, sigma_v,
                                               R0, divid, rh
                                               tgrid, Nt, ti
    msv[stencil(-1,-1)] = sbeta_im1 * vbeta_jm1 * mixted_sv
    //msv[stencil(1,-1)] = sbeta_ip1 * vbeta_jm1 * mixted_sv
    G_msv += sbeta_ip1 * vbeta_jm1 * mixted_sv * sqrt(diffus
    bc_spot_max_forward(S0, sgrid, Ns, i,
                                                 V0, sigma_v
```

```
                                                          R0, divid,
                                                          tgrid, Nt,
            //msv[stencil(-1,1)] = sbeta_im1 * vbeta_jp1 * mixted_sv
            G_msv += sbeta_im1 * vbeta_jp1 * mixted_sv * sqrt(diffus
            bc_var_max_forward(S0, sgrid, Ns, i,
                                                        V0, sigma_v,
                                                        R0, divid, r
                                                        tgrid, Nt, t
            //msv[stencil(1,1)] = sbeta_ip1 * vbeta_jp1 * mixted_sv
            // !!!! Here COMPATIBILITY is OK !!!!
            G_msv += sbeta_ip1 * vbeta_jp1 * mixted_sv * sqrt(diffus
            bc_spot_max_forward(S0, sgrid, Ns, i,
                        V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
                        R0, divid, rho_sv,
                        tgrid, Nt, time_index, call_or_put, strike);
            msv[stencil(0,0)] = (
            sbeta_i0 * vbeta_j0
            + 1. * sbeta_ip1 * vbeta_j0 // Neumann
            + 1. * sbeta_i0 * vbeta_jp1 // Neumann
            + 1. * sbeta_ip1 * vbeta_jm1 // Neumann
            + 1. * sbeta_im1 * vbeta_jp1 // Neumann
            + 1. * sbeta_ip1 * vbeta_jp1 // Neumann
                                  ) * mixted_sv * sqrt(diffusion_s *
        }
    }
}

// Central point for order_0
MatrixA0[i][j][0] = msv[0] * SPDE * VPDE;
MatrixA1[i][j][0] = order_0 * (SPDE/(SPDE+VPDE)) + cs[0] * SPDE + ds
MatrixA2[i][j][0] = order_0 * (VPDE/(SPDE+VPDE)) + cv[0] * VPDE + dv

for (st=1;st<11;st++)
{
    MatrixA0[i][j][st] = msv[st] * SPDE * VPDE;
    MatrixA1[i][j][st] = cs[st] * SPDE + ds[st] * SPDE;
    // Be careful, convection_v is changed inside the building
    MatrixA2[i][j][st] = cv[st] * VPDE + dv[st] * VPDE;
}

// Second members
```

```
            G0[i][j] = G_msv * SPDE * VPDE;
            G1[i][j] = G_cs * SPDE + G_ds * SPDE;
            // Be careful, convection_v is changed inside the building
            G2[i][j] = G_cv * VPDE + G_dv * VPDE;

        }
    }

    free(msv);
    free(dv);
    free(cv);
    free(ds);
    free(cs);
}


static void build_all_matrix(double S0, double *sgrid, int Ns,
                             double V0, double sigma_v, double alpha_v, double b
                             double R0, double divid, double rho_sv,
                             double *tgrid, int Nt, int time_index,int call_or_p
                             double *VolSto,
                             double ***MatrixA0, double ***MatrixA1, double ***M
                             double **G0, double **G1, double **G2)
{
    double actualspoint;
    double actualvpoint;
    double actualrpoint; // To unify notations.
    double volsto_sm1, volsto_s, volsto_sp1;

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

    double convection_sm1, convection_s, convection_sp1;
    double diffusion_sm1, diffusion_s, diffusion_sp1;
    double convection_vm2, convection_vm1, convection_v, convection_vp1, convect
    double diffusion_vm1, diffusion_v, diffusion_vp1;
    double order_0, mixted_sv;

    double *cs;
    double *ds;
    double *cv;
```

```
double *dv;
double *msv;

// 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));
msv=(double*)malloc(11*sizeof(double));

double G_cs, G_ds, G_cv, G_dv, G_msv;

for(j=0;j<Nv+1;j++)
{
    for(i=0;i<Ns+1;i++)
    {
        for(st=0;st<11;st++)
        {
            cs[st]=0.;
            ds[st]=0.;
            cv[st]=0.;
            dv[st]=0.;
            msv[st]=0.;
        }
        G_cs=0.;
        G_ds=0.;
        G_cv=0.;
        G_dv=0.;
        G_msv=0.;
```

```
actualspoint = sgrid[i];
actualvpoint = vgrid[j];
actualrpoint = R0-divid;

volsto_s = VolSto[i];
volsto_sm1 = volsto_s;
volsto_sp1 = volsto_s;

diffusion_s = actualspoint * actualspoint * actualvpoint/2.0;
diffusion_sm1 = diffusion_s;
diffusion_sp1 = diffusion_s;

// Add vol sto to diffusion
diffusion_sm1 = diffusion_sm1 * volsto_sm1 * volsto_sm1;
diffusion_s = diffusion_s * volsto_s * volsto_s;
diffusion_sp1 = diffusion_sp1 * volsto_sp1 * volsto_sp1;

convection_s = 1.0 * actualrpoint * actualspoint;
convection_sm1 = convection_s;
convection_sp1 = convection_s;

convection_v = 1.0 * alpha_v*(beta_v - actualvpoint);
convection_vm2 = convection_v;
convection_vm1 = convection_v;
convection_vp1 = convection_v;
convection_vp2 = convection_v;

diffusion_v = sigma_v * sigma_v * actualvpoint/2.0;
diffusion_vm1 = diffusion_v;
diffusion_vp1 = diffusion_v;

// X#X#X#X#X#X#X
// X#X#X#X#X#X#X
// X#X#X#X#X#X#X
// X#X#X#X#X#X#X
// X#X#X#X#X#X#X
order_0 = -R0;
// X#X#X#X#X#X#X
// X#X#X#X#X#X#X
// X#X#X#X#X#X#X
```

```
// X#X#X#X#X#X#X
// X#X#X#X#X#X#X
// X#X#X#X#X#X#X

// Be careful mixted_sv should be rho_sv * sigma_v * actualspoint *
// but later in the code we use mix_derivatives <- mixted_sv * sqrt(
// which could be mixted_sv * actualspoint * sqrt(actualvpoint/2) *
mixted_sv = 2. * rho_sv;


if (j==2)
{
    //printf("Diff[%d] %f %f %f\ n",i,diffusion_sm1,diffusion_s,diff
}

/*
 convection_s = convection_s * SPDE;
 diffusion_s = diffusion_s * SPDE;
 convection_vm2 = convection_vm2 * VPDE;
 convection_vm1 = convection_vm1 * VPDE;
 convection_v = convection_v * VPDE;
 convection_vp1 = convection_vp1 * VPDE;
 convection_vp2 = convection_vp2 * VPDE;
 diffusion_v = diffusion_v * VPDE;
 mixted_sv = mixted_sv * SPDE * VPDE;
 */

// Method for boundary conditions.
// Compute all the ?grid_step for the finite difference scheme.
// Call the function bc_?grid_min/max at the concerned point
// and put it in G * ?grid_step
// Suppress the bad coefficient in the matrix.
// Modify or not the diagonal in the matrix depending on the boundar

// Example 1 : Dirichlet for diffusion at minimal value on asset gri
// Compute Dsip1, copy it in Dsi.
// Call bc_spot_min at point i-1.
// Suppress ds[stencil(-1,0)]
// Do not modify the diagonal of the matrix.

// Example 2 : Neumann for diffusion at maximal value on variance gr
```

```
// Compute Dvj, copy it in Dvjp1.
// Call bc_var_max at point j.
// Suppress ds[stencil(0,1)]
// Modify the diagonal of the matrix with diffusion coefficient.



////////////////////
// Diffusion and Convection S
////////////////////
//printf("Diffusion and Convection S.\ n");
{
    if (i==0) // S=Smin -> DIRICHLET
    {
        Dsip1 = sgrid[i+1]-sgrid[i];
        Dsi = Dsip1;
        //ds[stencil(-1,0)]=2.0/(Dsi*(Dsi+Dsip1)) * diffusion_sm1;
        // For Neumann ds[stencil(0,0)]=(-2.0/(Dsi*Dsip1) + 2.0/(Dsi
        // For Dirichlet ds[stencil(0,0)]=(-2.0/(Dsi*Dsip1)) * diffu
        ds[stencil(0,0)]=(-2.0/(Dsi*Dsip1)) * diffusion_s;
        ds[stencil(1,0)]=2.0/(Dsip1*(Dsi+Dsip1)) * diffusion_sp1;
        G_ds += 2.0/(Dsi*(Dsi+Dsip1)) * diffusion_sm1 *
        bc_spot_min(S0, sgrid, Ns, i,
                    V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
                    R0, divid, rho_sv,
                    tgrid, Nt, time_index, call_or_put, strike);

        //cs[stencil(-1,0)]=-Dsip1/(Dsi*(Dsi+Dsip1)) * convection_sm
        // For Neumann cs[stencil(0,0)]=((Dsip1-Dsi)/(Dsi*Dsip1) - D
        // For Dirichlet cs[stencil(0,0)]=((Dsip1-Dsi)/(Dsi*Dsip1))
        cs[stencil(0,0)]=((Dsip1-Dsi)/(Dsi*Dsip1)) * convection_s;
        cs[stencil(1,0)]=Dsi/(Dsip1*(Dsi+Dsip1)) * convection_sp1;
        G_cs += -Dsip1/(Dsi*(Dsi+Dsip1)) * convection_sm1 *
        bc_spot_min(S0, sgrid, Ns, i,
                    V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
                    R0, divid, rho_sv,
                    tgrid, Nt, time_index, call_or_put, strike);
    }
    else
    {
        if (i==Ns) // S=Smax -> Neumann
```

```
        {
            Dsi = sgrid[i]-sgrid[i-1];
            Dsip1 = Dsi;
            ds[stencil(-1,0)]=2.0/(Dsi*(Dsi+Dsip1)) * diffusion_sm1;
            ds[stencil(0,0)]=(-2.0/(Dsi*Dsip1) + 2.0/(Dsip1*(Dsi+Dsi
            //ds[stencil(1,0)]=2.0/(Dsip1*(Dsi+Dsip1)) * diffusion_s
            G_ds = 2.0/(Dsip1*(Dsi+Dsip1)) * diffusion_sp1 *
            bc_spot_max(S0, sgrid, Ns, i,
                        V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
                        R0, divid, rho_sv,
                        tgrid, Nt, time_index, call_or_put, strike);

            cs[stencil(-1,0)]=-Dsip1/(Dsi*(Dsi+Dsip1)) * convection_
            cs[stencil(0,0)]=((Dsip1-Dsi)/(Dsi*Dsip1) + Dsi/(Dsip1*(
            //cs[stencil(1,0)]=Dsi/(Dsip1*(Dsi+Dsip1)) * convection_
            G_cs += Dsi/(Dsip1*(Dsi+Dsip1)) * convection_sp1 *
            bc_spot_max(S0, sgrid, Ns, i,
                        V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
                        R0, divid, rho_sv,
                        tgrid, Nt, time_index, call_or_put, strike);
        }
        else
        {
            Dsi = sgrid[i]-sgrid[i-1];
            Dsip1 = sgrid[i+1]-sgrid[i];
            ds[stencil(-1,0)]=2.0/(Dsi*(Dsi+Dsip1)) * diffusion_sm1;
            ds[stencil(0,0)]=-2.0/(Dsi*Dsip1) * diffusion_s;
            ds[stencil(1,0)]=2.0/(Dsip1*(Dsi+Dsip1)) * diffusion_sp1

            cs[stencil(-1,0)]=-Dsip1/(Dsi*(Dsi+Dsip1)) * convection_
            cs[stencil(0,0)]=(Dsip1-Dsi)/(Dsi*Dsip1) * convection_s;
            cs[stencil(1,0)]=Dsi/(Dsip1*(Dsi+Dsip1)) *  convection_s
        }
    }
}
////////////////////
// Diffusion and Convection V
////////////////////
//printf("Diffusion and Convection V.\ n");
{
    if (j==0) // V=Vmin
```

```
{
    // V=Vmin -> Diffusion = 0 and no boundary conditions.

    // V=Vmin -> Forward in convection.
    Dvjp1 = vgrid[j+1]-vgrid[j];
    Dvjp2 = vgrid[j+2]-vgrid[j+1];

    //cv[stencil(0,-1)]=-Dvjp1/(Dvj*(Dvj+Dvjp1)) * convection_vm
    //cv[stencil(0,0)]=((Dvjp1-Dvj)/(Dvj*Dvjp1) - Dvjp1/(Dvj*(Dv
    //cv[stencil(0,1)]=Dvj/(Dvjp1*(Dvj+Dvjp1)) * convection_vp1;
    //G_cv += -Dvjp1/(Dvj*(Dvj+Dvjp1)) * convection_vm1 *
    //bc_var_min(S0, sgrid, Ns, i,
    //           V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
    //           R0, divid, rho_sv,
    //           tgrid, Nt, time_index);

    cv[stencil(0,0)]=-(2.0*Dvjp1+Dvjp2)/(Dvjp1*(Dvjp1+Dvjp2)) *
    cv[stencil(0,1)]=(Dvjp1+Dvjp2)/(Dvjp1*Dvjp2) * convection_vp
    cv[stencil(0,2)]=-Dvjp1/(Dvjp2*(Dvjp1+Dvjp2)) * convection_v
}
else
{
    if (j==Nv) // V=Vmax
    {
        // V=Vmax -> Neumann for diffusion.
        Dvjm1 = vgrid[j-1]-vgrid[j-2];
        Dvj = vgrid[j]-vgrid[j-1];
        Dvjp1 = Dvj;
        dv[stencil(0,-1)]=2.0/(Dvj*(Dvj+Dvjp1)) * diffusion_vm1;
        dv[stencil(0,0)]=(-2.0/(Dvj*Dvjp1) + 2.0/(Dvjp1*(Dvj+Dvj
        //dv[stencil(0,1)]=2.0/(Dvjp1*(Dvj+Dvjp1)) * diffusion_v
        G_dv += 2.0/(Dvjp1*(Dvj+Dvjp1)) * diffusion_vp1 *
        bc_var_max(S0, sgrid, Ns, i,
                   V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
                   R0, divid, rho_sv,
                   tgrid, Nt, time_index, call_or_put, strike);

        // V=Vmax -> Backward for convection.
        //cv[stencil(0,0)]=(Dvjm1+2.0*Dvj)/(Dvj*(Dvjm1+Dvj)) * c
        //cv[stencil(0,-1)]=-(Dvjm1+Dvj)/(Dvjm1*Dvj) * convectio
        //cv[stencil(0,-2)]=Dvj/(Dvjm1*(Dvjm1+Dvj)) * convection
```

```
    cv[stencil(0,-1)]=-Dvjp1/(Dvj*(Dvj+Dvjp1)) * convection_
    cv[stencil(0,0)]=((Dvjp1-Dvj)/(Dvj*Dvjp1) + Dvj/(Dvjp1*(
    //cv[stencil(0,1)]=Dvj/(Dvjp1*(Dvj+Dvjp1)) * convection_
    G_cv += Dvj/(Dvjp1*(Dvj+Dvjp1)) * convection_vp1 *
    bc_var_max(S0, sgrid, Ns, i,
               V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
               R0, divid, rho_sv,
               tgrid, Nt, time_index, call_or_put, strike);

}
else
{
    // If 1 <= j <= Nv-1 -> use central scheme for diffusion
    Dvj = vgrid[j]-vgrid[j-1];
    Dvjp1 = vgrid[j+1]-vgrid[j];

    dv[stencil(0,-1)]=2.0/(Dvj*(Dvj+Dvjp1)) * diffusion_vm1;
    dv[stencil(0,0)]=-2.0/(Dvj*Dvjp1) * diffusion_v;
    dv[stencil(0,1)]=2.0/(Dvjp1*(Dvj+Dvjp1)) * diffusion_vp1

    // If 1 <= j <= Nv-1 -> scheme for convection depends on
    //if (j==1)
    //{
    //Dvjm1 = vgrid[j-1]-vgrid[j-2];

    //cv[stencil(0,-1)]=-Dvjp1/(Dvj*(Dvj+Dvjp1)) * convectio
    //cv[stencil(0,0)]=((Dvjp1-Dvj)/(Dvj*Dvjp1) - Dvjp1/(Dvj
    //cv[stencil(0,1)]=Dvj/(Dvjp1*(Dvj+Dvjp1)) * convection_
    //G_cv += -Dvjp1/(Dvj*(Dvj+Dvjp1)) * convection_vm1 *
    //bc_var_min(S0, sgrid, Ns, i,
    //           V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j
    //           R0, divid, rho_sv,
    //           tgrid, Nt, time_index);
    //}
    //else // Centered scheme
    //{
    //if ((convection_v<0.)) // var > beta_v and j>1 -> Forw
    //{
    //    Dvjm1 = vgrid[j-1]-vgrid[j-2];
    //
```

```
            //    cv[stencil(0,-2)]=Dvj/(Dvjm1*(Dvjm1+Dvj)) * convec
            //    cv[stencil(0,-1)]=-(Dvjm1+Dvj)/(Dvjm1*Dvj) * conve
            //    cv[stencil(0,0)]=(Dvjm1+2.0*Dvj)/(Dvj*(Dvjm1+Dvj))
            //}
            //else // var <= beta_v or j==1 -> Central.
            //{
            //Dvjm1 = vgrid[j]-vgrid[j-1];

            cv[stencil(0,-1)]=-Dvjp1/(Dvj*(Dvj+Dvjp1)) * convection_
            cv[stencil(0,0)]=(Dvjp1-Dvj)/(Dvj*Dvjp1) * convection_v;
            cv[stencil(0,1)]=Dvj/(Dvjp1*(Dvj+Dvjp1)) * convection_vp

        }
    }
}

// Mixted SV
//printf("Mixted SV.\ n");
{
    // If S=0 or V=0 ->> misted_sv = 0. GOOD NEWS since no treatment

    // Boundary condition elsewhere :
    // if V=Vmax, we impose Neumann on V independent of S
    // ->> no problem at S=Smin since mixted_sv = 0.
    // ->> no problem at S=Smax since also Neumann on S independent
    // if S=Smax, we impose Neumann on S independent of V
    // ->> no problem at V=Vmin since mixted_sv = 0.
    // ->> no problem at V=Vmax since also Neumann on V independent

    if ((0<i) && (i<Ns) && (0<j) && (j<Nv)) // Strict interior 0<i<N
    {
        Dsi = sgrid[i]-sgrid[i-1];
        Dsip1 = sgrid[i+1]-sgrid[i];
        Dvj = vgrid[j]-vgrid[j-1];
        Dvjp1 = vgrid[j+1]-vgrid[j];

        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));

    msv[stencil(0,0)] = sbeta_i0 * vbeta_j0 * mixted_sv * sqrt(d
    msv[stencil(-1,0)] = sbeta_im1 * vbeta_j0 * mixted_sv * sqrt
    msv[stencil(1,0)] = sbeta_ip1 * vbeta_j0 * mixted_sv * sqrt(
    msv[stencil(0,-1)] = sbeta_i0 * vbeta_jm1 * mixted_sv * sqrt
    msv[stencil(0,1)] = sbeta_i0 * vbeta_jp1 * mixted_sv * sqrt(
    msv[stencil(-1,-1)] = sbeta_im1 * vbeta_jm1 * mixted_sv * sq
    msv[stencil(1,-1)] = sbeta_ip1 * vbeta_jm1 * mixted_sv * sqr
    msv[stencil(-1,1)] = sbeta_im1 * vbeta_jp1 * mixted_sv * sqr
    msv[stencil(1,1)] = sbeta_ip1 * vbeta_jp1 * mixted_sv * sqrt
}
else // i==0 or i==Ns or j==0 or j==Nv
{
    if (j==Nv) // V=Vmax
    {
        // Three cases :
        // i==0 -> mixted_sv = 0 -> Nothing to do.
        // 0<i<Ns -> Condition on V=Vmax.
        // i==Ns -> Condition on V=Vmax compatible with conditio
        if ((0<i) && (i<Ns))
        {
            Dsi = sgrid[i]-sgrid[i-1];
            Dsip1 = sgrid[i+1]-sgrid[i];
            Dvj = vgrid[j]-vgrid[j-1];
            Dvjp1 = Dvj;

            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));


            msv[stencil(-1,0)] = sbeta_im1 * vbeta_j0 * mixted_s
            msv[stencil(1,0)] = sbeta_ip1 * vbeta_j0 * mixted_sv
            msv[stencil(0,-1)] = sbeta_i0 * vbeta_jm1 * mixted_s
            //msv[stencil(0,1)] = sbeta_i0 * vbeta_jp1;
            G_msv += sbeta_i0 * vbeta_jp1 * mixted_sv * sqrt(dif
            bc_var_max(S0, sgrid, Ns, i,
```

```
                    V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                    R0, divid, rho_sv,
                    tgrid, Nt, time_index, call_or_put, strik
            msv[stencil(-1,-1)] = sbeta_im1 * vbeta_jm1 * mixted
            msv[stencil(1,-1)] = sbeta_ip1 * vbeta_jm1 * mixted_
            //msv[stencil(-1,1)] = sbeta_im1 * vbeta_jp1;
            G_msv += sbeta_im1 * vbeta_jp1 * mixted_sv * sqrt(di
            bc_var_max(S0, sgrid, Ns, i-1,
                    V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                    R0, divid, rho_sv,
                    tgrid, Nt, time_index, call_or_put, strik
            //msv[stencil(1,1)] = sbeta_ip1 * vbeta_jp1;
            G_msv += sbeta_ip1 * vbeta_jp1 * mixted_sv * sqrt(di
            bc_var_max(S0, sgrid, Ns, i+1,
                    V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                    R0, divid, rho_sv,
                    tgrid, Nt, time_index, call_or_put, strik
            msv[stencil(0,0)] =(
                                sbeta_i0 * vbeta_j0
                                + 1. * sbeta_im1 * vbeta_jp1 //
                                + 1. * sbeta_i0 * vbeta_jp1 // N
                                + 1. * sbeta_ip1 * vbeta_jp1 //
                                ) * mixted_sv * sqrt(diffusion_s
        }
    } // End of V=Vmax -> we do not have done i==Ns in j==Nv.
    if (i==Ns) // S=Smax
    {
        // Three cases :
        // j==0 -> mixted_sv = 0 -> Nothing to do.
        // 0<j<Nv -> Condition on S=Smax.
        // j==Nv -> Condition on S=Smax compatible with conditio
        if ((0<j) && (j<Nv))
        {
            Dsi = sgrid[i]-sgrid[i-1];
            Dsip1 = Dsi;
            Dvj = vgrid[j]-vgrid[j-1];
            Dvjp1 = Dvj;

            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));


            msv[stencil(-1,0)] = sbeta_im1 * vbeta_j0 * mixted_s
            //msv[stencil(1,0)] = sbeta_ip1 * vbeta_j0 * mixted_
            G_msv += sbeta_ip1 * vbeta_j0 * mixted_sv * sqrt(dif
            bc_spot_max(S0, sgrid, Ns, i,
                        V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                        R0, divid, rho_sv,
                        tgrid, Nt, time_index, call_or_put, stri
            msv[stencil(0,-1)] = sbeta_i0 * vbeta_jm1 * mixted_s
            msv[stencil(0,1)] = sbeta_i0 * vbeta_jp1 * mixted_sv
            msv[stencil(-1,-1)] = sbeta_im1 * vbeta_jm1 * mixted
            //msv[stencil(1,-1)] = sbeta_ip1 * vbeta_jm1 * mixte
            G_msv += sbeta_ip1 * vbeta_jm1 * mixted_sv * sqrt(di
            bc_spot_max(S0, sgrid, Ns, i,
                        V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                        R0, divid, rho_sv,
                        tgrid, Nt, time_index, call_or_put, stri
            msv[stencil(-1,1)] = sbeta_im1 * vbeta_jp1 * mixted_
            //msv[stencil(1,1)] = sbeta_ip1 * vbeta_jp1 * mixted
            G_msv += sbeta_ip1 * vbeta_jp1 * mixted_sv * sqrt(di
            bc_spot_max(S0, sgrid, Ns, i,
                        V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                        R0, divid, rho_sv,
                        tgrid, Nt, time_index, call_or_put, stri
            msv[stencil(0,0)] = (
                                sbeta_i0 * vbeta_j0
                                + 1. * sbeta_ip1 * vbeta_jm1 //
                                + 1. * sbeta_ip1 * vbeta_j0 //
                                + 1. * sbeta_ip1 * vbeta_jp1 //
                                ) * mixted_sv * sqrt(diffusion_
        }
    } // End of S=Smax -> we do not have done j==Nv in i==Ns. Do
    if ((i==Ns) && (j==Nv)) // Corner ! The two Neumann conditio
    {
        Dsi = sgrid[i]-sgrid[i-1];
        Dsip1 = Dsi;
        Dvj = vgrid[j]-vgrid[j-1];
```

```
Dvjp1 = Dvj;

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));


msv[stencil(-1,0)] = sbeta_im1 * vbeta_j0 * mixted_sv *
//msv[stencil(1,0)] = sbeta_ip1 * vbeta_j0 * mixted_sv *
G_msv += sbeta_ip1 * vbeta_j0 * mixted_sv * sqrt(diffusi
bc_spot_max(S0, sgrid, Ns, i,
            V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
            R0, divid, rho_sv,
            tgrid, Nt, time_index, call_or_put, strike);
msv[stencil(0,-1)] = sbeta_i0 * vbeta_jm1 * mixted_sv *
//msv[stencil(0,1)] = sbeta_i0 * vbeta_jp1 * mixted_sv *
G_msv += sbeta_i0 * vbeta_jp1 * mixted_sv * sqrt(diffusi
bc_var_max(S0, sgrid, Ns, i,
           V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
           R0, divid, rho_sv,
           tgrid, Nt, time_index, call_or_put, strike);
msv[stencil(-1,-1)] = sbeta_im1 * vbeta_jm1 * mixted_sv
//msv[stencil(1,-1)] = sbeta_ip1 * vbeta_jm1 * mixted_sv
G_msv += sbeta_ip1 * vbeta_jm1 * mixted_sv * sqrt(diffus
bc_spot_max(S0, sgrid, Ns, i,
            V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j-1
            R0, divid, rho_sv,
            tgrid, Nt, time_index, call_or_put, strike);
//msv[stencil(-1,1)] = sbeta_im1 * vbeta_jp1 * mixted_sv
G_msv += sbeta_im1 * vbeta_jp1 * mixted_sv * sqrt(diffus
bc_var_max(S0, sgrid, Ns, i,
           V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j+1,
           R0, divid, rho_sv,
           tgrid, Nt, time_index, call_or_put, strike);
//msv[stencil(1,1)] = sbeta_ip1 * vbeta_jp1 * mixted_sv
// !!!! Here COMPATIBILITY is OK !!!!
G_msv += sbeta_ip1 * vbeta_jp1 * mixted_sv * sqrt(diffus
bc_spot_max(S0, sgrid, Ns, i,
```

```
                                    V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
                                    R0, divid, rho_sv,
                                    tgrid, Nt, time_index, call_or_put, strike);
                        msv[stencil(0,0)] = (
                                            sbeta_i0 * vbeta_j0
                                            + 1. * sbeta_ip1 * vbeta_j0 // Neum
                                            + 1. * sbeta_i0 * vbeta_jp1 // Neum
                                            + 1. * sbeta_ip1 * vbeta_jm1 // Neu
                                            + 1. * sbeta_im1 * vbeta_jp1 // Neu
                                            + 1. * sbeta_ip1 * vbeta_jp1 // Neu
                                            ) * mixted_sv * sqrt(diffusion_s *
                    }
                }
            }

            // Central point for order_0
            MatrixA0[i][j][0] = msv[0] * SPDE * VPDE;
            MatrixA1[i][j][0] = order_0 * (SPDE/(SPDE+VPDE)) + cs[0] * SPDE + ds
            MatrixA2[i][j][0] = order_0 * (VPDE/(SPDE+VPDE)) + cv[0] * VPDE + dv
            //printf("MatrixA1[%d][%d][0] = %f\ n",i,j,MatrixA1[i][j][0]);

            for (st=1;st<11;st++)
            {
                MatrixA0[i][j][st] = msv[st] * SPDE * VPDE;
                MatrixA1[i][j][st] = cs[st] * SPDE + ds[st] * SPDE;
                // Be careful, convection_v is changed inside the building
                MatrixA2[i][j][st] = cv[st] * VPDE + dv[st] * VPDE;
                //printf("MatrixA2[%d][%d][%d] = %f\ n",i,j,st,MatrixA2[i][j][st
            }

            // Second members
            G0[i][j] = G_msv * SPDE * VPDE;
            G1[i][j] = G_cs * SPDE + G_ds * SPDE;
            // Be careful, convection_v is changed inside the building
            G2[i][j] = G_cv * VPDE + G_dv * VPDE;

        }
    }

    free(msv);
    free(dv);
```

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

static void compute_explicit_syslin_all_matrix(double coeff,
                                               double *sgrid, int Ns, double *vg
                                               double ***MatrixA0, double ***Mat
                                               double **G0nm1, double **G1nm1, d
                                               double **Unm1, double **Y0)
{

    int i, j, st;
    double val=0.;
    int istencil, jstencil;

    for (i=0; i<Ns+1; i++)
    {
        for (j=0; j<Nv+1; j++)
        {
            val = Unm1[i][j];
            for (st=0; st<11; st++)
            {
                if (it_exists_stencil(i, Ns, j, Nv, 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,
```

```
                                                    double *sgrid, int Ns, doubl
                                                    double ***MatrixA0, double *
                                                    double **G0nm1, double **G1n
                                                    double **Unm1, double **Y0,
{
    int i, j, st;
    double val;
    int istencil, jstencil;

    val=0.;

    for (j=0; j<Nv+1; j++)
    {
        for (i=0; i<Ns+1; i++)
        {
            val = Y0[i][j];
            for (st=0; st<11; st++)
            {
                if (it_exists_stencil(i, Ns, j, Nv, 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,
                                                    double *sgrid, int Ns, doubl
                                                    double ***MatrixA0, double *
                                                    double **G0, double **G1, do
                                                    double **rhs, double **lhs)
{
    int i, j;

    // Only points from i=0 to i=Ns.
    // Only points from j=0 to j=Nv.
```

```
PnlTridiagMat *Identity_Matrix;
PnlTridiagMat *Working_Matrix;
PnlVect *Entree;
PnlVect *Sortie;

Identity_Matrix = pnl_tridiag_mat_create_from_two_double(Ns+1, 1.0, 0.0); //
Working_Matrix = pnl_tridiag_mat_create(Ns+1);
Entree = pnl_vect_create(Ns+1);
Sortie = pnl_vect_create(Ns+1);

for (j=0; j<Nv+1; j++)
{
    // Build the matrix

    //pnl_tridiag_mat_set (Working_Matrix, 0, -1, MatrixA1[0][j][1]);
    pnl_tridiag_mat_set (Working_Matrix, 0,  0, MatrixA1[0][j][0]);
    pnl_tridiag_mat_set (Working_Matrix, 0,  1, MatrixA1[0][j][2]);
    for (i=0; i<Ns-1; i++)
    {
        pnl_tridiag_mat_set (Working_Matrix, i+1, -1, MatrixA1[i+1][j][1]);
        pnl_tridiag_mat_set (Working_Matrix, i+1,  0, MatrixA1[i+1][j][0]);
        pnl_tridiag_mat_set (Working_Matrix, i+1,  1, MatrixA1[i+1][j][2]);
    }
    pnl_tridiag_mat_set (Working_Matrix, Ns, -1, MatrixA1[Ns][j][1]);
    pnl_tridiag_mat_set (Working_Matrix, Ns,  0, MatrixA1[Ns][j][0]);
    //pnl_tridiag_mat_set (Working_Matrix, Ns,  1, MatrixA1[Ns][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<Ns+1; i++)
    {
        pnl_vect_set (Entree, i, rhs[i][j] + coeff * G1[i][j]);
    }

    pnl_tridiag_mat_syslin(Sortie, Working_Matrix, Entree);

    for (i=0; i<Ns+1; i++)
```

```
        {
            lhs[i][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,
                                                   double *sgrid, int Ns, double
                                                   double ***MatrixA0, double **
                                                   double **G0nm1, double **G1nm
                                                   double **Unm1, double **Y1, d
{
    int i, j, st;
    double val;
    int istencil, jstencil;

    val =0.;

    for (i=0; i<Ns+1; i++)
    {
        for (j=0; j<Nv+1; j++)
        {
            val = Y1[i][j];
            for (st=0; st<11; st++)
            {
                if (it_exists_stencil(i, Ns, j, Nv, 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,
                                                   double *sgrid, int Ns, double
                                                   double ***MatrixA0, double **
                                                   double **G0, double **G1, dou
                                                   double **rhs, double **lhs)
{
    int i, j;

    // Only points from i=0 to i=Ns.
    // Only points from j=0 to j=Nv.
    // Pentadiagonal matrix.
    PnlBandMat *Working_Matrix;
    PnlVect *Entree;
    PnlVect *Sortie;

    Entree = pnl_vect_create(Nv+1);
    Sortie = pnl_vect_create(Nv+1);

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

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

        // Build the vectors.
        for (j=0; j<Nv+1; 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<Nv+1; j++)
        {
            lhs[i][j] = pnl_vect_get (Sortie, j);
        }
        pnl_band_mat_free(&Working_Matrix);
    }

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

```
////////////////////////////////////////////////////////////////////////////
// Function to make one step in ADI
////////////////////////////////////////////////////////////////////////////
static void adi_step
(int forward_or_backward,
 double S0, double *sgrid, int Ns,
 double V0, double sigma_v, double alpha_v, double beta_v, double *vgrid, int Nv
 double R0, double divid, double rho_sv,
 double *tgrid, int Nt, int time_index,
 double theta, int scheme,
 int call_or_put, double strike, int am,
 // 2-vectors
 double **Unm1, double **Utmp, double **Y0, double **Y1, //double **Y2, //double
 double **G0nm1, double **G1nm1, double **G2nm1, //double **G3nm1,
 //double **G0n;
 double **G1n, double **G2n, //double **G3n,
 double **GSecondMember,
 double *VolSto,
 // Matrix (=3-vectors)
 double ***MatrixA0nm1, double ***MatrixA1nm1, double ***MatrixA2nm1, //double *
 //double ***MatrixA0n;
 double ***MatrixA1n, double ***MatrixA2n) //double ***MatrixA3n)
{
    int i, j;
    double deltat;

    // Time Step
    deltat=tgrid[time_index]-tgrid[time_index-1];

    // Initialization
    for (i=0; i<Ns+1; i++)
    {
        for (j=0; j<Nv+1; j++)
        {
            Utmp[i][j] = 0;
            Y0[i][j] = 0.;
            Y1[i][j] = 0.;
            //Y2[i][j] = 0.;
            //Y3[i][j] = 0.;
        }
```

```
}

    /*
     printf("In adi cycle, we are in the period %d et index_t_begin=%d.\ n",
     printf("It corresponds to the temporal interval [%f,%f[ contains %f.\ n
     //*/

    if (scheme==0) // Douglas scheme
    {
        //print_vector(Unm1, Ns, Nv, Nr);

        /*
        if (OPTI)
        {
            // Build second member values
            for(j=0;j<Nv+1;j++)
            {
                for(i=0;i<Ns+1;i++)
                {
                    G1n[i][j] = GSecondMember[i][j];
                    G1nm1[i][j] = GSecondMember[i][j];
                }
            }

        }
        else
        {
            */
            ///*

            //printf("Build matrix at step time_index-1 (in t-past=tau-futur
            // Compute the elements at time step time_index-1, which is in t
            // (except A0n and G0n which are not used, so they are erased by
        if (forward_or_backward==1)
        {
            build_all_matrix_forward(S0, sgrid, Ns,
                                     V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                                     R0, divid, rho_sv,
                                     tgrid, Nt, time_index-1, call_or_put, strike,
                                     VolSto,
                                     MatrixA0nm1, MatrixA1n, MatrixA2n,
```

```
                    G0nm1, G1n, G2n);
    //printf("Build matrix at step time_index (in t-present=tau-pres
// Compute the elements at time step time_index.
    build_all_matrix_forward(S0, sgrid, Ns,
                V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                R0, divid, rho_sv,
                tgrid, Nt, time_index, call_or_put, strike,
                VolSto,
                MatrixA0nm1, MatrixA1nm1, MatrixA2nm1,
                G0nm1, G1nm1, G2nm1);
}
else
{
    build_all_matrix(S0, sgrid, Ns,
                    V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                    R0, divid, rho_sv,
                    tgrid, Nt, time_index-1, call_or_put, strike,
                    VolSto,
                    MatrixA0nm1, MatrixA1n, MatrixA2n,
                    G0nm1, G1n, G2n);
    //printf("Build matrix at step time_index (in t-present=tau-pres
    // Compute the elements at time step time_index.
    build_all_matrix(S0, sgrid, Ns,
                    V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                    R0, divid, rho_sv,
                    tgrid, Nt, time_index, call_or_put, strike,
                    VolSto,
                    MatrixA0nm1, MatrixA1nm1, MatrixA2nm1,
                    G0nm1, G1nm1, G2nm1);

}
     //*/

//printf("Iteration %d\ n", time_index);
//printf("Matrix A1\ n");
//print_matrix(Ns,Nv,MatrixA1nm1);
//printf("Matrix A2\ n");
//print_matrix(Ns,Nv,MatrixA2nm1);

//printf("Compute Douglas scheme.\ n");
```

```
//printf("Y0 = U[n-1] + Dt (A0+A1+A2+A3)[n-1] * U[n-1] + Dt (G0+G1+G
//Y0 = U[n-1] + Dt (A0+A1+A2+A3)[n-1] * U[n-1] + Dt (G0+G1+G2+G3)[n-
compute_explicit_syslin_all_matrix(deltat,
                                   sgrid, Ns, vgrid, Nv,
                                   MatrixA0nm1, MatrixA1nm1, MatrixA
                                   G0nm1, G1nm1, G2nm1,
                                   Unm1, Y0);
//printf("Y0 = \ n");
//print_vector(Y0, Ns, Nv);
positivizing(Y0,Ns,Nv);

//printf("Utmp = Y0 - theta*Dt A1[n-1] * U[n-1] - theta*Dt G1[n-1].\
//Utmp = Y0 - theta*Dt A1[n-1] * U[n-1] - theta*Dt G1[n-1]
computation_explicit_syslin_spot_matrix(theta * deltat,
                                        sgrid, Ns, vgrid, Nv,
                                        MatrixA0nm1, MatrixA1nm1, Ma
                                        G0nm1, G1nm1, G2nm1,
                                        Unm1, Y0, Utmp);
//printf("Utmp = \ n");
//print_vector(Utmp, Ns, Nv);
positivizing(Utmp,Ns,Nv);

//printf("Y1 = (Id - theta*Dt A1[n])^-1 ( Utmp + theta*Dt G1[n] ).\
//Y1 = (Id - theta*Dt A1[n])^-1 ( Utmp + theta*Dt G1[n] )
computation_implicit_syslin_spot_matrix(theta * deltat,
                                        sgrid, Ns, vgrid, Nv,
                                        MatrixA0nm1, MatrixA1n, Matr
                                        G0nm1, G1n, G2n,
                                        Utmp, Y1);
//printf("Y1 = \ n");
//print_vector(Y1, Ns, Nv);

//printf("Utmp = Y1 - theta*Dt A2[n-1] * U[n-1] - theta*Dt G2[n-1].\
//Utmp = Y1 - theta*Dt A2[n-1] * U[n-1] - theta*Dt G2[n-1]
computation_explicit_syslin_var_matrix(theta * deltat,
                                       sgrid, Ns, vgrid, Nv,
                                       MatrixA0nm1, MatrixA1nm1, Mat
                                       G0nm1, G1nm1, G2nm1,
                                       Unm1, Y1, Utmp);
//printf("Utmp = \ n");
//print_vector(Utmp, Ns, Nv);
```

```
            positivizing(Utmp,Ns,Nv);

            //printf("Y2 = (Id - theta*Dt A2[n])^-1 ( Utmp + theta*Dt G2[n] ).\
            //Y2 = (Id - theta*Dt A2[n])^-1 ( Utmp + theta*Dt G2[n] )
            computation_implicit_syslin_var_matrix(theta * deltat,
                                                   sgrid, Ns, vgrid, Nv,
                                                   MatrixA0nm1, MatrixA1n, Matri
                                                   G0nm1, G1n, G2n,
                                                   Utmp, Unm1);
            // /!\  with Y2=Un which becomes Unm1 for the next loop.
            //printf("Unm1 = \ n");
            //print_vector(Unm1, Ns, Nv);

            //printf("Loop over time.\ n");
        }
        if (scheme==1) // Craig-Sneyd scheme
        {

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

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

        }
    for (i=0; i<Ns+1; i++)
    {
        for (j=0; j<Nv+1; j++)
        {
            Utmp[i][j] =  Unm1[i][j];
        }
    }
    //print_vector(Unm1, Ns, Nv);

}

////////////////////////////////////////////////////////////////////////////////
//Compute vol then price.
////////////////////////////////////////////////////////////////////////////////
```

```
static double compute_vol_then_price(double S0, double *sgrid, int Ns,
                                     double V0, double sigma_v, double alpha_v,
                                     double *vgrid, int Nv,
                                     double R0, double divid, double rho_sv,
                                     double *tgrid, int Nt,
                                     double theta,  int scheme,
                                     int call_or_put, double strike, int sig_t,
                                     double *ptprice, double *ptdelta)
{
    // Variables for loops.
    int i, j, st;
    int time_index;

    // Variables to compute price and delta.
    int IndexS, IndexV;
//    double price, delta;
    double Psum, PVsum;
    double price_sd_vd, price_sd_vu, price_su_vd, price_su_vu;

    double *VolSto;

    // 2-vectors
    double **P_new;
    double **P_old;
    double **P0;
    double **P1;
    //double **P2;
    //double **P3;

    double **TimeVolSto;




    double **G0nm1;
    double **G1nm1;
    double **G2nm1;
    //double **G3nm1;
    //double **G0n;
    double **G1n;
    double **G2n;
```

```
//double **G3n;
double **GSecondMember;

// Matrix (=3-vectors)
double ***MatrixA0nm1;
double ***MatrixA1nm1;
double ***MatrixA2nm1;
//double ***MatrixA3nm1;
//double ***MatrixA0n;
double ***MatrixA1n;
double ***MatrixA2n;
//double ***MatrixA3n;

// Memory allocations.
{

    VolSto = (double*) malloc((Ns+1) * sizeof(double));
    TimeVolSto = (double**) malloc((Nt+1) * sizeof(double));
    for (time_index = 0; time_index < Nt+1; time_index++)
    {
        TimeVolSto[time_index] = (double*) malloc((Ns+1) * sizeof(double));

    }

    // Memory allocation of 2-vectors.
    P_old = (double**) malloc((Ns+1) * sizeof(double*));
    P_new = (double**) malloc((Ns+1) * sizeof(double*));
    P0 = (double**) malloc((Ns+1) * sizeof(double*));
    P1 = (double**) malloc((Ns+1) * sizeof(double*));
    //Y2 = (double**) malloc((Ns+1) * sizeof(double*));
    //Y3 = (double**) malloc((Ns+1) * sizeof(double*));
    G0nm1 = (double**) malloc((Ns+1) * sizeof(double*));
    G1nm1 = (double**) malloc((Ns+1) * sizeof(double*));
    G2nm1 = (double**) malloc((Ns+1) * sizeof(double*));
    //G3nm1 = (double**) malloc((Ns+1) * sizeof(double*));
    //G0n = (double**) malloc((Ns+1) * sizeof(double*));
    G1n = (double**) malloc((Ns+1) * sizeof(double*));
    G2n = (double**) malloc((Ns+1) * sizeof(double*));
    //G3n = (double**) malloc((Ns+1) * sizeof(double*));
    for (i = 0; i < Ns+1; i++)
    {
```

```
    P_old[i] = (double*) malloc((Nv+1) * sizeof(double));
    P_new[i] = (double*) malloc((Nv+1) * sizeof(double));
    P0[i] = (double*) malloc((Nv+1) * sizeof(double));
    P1[i] = (double*) malloc((Nv+1) * sizeof(double));
    //Y2[i] = (double*) malloc((Nv+1) * sizeof(double));
    //Y3[i] = (double*) malloc((Nv+1) * sizeof(double));
    G0nm1[i] = (double*) malloc((Nv+1) * sizeof(double));
    G1nm1[i] = (double*) malloc((Nv+1) * sizeof(double));
    G2nm1[i] = (double*) malloc((Nv+1) * sizeof(double));
    //G3nm1[i] = (double*) malloc((Nv+1) * sizeof(double));
    //G0n[i] = (double*) malloc((Nv+1) * sizeof(double));
    G1n[i] = (double*) malloc((Nv+1) * sizeof(double));
    G2n[i] = (double*) malloc((Nv+1) * sizeof(double));
    //G3n[i] = (double*) malloc((Nv+1) * sizeof(double));
}

// Memory allocation of matrix (=3-vectors).
MatrixA0nm1 = (double***) malloc((Ns+1) * sizeof(double**));
MatrixA1nm1 = (double***) malloc((Ns+1) * sizeof(double**));
MatrixA2nm1 = (double***) malloc((Ns+1) * sizeof(double**));
//MatrixA3nm1 = (double***) malloc((Ns+1) * sizeof(double**));
//MatrixA0n = (double***) malloc((Ns+1) * sizeof(double**));
MatrixA1n = (double***) malloc((Ns+1) * sizeof(double**));
MatrixA2n = (double***) malloc((Ns+1) * sizeof(double**));
//MatrixA3n = (double***) malloc((Ns+1) * sizeof(double**));
for (i=0; i<Ns+1; i++)
{
    MatrixA0nm1[i] = (double**) malloc((Nv+1) * sizeof(double*));
    MatrixA1nm1[i] = (double**) malloc((Nv+1) * sizeof(double*));
    MatrixA2nm1[i] = (double**) malloc((Nv+1) * sizeof(double*));
    //MatrixA3nm1[i] = (double**) malloc((Nv+1) * sizeof(double*));
    //MatrixA0n[i] = (double**) malloc((Nv+1) * sizeof(double*));
    MatrixA1n[i] = (double**) malloc((Nv+1) * sizeof(double*));
    MatrixA2n[i] = (double**) malloc((Nv+1) * sizeof(double*));
    //MatrixA3n[i] = (double**) malloc((Nv+1) * sizeof(double*));
    for (j=0; j<Nv+1; j++)
    {
        MatrixA0nm1[i][j] = (double*) malloc(11 * sizeof(double));
        MatrixA1nm1[i][j] = (double*) malloc(11 * sizeof(double));
        MatrixA2nm1[i][j] = (double*) malloc(11 * sizeof(double));
        //MatrixA3nm1[i][j] = (double*) malloc(11 * sizeof(double));
```

```
            //MatrixA0n[i][j] = (double*) malloc(11 * sizeof(double));
            MatrixA1n[i][j] = (double*) malloc(11 * sizeof(double));
            MatrixA2n[i][j] = (double*) malloc(11 * sizeof(double));
            //MatrixA3n[i][j] = (double*) malloc(11 * sizeof(double));
        }
    }
}// End of memory allocations.


// Initialization.
for (i=0; i<Ns+1; i++)
{
    VolSto[i] = 0.;
    for (j=0; j<Nv+1; j++)
    {
        P_old[i][j] = 0;
        P_new[i][j] = 0;
        P0[i][j] = 0.;
        P1[i][j] = 0.;
        //Y2[i][j] = 0.;
        //Y3[i][j] = 0.;
        G0nm1[i][j] = 0.;
        G1nm1[i][j] = 0.;
        G2nm1[i][j] = 0.;
        //G3nm1[i][j] = 0.;
        //G0n[i][j] = 0.;
        G1n[i][j] = 0.;
        G2n[i][j] = 0.;
        //G3n[i][j] = 0.;
    }
}

for (i=0; i<Ns+1; i++)
{
    for (j=0; j<Nv+1; j++)
    {
        for (st=0; st<11; st++)
        {
            MatrixA0nm1[i][j][st] = 0.;
            MatrixA1nm1[i][j][st] = 0.;
            MatrixA2nm1[i][j][st] = 0.;
```

```
            //MatrixA3nm1[i][j][st] = 0.;
            //MatrixA0n[i][j][st] = 0.;
            MatrixA1n[i][j][st] = 0.;
            MatrixA2n[i][j][st] = 0.;
            //MatrixA3n[i][j][st] = 0.;
        }

    }
}

if (OPTI)
{
    GSecondMember = (double**) malloc((Ns+1) * sizeof(double*));
    for (i = 0; i < Ns+1; i++)
    {
        GSecondMember[i] = (double*) malloc((Nv+1) * sizeof(double));
    }
    for (i=0; i<Ns+1; i++)
    {
        for (j=0; j<Nv+1; j++)
        {
            GSecondMember[i][j]=0.;
        }
    }
    /*
    // Construction before loop. Using last time... (Nt, number_of_periods).
    build_all_matrix(S0, sgrid, Ns,
                     V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                     R0, divid, rho_sv,
                     tgrid, Nt, Nt-1,
                     VolSto,
                     MatrixA0nm1, MatrixA1n, MatrixA2n,
                     G0nm1, G1n, G2n);

    build_all_matrix(S0, sgrid, Ns,
                     V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                     R0, divid, rho_sv,
                     tgrid, Nt, Nt,
                     VolSto,
                     MatrixA0nm1, MatrixA1nm1, MatrixA2nm1,
                     G0nm1, G1nm1, G2nm1);
```

```
    */

    // Record time independent values.
    for(j=0;j<Nv+1;j++)
    {
        for(i=0;i<Ns+1;i++)
        {
            GSecondMember[i][j] = G1n[i][j];
        }
    }
}

// Terminal and initial values
for (i=0; i<Ns+1; i++) {
    for (j=0; j<Nv+1; j++) {
        P_old[i][j] = 0.;
    }
}
// Find the index in sgrid corresponding to the price S0 of the asset.
IndexS = lower_index(sgrid,Ns+1,S0);
// Find the index in vgrid corresponding to the variance V0 of the asset.
IndexV = lower_index(vgrid,Nv+1,V0);

P_old[IndexS][IndexV] = 1.;//1./((sgrid[IndexS+1]-sgrid[IndexS])*(vgrid[Inde

for (i=0; i<Ns+1; i++) {
        VolSto[i] = premia_local_vol(0,sgrid[i],sig_t)/sqrt(V0); // Sigma =
}

// Loop over the time
for (time_index=1; time_index<=Nt; ++time_index)
{
    //printf("Step %d VolSto\ n", time_index);
    //print_array(VolSto,Ns);
    //printf("Step %d P_old\ n", time_index);
    //print_vector(P_old,Ns,Nv);

    //printf("We are at time %d.\ n",time_index);
    // Make one ADI step
    adi_step(1,
             S0, sgrid, Ns,
```

```
            V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
            R0, divid, rho_sv,
            tgrid, Nt, time_index,
            theta, scheme, call_or_put, strike, sig_t,
            // 2-vectors
            P_old, P_new, P0, P1, //Y2, //Y3,
            G0nm1, G1nm1, G2nm1, //G3nm1, //G0n,
            G1n, G2n, //G3n,
            GSecondMember,
            VolSto,
            // Matrix (=3-vectors)
            MatrixA0nm1, MatrixA1nm1, MatrixA2nm1, //MatrixA3nm1, //MatrixA
            MatrixA1n, MatrixA2n); //MatrixA3n);
// Compute the stochastic volatility
for (i=0; i<Ns+1; i++) {
    // Compute first the conditional expectation (on VolSto array)

    Psum = 0.;
    PVsum = 0.;
    for (j=0; j<Nv+1; j++) {

        PVsum += P_new[i][j] * vgrid[j];
        Psum += P_new[i][j];
    }
    //printf("PVsum(%d) = %f, Psum(%d) = %f, Esp = %f / %f\ n",i+1, PVsu
    // Update VolSto
    VolSto[i] =premia_local_vol(tgrid[time_index],sgrid[i],sig_t)/sqrt(f
}
// Record VolSto
for (i=0; i<Ns+1; i++) {
    TimeVolSto[time_index][i] = VolSto[i];
    //printf("TimeVolSto(%d) = %f\ n",i, VolSto[i]);

}
// Record P_new in P_old
for (i=0; i<Ns+1; i++) {
    for (j=0; j<Nv+1; j++) {
        P_old[i][j] = fabs(P_new[i][j]);
    }
}
```

```
}//End of loop over time.

// Terminal and initial values
for (i=0; i<Ns+1; i++) {
    for (j=0; j<Nv+1; j++) {
        P_old[i][j] = MAX(call_or_put*(sgrid[i] - strike),0.0);
    }
    //printf("P_old[%d] = %f \ n",i, P_old[i][0]);
}

// Loop over the time
for (time_index=Nt; time_index>=1; --time_index)
{
    //printf("Step %d VolSto\ n", time_index);
    //print_array(VolSto,Ns);
    //printf("Step %d P_old\ n", time_index);
    //print_vector(P_old,Ns,Nv);

    //printf("We are at index time %d, at time %f.\ n",time_index, tgrid[tim
    // Make one ADI step


    /* if (am==1) */
    /* { */
    /*     // American condition = Unknowns >= payoff */
    /*     for (i=0; i<Ns+1; i++) */
    /*     { */
    /*         for (j=0; j<Nv+1; j++) */
    /*         { */
    /*                 if (call_or_put==1) //Call */
    /*                 { */
    /*                     P_old[i][j] = MAX(P_old[i][j],MAX(sgrid[i] - stri
    /*                 } */
    /*                 if (call_or_put==-1) //Put */
    /*                 { */
    /*                     P_old[i][j] = MAX(P_old[i][j],MAX(strike - sgrid[
    /*                 } */
    /*         } */
    /*     } */
    /* } */
```

```
for (i=0; i<Ns+1; i++) {
    // X#X#X#X#X#X#X
    // X#X#X#X#X#X#X
    // X#X#X#X#X#X#X
    // X#X#X#X#X#X#X
    VolSto[i] =  TimeVolSto[time_index][i];
    //VolSto[i] =  1.;
    // X#X#X#X#X#X#X
    // X#X#X#X#X#X#X
    // X#X#X#X#X#X#X
    // X#X#X#X#X#X#X
    // X#X#X#X#X#X#X
}

/* if (time_index==Nt)//((time_index)/((double)Nt) > 0.9) */
/* { */
/*     // Make one small step with Black-Scholes, since ADI not stable.

/*     for (i=0; i<Ns+1; i++) { */
/*         // For j>0, vol > 0 */
/*         for (j=1; j<Nv+1; j++) { */
/*             if (call_or_put==1) //Call */
/*                 pnl_cf_call_bs(sgrid[i], strike, tgrid[Nt]-tgrid[Nt-1
/*             if (call_or_put==-1) //Put */
/*                 pnl_cf_put_bs(sgrid[i], strike, tgrid[Nt]-tgrid[Nt-1]
/*             P_new[i][j] = fabs(*ptprice); */
/*             //pnl_cf_call_bs(sgrid[i], strike, tgrid[Nt]-tgrid[Nt-1],
/*             //printf("%d -> P_old[%d,%d] = %.16f\ n",time_index,i,j,P
/*         } */
/*         // For j=0, vol=0, price extrapolated */
/*         j=0; */
/*         { */
/*             if (call_or_put==1) //Call */
/*             { */
/*                 pnl_cf_call_bs(sgrid[i], strike, tgrid[Nt]-tgrid[Nt-1
/*                 P_new[i][j] = (vgrid[0]-vgrid[1])/(vgrid[2]-vgrid[1])
/*                 pnl_cf_call_bs(sgrid[i], strike, tgrid[Nt]-tgrid[Nt-1
/*                 P_new[i][j] += (vgrid[2]-vgrid[0])/(vgrid[2]-vgrid[1]
```

```
/*             } */
/*             if (call_or_put==-1) //Put */
/*             { */
/*                 pnl_cf_put_bs(sgrid[i], strike, tgrid[Nt]-tgrid[Nt-1]
/*                 P_new[i][j] = (vgrid[0]-vgrid[1])/(vgrid[2]-vgrid[1])
/*                 pnl_cf_put_bs(sgrid[i], strike, tgrid[Nt]-tgrid[Nt-1]
/*                 P_new[i][j] += (vgrid[2]-vgrid[0])/(vgrid[2]-vgrid[1]
/*             } */
/*             //printf("%d -> P_old[%d,%d] = %.16f\ n",time_index,i,j,P
/*             P_new[i][j] = fabs(P_new[i][j]); */
/*         } */
/*     } */
/* } */
/* else */
{
    adi_step(0,
             S0, sgrid, Ns,
             V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
             R0, divid, rho_sv,
             tgrid, Nt, time_index,
             theta, scheme, call_or_put, strike, sig_t,
             // 2-vectors
             P_old, P_new, P0, P1, //Y2, //Y3,
             G0nm1, G1nm1, G2nm1, //G3nm1, //G0n,
             G1n, G2n, //G3n,
             GSecondMember,
             VolSto,
             // Matrix (=3-vectors)
             MatrixA0nm1, MatrixA1nm1, MatrixA2nm1, //MatrixA3nm1, //Mat
             MatrixA1n, MatrixA2n); //MatrixA3n);
}


for (i=0; i<Ns+1; i++) {
    for (j=0; j<Nv+1; j++) {
        P_old[i][j] = fabs(P_new[i][j]);
        //P_old[i][j] = MAX(P_new[i][j],0.);
    }
    //printf("%d -> P_old[%d] = %f\ n",time_index,i,P_old[i][0]);
}
//printf("time_index = %d\ n",time_index);
```

```
    //print_vector(P_old, Ns, Nv);




}//End of loop over time.


// Index in the domain for the price.
// Find the index in sgrid corresponding to the price S0 of the asset.
IndexS = lower_index(sgrid,Ns+1,S0);
// Find the index in vgrid corresponding to the variance V0 of the asset.
IndexV = lower_index(vgrid,Nv+1,V0);

// First compute the delta (using price as temporary variable). We do an int
price_sd_vd = P_new[IndexS+1][IndexV];
price_sd_vu = P_new[IndexS+1][IndexV+1];
price_su_vd = P_new[IndexS+2][IndexV];
price_su_vu = P_new[IndexS+2][IndexV+1];

*ptprice=double_interpolation(price_sd_vd, price_sd_vu, price_su_vd, price_s
                              sgrid[IndexS+1], sgrid[IndexS+2],
                              vgrid[IndexV], vgrid[IndexV+1],
                              sgrid[IndexS+1], V0);

price_sd_vd = P_new[IndexS][IndexV];
price_sd_vu = P_new[IndexS][IndexV+1];
price_su_vd = P_new[IndexS+1][IndexV];
price_su_vu = P_new[IndexS+1][IndexV+1];

*ptdelta = (*ptprice - double_interpolation(price_sd_vd, price_sd_vu, price_
                                            sgrid[IndexS], sgrid[IndexS+1],
                                            vgrid[IndexV], vgrid[IndexV+1],
                                            sgrid[IndexS], V0)     )
//    /(S0*exp(sgrid[IndexS+1]) - S0*exp(sgrid[IndexS]));
    /(sgrid[IndexS+1] - sgrid[IndexS]);

// Next compute the price. We do an interpolation.
*ptprice = double_interpolation(price_sd_vd, price_sd_vu, price_su_vd, price
                                sgrid[IndexS], sgrid[IndexS+1],
```

```
                                    vgrid[IndexV], vgrid[IndexV+1],
                                    S0, V0);

// Memory desallocations.
{

    free(VolSto);
    for (time_index = 0; time_index < Nt+1; time_index++)
    {
        free(TimeVolSto[time_index]);
    }
    free(TimeVolSto);


    // Memory desallocation of matrix (=4-vectors).
    for (i=0; i<Ns+1; i++)
    {
        for (j=0; j<Nv+1; j++)
        {
            free(MatrixA0nm1[i][j]);
            free(MatrixA1nm1[i][j]);
            free(MatrixA2nm1[i][j]);
            //free(MatrixA3nm1[i][j]);
            //free(MatrixA0n[i][j]);
            free(MatrixA1n[i][j]);
            free(MatrixA2n[i][j]);
            //free(MatrixA3n[i][j]);
        }
        free(MatrixA0nm1[i]);
        free(MatrixA1nm1[i]);
        free(MatrixA2nm1[i]);
        //free(MatrixA3nm1[i]);
        //free(MatrixA0n[i]);
        free(MatrixA1n[i]);
        free(MatrixA2n[i]);
        //free(MatrixA3n[i]);
    }
    free(MatrixA0nm1);
    free(MatrixA1nm1);
    free(MatrixA2nm1);
    //free(MatrixA3nm1);
```

```
    //free(MatrixA0n);
    free(MatrixA1n);
    free(MatrixA2n);
    //free(MatrixA3n);

    // Memory desallocation of 3-vectors.
    for (i=0; i<Ns+1; i++)
    {
        free(P_old[i]);
        free(P_new[i]);
        free(P0[i]);
        free(P1[i]);
        //free(Y2[i]);
        //free(Y3[i])
        free(G0nm1[i]);
        free(G1nm1[i]);
        free(G2nm1[i]);
        //free(G3nm1[i]);
        //free(G0n[i]);
        free(G1n[i]);
        free(G2n[i]);
        //free(G3n[i]);
        free(GSecondMember[i]);
    }
    free(P_old);
    free(P_new);
    free(P0);
    free(P1);
    //free(Y2);
    //free(Y3)
    free(G0nm1);
    free(G1nm1);
    free(G2nm1);
    //free(G3nm1);
    //free(G0n);
    free(G1n);
    free(G2n);
    //free(G3n);
    free(GSecondMember);
}// End of memory desallocations.
```

```
    return 0.;
}

//premia_local_vol(j * delta, sij, sig_t)
int FDHoutLVSV(double R0,double divid,double S0,double maturity,double V0,double
{

  int call_or_put;
  double strike;
  // Numerical parameters
  double *sgrid, *vgrid, *tgrid;
  double Smax,Vmax,Sleft,Sright,Coeff_s,Coeff_v;
  int scheme=0;
  double theta=1.;
  int i;

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

  // Spot
    Smax = 5.*S0;
    Sleft = 0.8*S0;
    Sright = 1.2*S0;
    Coeff_s = S0/20.; // Environ entre 2 et 5 ou fraction de Ns/2
    // Variance
    Vmax = -MAX(-MAX(5.*V0,5.),-5.);
    Coeff_v = Vmax/500.;
    // Scheme numerical parameters.
    theta=1;//Theta schema
    scheme=0.;//Douglas Scheme

    ////////////////////////////////////
    // Compute sgrid, vgrid and tgrid. //
    ////////////////////////////////////
    // Memory allocation of 1-vectors.
    sgrid=(double *)malloc((Ns+1)*sizeof(double));
    vgrid=(double *)malloc((Nv+1)*sizeof(double));
```

```
    tgrid=(double *)malloc((Nt+1)*sizeof(double));
    grid_generation_spot(sgrid, Sleft, Sright, Smax, Ns, Coeff_s);
    if (S0_IN_GRID)
    {
        i = lower_index(sgrid, Ns, S0);
        sgrid[i] = S0;
    }

    grid_generation_variance(vgrid, Vmax, Nv, Coeff_v, V0);

    for (i=0; i<=Nt; i++)
        tgrid[i] =  (i * maturity)/(double)Nt;

    ////////////////////
    // Compute prices. //
    ////////////////////

       //Compute value
    compute_vol_then_price(S0, sgrid, Ns,
                V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
   R0,divid, rho_sv,
                tgrid, Nt,
   theta, scheme, call_or_put, strike,sig_t,
                ptprice,ptdelta);

    free(sgrid);
    free(vgrid);
    free(tgrid);


  return OK;
}

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


  r = log(1. + ptMod->R.Val.V_DOUBLE / 100.);
```

```
  divid = log(1. + ptMod->Divid.Val.V_DOUBLE / 100.);

  return  FDHoutLVSV(r,divid,ptMod->S0.Val.V_PDOUBLE,ptOpt->Maturity.Val.V_DATE
                     ptMod->SigmaLV.Val.V_ENUM.value,
                     Met->Par[0].Val.V_PINT,
                     Met->Par[1].Val.V_PINT,
                     Met->Par[2].Val.V_PINT,
 &(Met->Res[0].Val.V_DOUBLE),
                     &(Met->Res[1].Val.V_DOUBLE));

  return OK;
}



static int CHK_OPT(FD_HOUT_LVSV)(void *Opt, void *Mod)
{

  if ((strcmp(((Option *)Opt)->Name, "CallEuro") == 0) || (strcmp(((Option *)Opt
    return OK;

  return  WRONG;
}
#endif //PremiaCurrentVersion

static int MET(Init)(PricingMethod *Met, Option *Opt)
{
  //int type_generator;
  if (Met->init == 0)
    {
      Met->init = 1;

      Met->Par[0].Val.V_INT2 = 500;
      Met->Par[1].Val.V_INT2 = 100;
  Met->Par[2].Val.V_INT2 = 40;
    }

  return OK;
}
```

```
PricingMethod MET(FD_HOUT_LVSV) =
{
  "FD_HOUT_LVSV",
  { {"TimeStepNumber", PINT, {100}, ALLOW},
    {"S StepNumber", PINT, {100}, ALLOW},
{"V StepNumber", PINT, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(FD_HOUT_LVSV),
  { {"Price", DOUBLE, {100}, FORBID},
    {"Delta", DOUBLE, {100}, FORBID} ,
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(FD_HOUT_LVSV),
  CHK_mc,
  MET(Init)
};
```