Help

```
#include  "
href../../../mod/hes1d/hes1d_stda/hes1d_stda_h_src.pdfhes1d_stda.h"
#include "pnl/pnl_mathtools.h"
#include "pnl/pnl_tridiag_matrix.h"
#include "pnl/pnl_band_matrix.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2017+2) //The "#els
static int CHK_OPT(FD_GMWB_HES)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(FD_GMWB_HES)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else
// This file contains all the code for the pricing of GMWB product under the Hes
// Heston = Black-Scholes process for asset and CIR process for stochastic volat
// It uses a finite difference scheme to simulate the associated PDE.
// Moreover an ADI-scheme is used to accelerate the computation.
// The model for GMWB is described in Chen-Forsyth's paper.

//static int FD_gmwb_bs(double w0_fixed, double t_fixed, double r_fixed, double

//////////
// 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_guarantee(double Aleft, double Aright, double Amax,
{
    int i;
    double deltaxi;
    if (Na>0)
        deltaxi = Amax/Na;
    else
        deltaxi = 0.;

    // Definition of uniform grid.
    for (i=0; i<=Na; i++)
    {
        agrid[i] = 0 + i * deltaxi;
    }
}
```

```
static void grid_generation_sub_account(double Wleft, double Wright, double Wmax
{
    int i;
    double Ximin;
    double Xiint;
    double Ximax;
    double deltaxi;

    Ximin = asinh1(-Wleft/Coeff_w);
    Xiint = (Wright-Wleft)/Coeff_w;
    Ximax = Xiint + asinh1((Wmax-Wright)/Coeff_w);
    deltaxi = (Ximax-Ximin)/Nw;

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

    // Definition of the sub-account grid with the uniform grid Xi.
    wgrid[0] = 0;
    for (i=1; i<=Nw; i++)
    {
        if (wgrid[i]<0)
        {
            wgrid[i] = Wleft+Coeff_w*sinh(wgrid[i]);
        }
        else
        {
            if (wgrid[i]<=Xiint)
            {
                wgrid[i] = Wleft+Coeff_w*wgrid[i];
            }
            else
            {
                wgrid[i] = Wright+Coeff_w*sinh(wgrid[i]-Xiint);
            }
        }
    }
}
```

```
static void grid_generation_variance(double Vmax, double Center_v, double Coeff_
{
    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]);
    }

    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_wd_vd, double value_wd_vu,
                                   double value_wu_vd, double value_wu_vu,
                                   double grid_wd, double grid_wu,
                                   double grid_vd, double grid_vu,
                                   double w, double v)
{
    double value_wd,value_wu;
```

```
    value_wd = interpolation (grid_vd, value_wd_vd, grid_vu, value_wd_vu, v);
    value_wu = interpolation (grid_vd, value_wu_vd, grid_vu, value_wu_vu, v);
    return interpolation (grid_wd, value_wd, grid_wu, value_wu, w);
}

static int it_exists_stencil(int i, int Nw, int j, int Nv, int stencil)
{
    // We use i from 0 to Nw, 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<Nw)&& (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==Nw)
```

```
        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;}
}

//////////
// Functions to manage boundary conditions.
//////////
```

```
static double bc_w_min(double alpha_g,
                       double alpha_m, double Gr, double kappa,
                       //double A0, double *agrid, int Na, int ia,
                       double W0, double *wgrid, int Nw, int iw,
                       double V0, double sigma_v, double alpha_v, double beta_v,
                       double R0, double rho_wv,
                       double *tgrid, int Nt, int time_index, int period)
{
    // dt U = sigma_v^2 V/2 dvv U + alpha_v (beta_v - V) dv U - r U
    // + 0 dww U + 0 dw U + 0 dwv U + alpha_m W
    // No necessary condition -> Neumann = 0.

    return 0.;
}

static double bc_w_max(double alpha_g,
                       double alpha_m, double Gr, double kappa,
                       //double A0, double *agrid, int Na, int ia,
                       double W0, double *wgrid, int Nw, int iw,
                       double V0, double sigma_v, double alpha_v, double beta_v,
                       double R0, double rho_wv,
                       double *tgrid, int Nt, int time_index, int period)
{
    // dt U = WW V/2 dww U + (r-alpha_g) W dw U + rho_wv sigma_v W V dwv U
    // + sigma_v^2 V/2 dvv U + alpha_v (beta_v - V) dv U - r U + alpha_m W
    // Easy boundary condition -> Neumann = 0.

    return 0.;
}

/*
 static double bc_var_min(double alpha_g,
 double alpha_m, double Gr, double kappa,
 //double A0, double *agrid, int Na, int ia,
 double W0, double *wgrid, int Nw, int iw,
 double V0, double sigma_v, double alpha_v, double beta_v, double *vgrid, int Nv
 double R0, double rho_wv,
 double *tgrid, int Nt, int time_index, int period)
 {
 // Neumann = 0.
 return 0.;
```

```
 }
 //*/

static double bc_var_max(double alpha_g,
                         double alpha_m, double Gr, double kappa,
                         //double A0, double *agrid, int Na, int ia,
                         double W0, double *wgrid, int Nw, int iw,
                         double V0, double sigma_v, double alpha_v, double beta_
                         double R0, double rho_wv,
                         double *tgrid, int Nt, int time_index, int period)
{
    // Neumann = 0.
    return 0.;
}

static double supplementary_terms(double alpha_m, double Gr, double kappa,
                                  double *tgrid, int Nt, int time_index, int per
{
    return alpha_m;
}

//////////
// Functions to build matrix and solve systems.
//////////
static void build_all_matrix(double alpha_g,
                             double alpha_m, double Gr, double kappa,
                             //double A0, double *agrid, int Na,
                             double W0, double *wgrid, int Nw,
                             double V0, double sigma_v, double alpha_v, double b
                             double R0, double rho_wv,
                             double *tgrid, int Nt, int time_index, int period,
                             double ***Matrix0, double ***Matrix1, double ***Mat
                             double **G0, double **G1, double **G2)
{
    double actualwpoint;
    double actualvpoint;
    double actualrpoint; // To unify notations.

    double Dwi, Dwip1;
    double Dvjm1, Dvj, Dvjp1, Dvjp2;
```

```
double convection_w, diffusion_w;
double convection_v, diffusion_v;
double order_0, mixted_wv;

double *cw;
double *dw;
double *cv;
double *dv;
double *mwv;

// Coefficients
//    double walpha_im2, walpha_im1, walpha_i0;
double wbeta_im1, wbeta_i0, wbeta_ip1;
//    double wgamma_i0, wgamma_ip1, wgamma_ip2;
//    double wdelta_im1, wdelta_i0, wdelta_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;

cw=(double*)malloc(11*sizeof(double));
dw=(double*)malloc(11*sizeof(double));
cv=(double*)malloc(11*sizeof(double));
dv=(double*)malloc(11*sizeof(double));
mwv=(double*)malloc(11*sizeof(double));

double G_cw, G_dw, G_cv, G_dv, G_mwv;

for(j=0;j<Nv+1;j++)
{
    for(i=0;i<Nw+1;i++)
    {
        for(st=0;st<11;st++)
        {
            cw[st]=0.;
            dw[st]=0.;
            cv[st]=0.;
            dv[st]=0.;
```

```
    mwv[st]=0.;
}
G_cw=0.;
G_dw=0.;
G_cv=0.;
G_dv=0.;
G_mwv=0.;

actualwpoint = wgrid[i];
actualvpoint = vgrid[j];
actualrpoint = R0;

convection_w = (actualrpoint - alpha_g) * actualwpoint;
diffusion_w = actualwpoint * actualwpoint * actualvpoint/2.0;
convection_v = alpha_v * (beta_v - actualvpoint); //convection_v = k
diffusion_v = sigma_v * sigma_v * actualvpoint/2.0;
order_0 = -actualrpoint;
mixted_wv = rho_wv * sigma_v * actualwpoint * actualvpoint;

// Method for boundary conditions.
// Compute all the ?grid_step for the finite difference scheme.
// Call bc_?_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 account g
// Compute Dwip1, copy it in Dwi.
// Call bc_w_min at point i-1.
// Suppress dw[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 dw[stencil(0,1)]
// Modify the diagonal of the matrix with diffusion coefficient.



//////////////////////
```

```
// Diffusion and Convection W
////////////////////
//printf("Diffusion and Convection W.\ n");
{
    if (i==0) // S=Smin -> Neumann
    {
        Dwip1 = wgrid[i+1]-wgrid[i];
        Dwi = Dwip1;
        //dw[stencil(-1,0)]=2.0/(Dwi*(Dwi+Dwip1));
        dw[stencil(0,0)]=-2.0/(Dwi*Dwip1) + 2.0/(Dwi*(Dwi+Dwip1));
        dw[stencil(1,0)]=2.0/(Dwip1*(Dwi+Dwip1));
        G_dw += 2.0/(Dwi*(Dwi+Dwip1)) * bc_w_min(alpha_g,
                                                 alpha_m, Gr, kappa,
                                                 //A0, agrid, Na, k,
                                                 W0, wgrid, Nw, i,
                                                 V0, sigma_v, alpha_
                                                 vgrid, Nv, j,
                                                 R0, rho_wv,
                                                 tgrid, Nt, time_ind

        //cw[stencil(-1,0)]=-Dwip1/(Dwi*(Dwi+Dwip1));
        cw[stencil(0,0)]=(Dwip1-Dwi)/(Dwi*Dwip1) - Dwip1/(Dwi*(Dwi+D
        cw[stencil(1,0)]=Dwi/(Dwip1*(Dwi+Dwip1));
        G_cw += -Dwip1/(Dwi*(Dwi+Dwip1)) * bc_w_min(alpha_g,
                                                    alpha_m, Gr, kap
                                                    //A0, agrid, Na,
                                                    W0, wgrid, Nw, i
                                                    V0, sigma_v, alp
                                                    vgrid, Nv, j,
                                                    R0, rho_wv,
                                                    tgrid, Nt, time_
    }
    else
    {
        if (i==Nw) // S=Smax -> Neumann
        {
            Dwi = wgrid[i]-wgrid[i-1];
            Dwip1 = Dwi;
            dw[stencil(-1,0)]=2.0/(Dwi*(Dwi+Dwip1));
            dw[stencil(0,0)]=-2.0/(Dwi*Dwip1) + 2.0/(Dwip1*(Dwi+Dwip
            //ds[stencil(1,0)]=2.0/(Dwip1*(Dwi+Dwip1));
```

```
                G_dw = 2.0/(Dwip1*(Dwi+Dwip1)) * bc_w_max(alpha_g,
                                                          alpha_m, Gr, k
                                                          //A0, agrid, N
                                                          W0, wgrid, Nw,
                                                          V0, sigma_v, a
                                                          vgrid, Nv, j,
                                                          R0, rho_wv,
                                                          tgrid, Nt, tim

                cw[stencil(-1,0)]=-Dwip1/(Dwi*(Dwi+Dwip1));
                cw[stencil(0,0)]=(Dwip1-Dwi)/(Dwi*Dwip1) + Dwi/(Dwip1*(D
                //cw[stencil(1,0)]=Dwi/(Dwip1*(Dwi+Dwip1));
                G_cw += Dwi/(Dwip1*(Dwi+Dwip1)) * bc_w_max(alpha_g,
                                                           alpha_m, Gr,
                                                           //A0, agrid,
                                                           W0, wgrid, Nw
                                                           V0, sigma_v,
                                                           vgrid, Nv, j,
                                                           R0, rho_wv,
                                                           tgrid, Nt, ti
            }
            else
            {
                Dwi = wgrid[i]-wgrid[i-1];
                Dwip1 = wgrid[i+1]-wgrid[i];
                dw[stencil(-1,0)]=2.0/(Dwi*(Dwi+Dwip1));
                dw[stencil(0,0)]=-2.0/(Dwi*Dwip1);
                dw[stencil(1,0)]=2.0/(Dwip1*(Dwi+Dwip1));

                cw[stencil(-1,0)]=-Dwip1/(Dwi*(Dwi+Dwip1));
                cw[stencil(0,0)]=(Dwip1-Dwi)/(Dwi*Dwip1);
                cw[stencil(1,0)]=Dwi/(Dwip1*(Dwi+Dwip1));
            }
        }
    }
    ////////////////////
    // 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(0,0)]=-(2.0*Dvjp1+Dvjp2)/(Dvjp1*(Dvjp1+Dvjp2));
    cv[stencil(0,1)]=(Dvjp1+Dvjp2)/(Dvjp1*Dvjp2);
    cv[stencil(0,2)]=-Dvjp1/(Dvjp2*(Dvjp1+Dvjp2));
}
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));
        dv[stencil(0,0)]=-2.0/(Dvj*Dvjp1) + 2.0/(Dvjp1*(Dvj+Dvjp
        //dv[stencil(0,1)]=2.0/(Dvjp1*(Dvj+Dvjp1));
        G_dv += 2.0/(Dvjp1*(Dvj+Dvjp1)) * bc_var_max(alpha_g,
                                                     alpha_m, Gr
                                                     //A0, agrid
                                                     W0, wgrid,
                                                     V0, sigma_v
                                                     vgrid, Nv,
                                                     R0, rho_wv,
                                                     tgrid, Nt,

        // V=Vmax -> Backward for convection.
        cv[stencil(0,0)]=(Dvjm1+2.0*Dvj)/(Dvj*(Dvjm1+Dvj));
        cv[stencil(0,-1)]=-(Dvjm1+Dvj)/(Dvjm1*Dvj);
        cv[stencil(0,-2)]=Dvj/(Dvjm1*(Dvjm1+Dvj));
    }
    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));
                dv[stencil(0,0)]=-2.0/(Dvj*Dvjp1);
                dv[stencil(0,1)]=2.0/(Dvjp1*(Dvj+Dvjp1));

                // If 1 <= j <= Nv-1 -> scheme for convection depends on
                if ((convection_v<0) && (j>1)) // var > beta_v and j>1 -
                {
                    Dvjm1 = vgrid[j-1]-vgrid[j-2];

                    cv[stencil(0,-2)]=Dvj/(Dvjm1*(Dvjm1+Dvj));
                    cv[stencil(0,-1)]=-(Dvjm1+Dvj)/(Dvjm1*Dvj);
                    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));
                    cv[stencil(0,0)]=(Dvjp1-Dvj)/(Dvj*Dvjp1);
                    cv[stencil(0,1)]=Dvj/(Dvjp1*(Dvj+Dvjp1));
                }
            }
        }
    }

    // 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 W
        // ->> no problem at W=Wmin since mixted_wv = 0.
        // ->> no problem at W=Wmax since also Neumann on W independent
        // if W=Wmax, we impose Neumann on W independent of V
        // ->> no problem at V=Vmin since mixted_wv = 0.
        // ->> no problem at V=Vmax since also Neumann on V independent

        if ((0<i) && (i<Nw) && (0<j) && (j<Nv)) // Strict interior 0<i<N
        {
            Dwi = wgrid[i]-wgrid[i-1];
```

```
        Dwip1 = wgrid[i+1]-wgrid[i];
        Dvj = vgrid[j]-vgrid[j-1];
        Dvjp1 = vgrid[j+1]-vgrid[j];

        wbeta_im1 = -Dwip1/(Dwi*(Dwi+Dwip1));
        wbeta_i0 = (Dwip1-Dwi)/(Dwi*Dwip1);
        wbeta_ip1 = Dwi/(Dwip1*(Dwi+Dwip1));
        vbeta_jm1 = -Dvjp1/(Dvj*(Dvj+Dvjp1));
        vbeta_j0 = (Dvjp1-Dvj)/(Dvj*Dvjp1);
        vbeta_jp1 = Dvj/(Dvjp1*(Dvj+Dvjp1));

        mwv[stencil(0,0)] = wbeta_i0 * vbeta_j0;
        mwv[stencil(-1,0)] = wbeta_im1 * vbeta_j0;
        mwv[stencil(1,0)] = wbeta_ip1 * vbeta_j0;
        mwv[stencil(0,-1)] = wbeta_i0 * vbeta_jm1;
        mwv[stencil(0,1)] = wbeta_i0 * vbeta_jp1;
        mwv[stencil(-1,-1)] = wbeta_im1 * vbeta_jm1;
        mwv[stencil(1,-1)] = wbeta_ip1 * vbeta_jm1;
        mwv[stencil(-1,1)] = wbeta_im1 * vbeta_jp1;
        mwv[stencil(1,1)] = wbeta_ip1 * vbeta_jp1;
    }
    else // i==0 or i==Ns or j==0 or j==Nv
    {
        if (j==Nv) // V=Vmax
        {
            // Three cases :
            // i==0 -> mixted_wv = 0 -> Nothing to do.
            // 0<i<Nw -> Condition on V=Vmax.
            // i==Nw -> Condition on V=Vmax compatible with conditio
            if ((0<i) && (i<Nw))
            {
                Dwi = wgrid[i]-wgrid[i-1];
                Dwip1 = wgrid[i+1]-wgrid[i];
                Dvj = vgrid[j]-vgrid[j-1];
                Dvjp1 = Dvj;

                wbeta_im1 = -Dwip1/(Dwi*(Dwi+Dwip1));
                wbeta_i0 = (Dwip1-Dwi)/(Dwi*Dwip1);
                wbeta_ip1 = Dwi/(Dwip1*(Dwi+Dwip1));
                vbeta_jm1 = -Dvjp1/(Dvj*(Dvj+Dvjp1));
                vbeta_j0 = (Dvjp1-Dvj)/(Dvj*Dvjp1);
```

```
            vbeta_jp1 = Dvj/(Dvjp1*(Dvj+Dvjp1));


            mwv[stencil(-1,0)] = wbeta_im1 * vbeta_j0;
            mwv[stencil(1,0)] = wbeta_ip1 * vbeta_j0;
            mwv[stencil(0,-1)] = wbeta_i0 * vbeta_jm1;
            //mwv[stencil(0,1)] = wbeta_i0 * vbeta_jp1;
            G_mwv += wbeta_i0 * vbeta_jp1 * bc_var_max(alpha_g,
                                                       alpha_m,
                                                       //A0, agr
                                                       W0, wgrid
                                                       V0, sigma
                                                       vgrid, Nv
                                                       R0, rho_w
                                                       tgrid, Nt
            mwv[stencil(-1,-1)] = wbeta_im1 * vbeta_jm1;
            mwv[stencil(1,-1)] = wbeta_ip1 * vbeta_jm1;
            //mwv[stencil(-1,1)] = wbeta_im1 * vbeta_jp1;
            G_mwv += wbeta_im1 * vbeta_jp1 * bc_var_max(alpha_g,
                                                        alpha_m,
                                                        //A0, ag
                                                        W0, wgri
                                                        V0, sigm
                                                        vgrid, N
                                                        R0, rho_
                                                        tgrid, N
            //msv[stencil(1,1)] = wbeta_ip1 * vbeta_jp1;
            G_mwv += wbeta_ip1 * vbeta_jp1 * bc_var_max(alpha_g,
                                                        alpha_m,
                                                        //A0, ag
                                                        W0, wgri
                                                        V0, sigm
                                                        vgrid, N
                                                        R0, rho_
                                                        tgrid, N
            mwv[stencil(0,0)] = wbeta_i0 * vbeta_j0
            + 1. * wbeta_im1 * vbeta_jp1 // Neumann
            + 1. * wbeta_i0 * vbeta_jp1 // Neumann
            + 1. * wbeta_ip1 * vbeta_jp1; // Neumann
        }
    } // End of V=Vmax -> we do not have done i==Nw in j==Nv.
```

```
if (i==Nw) // W=Wmax
{
    // Three cases :
    // j==0 -> mixted_wv = 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))
    {
        Dwi = wgrid[i]-wgrid[i-1];
        Dwip1 = Dwi;
        Dvj = vgrid[j]-vgrid[j-1];
        Dvjp1 = Dvj;

        wbeta_im1 = -Dwip1/(Dwi*(Dwi+Dwip1));
        wbeta_i0 = (Dwip1-Dwi)/(Dwi*Dwip1);
        wbeta_ip1 = Dwi/(Dwip1*(Dwi+Dwip1));
        vbeta_jm1 = -Dvjp1/(Dvj*(Dvj+Dvjp1));
        vbeta_j0 = (Dvjp1-Dvj)/(Dvj*Dvjp1);
        vbeta_jp1 = Dvj/(Dvjp1*(Dvj+Dvjp1));


        mwv[stencil(-1,0)] = wbeta_im1 * vbeta_j0;
        //mwv[stencil(1,0)] = wbeta_ip1 * vbeta_j0;
        G_mwv += wbeta_ip1 * vbeta_j0 * bc_w_max(alpha_g,
                                                 alpha_m, Gr
                                                 //A0, agrid
                                                 W0, wgrid,
                                                 V0, sigma_v
                                                 vgrid, Nv,
                                                 R0, rho_wv,
                                                 tgrid, Nt,
        mwv[stencil(0,-1)] = wbeta_i0 * vbeta_jm1;
        mwv[stencil(0,1)] = wbeta_i0 * vbeta_jp1;
        mwv[stencil(-1,-1)] = wbeta_im1 * vbeta_jm1;
        //mwv[stencil(1,-1)] = wbeta_ip1 * vbeta_jm1;
        G_mwv += wbeta_ip1 * vbeta_jm1 * bc_w_max(alpha_g,
                                                  alpha_m, G
                                                  //A0, agri
                                                  W0, wgrid,
                                                  V0, sigma_
                                                  vgrid, Nv,
```

```
                                                    R0, rho_wv
                                                    tgrid, Nt,
        mwv[stencil(-1,1)] = wbeta_im1 * vbeta_jp1;
        //mwv[stencil(1,1)] = wbeta_ip1 * vbeta_jp1;
        G_mwv += wbeta_ip1 * vbeta_jp1 * bc_w_max(alpha_g,
                                                    alpha_m, G
                                                    //A0, agri
                                                    W0, wgrid,
                                                    V0, sigma_
                                                    vgrid, Nv,
                                                    R0, rho_wv
                                                    tgrid, Nt,
        mwv[stencil(0,0)] = wbeta_i0 * vbeta_j0
        + 1. * wbeta_ip1 * vbeta_jm1 // Neumann
        + 1. * wbeta_ip1 * vbeta_j0 // Neumann
        + 1. * wbeta_ip1 * vbeta_jp1; // Neumann
    }
} // End of W=Wmax -> we do not have done j==Nv in i==Nw. Do
if ((i==Nw) && (j==Nv)) // Corner ! The two Neumann conditio
{
    Dwi = wgrid[i]-wgrid[i-1];
    Dwip1 = Dwi;
    Dvj = vgrid[j]-vgrid[j-1];
    Dvjp1 = Dvj;

    wbeta_im1 = -Dwip1/(Dwi*(Dwi+Dwip1));
    wbeta_i0 = (Dwip1-Dwi)/(Dwi*Dwip1);
    wbeta_ip1 = Dwi/(Dwip1*(Dwi+Dwip1));
    vbeta_jm1 = -Dvjp1/(Dvj*(Dvj+Dvjp1));
    vbeta_j0 = (Dvjp1-Dvj)/(Dvj*Dvjp1);
    vbeta_jp1 = Dvj/(Dvjp1*(Dvj+Dvjp1));


    mwv[stencil(-1,0)] = wbeta_im1 * vbeta_j0;
    //mwv[stencil(1,0)] = wbeta_ip1 * vbeta_j0;
    G_mwv += wbeta_ip1 * vbeta_j0 * bc_w_max(alpha_g,
                                                alpha_m, Gr, ka
                                                //A0, agrid, Na
                                                W0, wgrid, Nw,
                                                V0, sigma_v, al
                                                vgrid, Nv, j,
```

```
                                                R0, rho_wv,
                                                tgrid, Nt, time
mwv[stencil(0,-1)] = wbeta_i0 * vbeta_jm1;
//msv[stencil(0,1)] = wbeta_i0 * vbeta_jp1;
G_mwv += wbeta_i0 * vbeta_jp1 * bc_var_max(alpha_g,
                                            alpha_m, Gr,
                                            //A0, agrid,
                                            W0, wgrid, Nw
                                            V0, sigma_v,
                                            vgrid, Nv, j,
                                            R0, rho_wv,
                                            tgrid, Nt, ti
mwv[stencil(-1,-1)] = wbeta_im1 * vbeta_jm1;
//mwv[stencil(1,-1)] = wbeta_ip1 * vbeta_jm1;
G_mwv += wbeta_ip1 * vbeta_jm1 * bc_w_max(alpha_g,
                                          alpha_m, Gr, k
                                          //A0, agrid, N
                                          W0, wgrid, Nw,
                                          V0, sigma_v, a
                                          vgrid, Nv, j-1
                                          R0, rho_wv,
                                          tgrid, Nt, tim
//mwv[stencil(-1,1)] = wbeta_im1 * vbeta_jp1;
G_mwv += wbeta_im1 * vbeta_jp1 * bc_var_max(alpha_g,
                                            alpha_m, Gr,
                                            //A0, agrid,
                                            W0, wgrid, N
                                            V0, sigma_v,
                                            vgrid, Nv, j
                                            R0, rho_wv,
                                            tgrid, Nt, t
//mwv[stencil(1,1)] = wbeta_ip1 * vbeta_jp1; !!!! Here C
G_mwv += wbeta_ip1 * vbeta_jp1 * bc_w_max(alpha_g,
                                          alpha_m, Gr, k
                                          //A0, agrid, N
                                          W0, wgrid, Nw,
                                          V0, sigma_v, a
                                          vgrid, Nv, j,
                                          R0, rho_wv,
                                          tgrid, Nt, tim
mwv[stencil(0,0)] = wbeta_i0 * vbeta_j0
```

```
                        + 1. * wbeta_ip1 * vbeta_j0 // Neumann
                        + 1. * wbeta_i0 * vbeta_jp1 // Neumann
                        + 1. * wbeta_ip1 * vbeta_jm1 // Neumann
                        + 1. * wbeta_im1 * vbeta_jp1 // Neumann
                        + 1. * wbeta_ip1 * vbeta_jp1; // Neumann
                    }
                }
            }

            // Central point for order_0
            Matrix0[i][j][0] = mixted_wv * mwv[0];
            Matrix1[i][j][0] = order_0 * 0.5 + convection_w * cw[0] + diffusion_
            Matrix2[i][j][0] = order_0 * 0.5 + convection_v * cv[0] + diffusion_

            for (st=1;st<11;st++)
            {
                Matrix0[i][j][st] = mixted_wv * mwv[st];
                Matrix1[i][j][st] = convection_w * cw[st] + diffusion_w * dw[st]
                Matrix2[i][j][st] = convection_v * cv[st] + diffusion_v * dv[st]
            }

            // Second members
            G0[i][j] = mixted_wv * G_mwv;
            G1[i][j] = convection_w * G_cw + diffusion_w * G_dw;
            G2[i][j] = convection_v * G_cv + diffusion_v * G_dv;

            G1[i][j] += supplementary_terms(alpha_m, Gr, kappa, tgrid, Nt, time_

        }
    }

    free(mwv);
    free(dv);
    free(cv);
    free(dw);
    free(cw);
}

static void compute_explicit_syslin_all_matrix(double coeff,
                                               double *wgrid, int Nw, double *vg
                                               double ***Matrix0, double ***Matr
```

```
                                                    double **G0nm1, double **G1nm1, d
                                                    double **Unm1, double **Y0)
{

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

    for (i=0; i<Nw+1; i++)
    {
        for (j=0; j<Nv+1; j++)
        {
            val = Unm1[i][j];
            for (st=0; st<11; st++)
            {
                if (it_exists_stencil(i, Nw, j, Nv, st))
                {
                    // If the point exists.
                    point_of_stencil(i, j, st, &istencil, &jstencil);
                    val += coeff * Matrix0[i][j][st] * Unm1[istencil][jstencil];
                    val += coeff * Matrix1[i][j][st] * Unm1[istencil][jstencil];
                    val += coeff * Matrix2[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 *wgrid, int Nw, doubl
                                                    double ***Matrix0, 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<Nw+1; i++)
        {
            val = Y0[i][j];
            for (st=0; st<11; st++)
            {
                if (it_exists_stencil(i, Nw, j, Nv, st))
                {
                    // If the point exists.
                    point_of_stencil(i, j, st, &istencil, &jstencil);
                    val += -coeff * Matrix1[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 *wgrid, int Nw, doubl
                                                    double ***Matrix0, double **
                                                    double **G0, double **G1, do
                                                    double **rhs, double **lhs)
{
    int i, j;

    // Only points from i=0 to i=Nw.
    // 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(Nw+1, 1.0, 0.0); //
    Working_Matrix = pnl_tridiag_mat_create(Nw+1);
    Entree = pnl_vect_create(Nw+1);
```

```
Sortie = pnl_vect_create(Nw+1);

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

    //pnl_tridiag_mat_set (Working_Matrix, 0, -1, Matrix1[0][j][1]);
    pnl_tridiag_mat_set (Working_Matrix, 0,  0, Matrix1[0][j][0]);
    pnl_tridiag_mat_set (Working_Matrix, 0,  1, Matrix1[0][j][2]);
    for (i=0; i<Nw-1; i++)
    {
        pnl_tridiag_mat_set (Working_Matrix, i+1, -1, Matrix1[i+1][j][1]);
        pnl_tridiag_mat_set (Working_Matrix, i+1,  0, Matrix1[i+1][j][0]);
        pnl_tridiag_mat_set (Working_Matrix, i+1,  1, Matrix1[i+1][j][2]);
    }
    pnl_tridiag_mat_set (Working_Matrix, Nw, -1, Matrix1[Nw][j][1]);
    pnl_tridiag_mat_set (Working_Matrix, Nw,  0, Matrix1[Nw][j][0]);
    //pnl_tridiag_mat_set (Working_Matrix, Nw,  1, Matrix1[Nw][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<Nw+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<Nw+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 *wgrid, int Nw, double
                                                   double ***Matrix0, 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<Nw+1; i++)
    {
        for (j=0; j<Nv+1; j++)
        {
            val = Y1[i][j];
            for (st=0; st<11; st++)
            {
                if (it_exists_stencil(i, Nw, j, Nv, st))
                {
                    // If the point exists.
                    point_of_stencil(i, j, st, &istencil, &jstencil);
                    val += -coeff * Matrix2[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 *wgrid, int Nw, double
                                                   double ***Matrix0, double ***
                                                   double **G0, double **G1, dou
                                                   double **rhs, double **lhs)
{
```

```
int i, j;

// Only points from i=1 to i=Nw.
// 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<Nw+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, Matrix2[i][0][3]);
    //pnl_band_mat_set (Working_Matrix, 0, 0-1, Matrix2[i][0][4]);
    pnl_band_mat_set (Working_Matrix, 0, 0+0, Matrix2[i][0][0]);
    pnl_band_mat_set (Working_Matrix, 0, 0+1, Matrix2[i][0][5]);
    pnl_band_mat_set (Working_Matrix, 0, 0+2, Matrix2[i][0][6]);
    //pnl_band_mat_set (Working_Matrix, 1, 1-2, Matrix2[i][1][3]);
    pnl_band_mat_set (Working_Matrix, 1, 1-1, Matrix2[i][1][4]);
    pnl_band_mat_set (Working_Matrix, 1, 1+0, Matrix2[i][1][0]);
    pnl_band_mat_set (Working_Matrix, 1, 1+1, Matrix2[i][1][5]);
    pnl_band_mat_set (Working_Matrix, 1, 1+2, Matrix2[i][1][6]);
    for (j=2; j<Nv-1; j++)
    {
        pnl_band_mat_set (Working_Matrix, j, j-2, Matrix2[i][j][3]);
        pnl_band_mat_set (Working_Matrix, j, j-1, Matrix2[i][j][4]);
        pnl_band_mat_set (Working_Matrix, j, j+0, Matrix2[i][j][0]);
        pnl_band_mat_set (Working_Matrix, j, j+1, Matrix2[i][j][5]);
        pnl_band_mat_set (Working_Matrix, j, j+2, Matrix2[i][j][6]);
    }
    pnl_band_mat_set (Working_Matrix, Nv-1, Nv-1-2, Matrix2[i][Nv-1][3]);
    pnl_band_mat_set (Working_Matrix, Nv-1, Nv-1-1, Matrix2[i][Nv-1][4]);
    pnl_band_mat_set (Working_Matrix, Nv-1, Nv-1+0, Matrix2[i][Nv-1][0]);
    pnl_band_mat_set (Working_Matrix, Nv-1, Nv-1+1, Matrix2[i][Nv-1][5]);
    //pnl_band_mat_set (Working_Matrix, Nv-1, Nv-1+2, Matrix2[i][Nv-1][6]);
    pnl_band_mat_set (Working_Matrix, Nv, Nv-2, Matrix2[i][Nv][3]);
```

```
            pnl_band_mat_set (Working_Matrix, Nv, Nv-1, Matrix2[i][Nv][4]);
            pnl_band_mat_set (Working_Matrix, Nv, Nv+0, Matrix2[i][Nv][0]);
            //pnl_band_mat_set (Working_Matrix, Nv, Nv+1, Matrix2[i][Nv][5]);
            //pnl_band_mat_set (Working_Matrix, Nv, Nv+2, Matrix2[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 manage the dynamic loop
////////////////////////////////////////////////////////////////////////////
static void adi_cycle (double alpha_g,

                       double alpha_m, double Gr, double kappa,

                       // double A0, double *agrid, int Na,
                       double W0, double *wgrid, int Nw,
                       double V0, double sigma_v, double alpha_v, double beta_v,
                       double R0, double rho_wv,
                       double *tgrid, int Nt,
                       double t_begin, double t_end, int index_t_begin, int peri
                       double theta, int scheme,
                       // 2-vectors
                       double **Unm1, double **Utmp, double **Y0, double **Y1, /
                       double **G0nm1, double **G1nm1, double **G2nm1, //double
                       //double **G0n;
                       double **G1n, double **G2n, //double **G3n,
                       double **GSecondMember,
                       // Matrix (=3-vectors)
                       double ***Matrix0nm1, double ***Matrix1nm1, double ***Mat
                       //double ***Matrix0n;
                       double ***Matrix1n, double ***Matrix2n) //double ***Matri
{
    int time_index, i, j;
    double deltat;

    // Time Step
    deltat=(t_end-t_begin)/((double)n_step_by_period); // t_begin < t_end

    // Initialization
    for (i=0; i<Nw+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.;
    }
}

// Finite difference cycle in t-backward order and tau-forward order.
for (time_index=index_t_begin+n_step_by_period;time_index>index_t_begin;time
{
    /*
     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[ which contains
     t_begin, t_end, tgrid[time_index]);
     //*/

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


        //printf("Build matrix at step time_index-1 (in t-past=tau-future)\
        // Compute the elements at time step time_index-1, which is in the t
        // (except A0n and G0n which are not used, so they are erased by the
        build_all_matrix(alpha_g,

                         alpha_m, Gr, kappa,

                         W0, wgrid, Nw,
                         V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                         R0, rho_wv,
                         tgrid, Nt, time_index-1, period,
                         Matrix0nm1, Matrix1n, Matrix2n,
                         G0nm1, G1n, G2n);
        //printf("Build matrix at step time_index (in t-present=tau-present)
        // Compute the elements at time step time_index.
        build_all_matrix(alpha_g,

                         alpha_m, Gr, kappa,

                         W0, wgrid, Nw,
```

```
                    V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                    R0, rho_wv,
                    tgrid, Nt, time_index, period,
                    Matrix0nm1, Matrix1nm1, Matrix2nm1,
                    G0nm1, G1nm1, G2nm1);


//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,
                                   wgrid, Nw, vgrid, Nv,
                                   Matrix0nm1, Matrix1nm1, Matrix2nm
                                   G0nm1, G1nm1, G2nm1,
                                   Unm1, Y0);
//print_vector(Y0, Ns, Nv, Nr);

//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,
                                        wgrid, Nw, vgrid, Nv,
                                        Matrix0nm1, Matrix1nm1, Matr
                                        G0nm1, G1nm1, G2nm1,
                                        Unm1, Y0, Utmp);
//print_vector(Utmp, Ns, Nv, Nr);

//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,
                                        wgrid, Nw, vgrid, Nv,
                                        Matrix0nm1, Matrix1n, Matrix
                                        G0nm1, G1n, G2n,
                                        Utmp, Y1);
//print_vector(Y1, Ns, Nv, Nr);

//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,
                                       wgrid, Nw, vgrid, Nv,
                                       Matrix0nm1, Matrix1nm1, Matri
```

```
                                                G0nm1, G1nm1, G2nm1,
                                                Unm1, Y1, Utmp);
            //print_vector(Utmp, Ns, Nv, Nr);

            //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,
                                                wgrid, Nw, vgrid, Nv,
                                                Matrix0nm1, Matrix1n, Matrix2
                                                G0nm1, G1n, G2n,
                                                Utmp, Unm1); // /!\  with Y2=
            //print_vector(Unm1, Ns, Nv, Nr);

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

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

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

        }
    }

    // End of temporal loop. Copy values in Utmp.
    for (i=0; i<Nw+1; i++)
    {
        for (j=0; j<Nv+1; j++)
        {
            Utmp[i][j] =  Unm1[i][j];
        }
    }

}

////////////////////////////////////////////////////////////////////////////////
```

```
//Compute GLWB Price.
////////////////////////////////////////////////////////////////////////////
static double compute_price(double alpha_g,

                            double alpha_m, double Gr, double* kappa_tabular,

                            double A0, double *agrid, int Na,
                            double W0, double *wgrid, int Nw,
                            double V0, double sigma_v, double alpha_v, double be
                            double *vgrid, int Nv,
                            double R0, double rho_wv,
                            double *tgrid, int number_of_periods, int n_step_by_
                            double *gamma_vect, int number_of_control_values,

                            double theta,  int scheme)
{
    // Variables for loops.
    int i, j, k, st;
    int period;

    // Variables for time loop.
    int index_t_begin;
    double t_begin;
    double t_end;

    // Variables for time events.
    int indx_wgrid, indx_agrid;
    double w_value, a_value;
    double new_U, val_prev, val_next;
    double worst_value;
    double gamma;
    double kappa;
    double withdrawal;
    int indx_gamma;
    int indx_gamma_of_worst_value;

    // Variables to compute price and delta.
    int IndexA, IndexW, IndexV;
    double price, delta;
    double price_wd_vd, price_wd_vu;
    double price_wu_vd, price_wu_vu;
```

```
// 2-vectors
double **G0nm1;
double **G1nm1;
double **G2nm1;
//double **G3nm1;
//double **G0n;
double **G1n;
double **G2n;
//double **G3n;
double **GSecondMember;

// 3-vectors
double ***U_3_new;
double ***U_3_old;

// 2-vectors
double **U_new;
double **U_old;
double **Y0;
double **Y1;
//double **Y2;
//double **Y3;

// Matrix (=3-vectors)
double ***Matrix0nm1;
double ***Matrix1nm1;
double ***Matrix2nm1;
//double ***Matrix3nm1;
//double ***Matrix0n;
double ***Matrix1n;
double ***Matrix2n;
//double ***Matrix3n;

// Memory allocations.
{
    // Memory allocation of 2-vectors for second members.
    G0nm1 = (double**) malloc((Nw+1) * sizeof(double*));
    G1nm1 = (double**) malloc((Nw+1) * sizeof(double*));
    G2nm1 = (double**) malloc((Nw+1) * sizeof(double*));
    //G3nm1 = (double**) malloc((Nw+1) * sizeof(double*));
```

```
//G0n = (double**) malloc((Nw+1) * sizeof(double*));
G1n = (double**) malloc((Nw+1) * sizeof(double*));
G2n = (double**) malloc((Nw+1) * sizeof(double*));
//G3n = (double**) malloc((Nw+1) * sizeof(double*));
for (i=0; i<Nw+1; i++)
{
    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 2-vectors.
U_old = (double**) malloc((Nw+1) * sizeof(double*));
U_new = (double**) malloc((Nw+1) * sizeof(double*));
Y0 = (double**) malloc((Nw+1) * sizeof(double*));
Y1 = (double**) malloc((Nw+1) * sizeof(double*));
//Y2 = (double**) malloc((Nw+1) * sizeof(double*));
//Y3 = (double**) malloc((Nw+1) * sizeof(double*));
for (i=0; i<Nw+1; i++)
{
    U_old[i] = (double*) malloc((Nv+1) * sizeof(double));
    U_new[i] = (double*) malloc((Nv+1) * sizeof(double));
    Y0[i] = (double*) malloc((Nv+1) * sizeof(double));
    Y1[i] = (double*) malloc((Nv+1) * sizeof(double));
    //Y2[i] = (double*) malloc((Nv+1) * sizeof(double));
    //Y3[i] = (double*) malloc((Nv+1) * sizeof(double));
}

// Memory allocation of 3-vectors.
U_3_old = (double***) malloc((Na+1) * sizeof(double**));
U_3_new = (double***) malloc((Na+1) * sizeof(double**));
for (k=0; k<Na+1; k++)
{
    U_3_old[k] = (double**) malloc((Nw+1) * sizeof(double*));
    U_3_new[k] = (double**) malloc((Nw+1) * sizeof(double*));
    for (i=0; i<Nw+1; i++)
```

```
        {
            U_3_old[k][i] = (double*) malloc((Nv+1) * sizeof(double));
            U_3_new[k][i] = (double*) malloc((Nv+1) * sizeof(double));
        }
    }

    // Memory allocation of matrix (=3-vectors).
    Matrix0nm1 = (double***) malloc((Nw+1) * sizeof(double**));
    Matrix1nm1 = (double***) malloc((Nw+1) * sizeof(double**));
    Matrix2nm1 = (double***) malloc((Nw+1) * sizeof(double**));
    //Matrix3nm1 = (double***) malloc((Nw+1) * sizeof(double**));
    //Matrix0n = (double***) malloc((Nw+1) * sizeof(double**));
    Matrix1n = (double***) malloc((Nw+1) * sizeof(double**));
    Matrix2n = (double***) malloc((Nw+1) * sizeof(double**));
    //Matrix3n = (double***) malloc((Nw+1) * sizeof(double**));
    for (i=0; i<Nw+1; i++)
    {
        Matrix0nm1[i] = (double**) malloc((Nv+1) * sizeof(double*));
        Matrix1nm1[i] = (double**) malloc((Nv+1) * sizeof(double*));
        Matrix2nm1[i] = (double**) malloc((Nv+1) * sizeof(double*));
        //Matrix3nm1[i] = (double**) malloc((Nv+1) * sizeof(double*));
        //Matrix0n[i] = (double**) malloc((Nv+1) * sizeof(double*));
        Matrix1n[i] = (double**) malloc((Nv+1) * sizeof(double*));
        Matrix2n[i] = (double**) malloc((Nv+1) * sizeof(double*));
        //Matrix3n[i] = (double**) malloc((Nv+1) * sizeof(double*));
        for (j=0; j<Nv+1; j++)
        {
            Matrix0nm1[i][j] = (double*) malloc(11 * sizeof(double));
            Matrix1nm1[i][j] = (double*) malloc(11 * sizeof(double));
            Matrix2nm1[i][j] = (double*) malloc(11 * sizeof(double));
            //Matrix3nm1[i][j] = (double*) malloc(11 * sizeof(double));
            //Matrix0n[i][j] = (double*) malloc(11 * sizeof(double));
            Matrix1n[i][j] = (double*) malloc(11 * sizeof(double));
            Matrix2n[i][j] = (double*) malloc(11 * sizeof(double));
            //Matrix3n[i][j] = (double*) malloc(11 * sizeof(double));
        }
    }
}// End of memory allocations.

// Initialization.
{
```

```
for (i=0; i<Nw+1; i++)
{
    for (j=0; j<Nv+1; j++)
    {
        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<Nw+1; i++)
{
    for (j=0; j<Nv+1; j++)
    {
        U_old[i][j] = 0.;
        U_new[i][j] = 0.;
        Y0[i][j] = 0.;
        Y1[i][j] = 0.;
        //Y2[i][j] = 0.;
        //Y3[i][j] = 0.;
    }
}
for (k=0; k<Na+1; k++)
{
    for (i=0; i<Nw+1; i++)
    {
        for (j=0; j<Nv+1; j++)
        {
            U_3_old[k][i][j] = 0.;
            U_3_new[k][i][j] = 0.;
        }
    }
}
for (i=0; i<Nw+1; i++)
{
    for (j=0; j<Nv+1; j++)
```

```
        {
            for (st=0; st<11; st++)
            {
                Matrix0nm1[i][j][st] = 0.;
                Matrix1nm1[i][j][st] = 0.;
                Matrix2nm1[i][j][st] = 0.;
                //Matrix3nm1[i][j][st] = 0.;
                //Matrix0n[i][j][st] = 0.;
                Matrix1n[i][j][st] = 0.;
                Matrix2n[i][j][st] = 0.;
                //Matrix3n[i][j][st] = 0.;
            }

        }
    }
}// End of initialization.

GSecondMember = (double**) malloc((Nw+1) * sizeof(double*));
for (i = 0; i < Nw+1; i++)
{
    GSecondMember[i] = (double*) malloc((Nv+1) * sizeof(double));
}
for (i=0; i<Nw+1; i++)
{
    for (j=0; j<Nv+1; j++)
    {
        GSecondMember[i][j]=0.;
    }
}

kappa = kappa_tabular[number_of_periods];

// Terminal values
for (k=0; k<Na+1; k++) {;
    for (i=0; i<Nw+1; i++) {
        for (j=0; j<Nv+1; j++) {
            // Terminal condition for GMWB
            U_3_new[k][i][j] = MAX(wgrid[i],agrid[k]*(1.-kappa));
            U_3_old[k][i][j] = MAX(wgrid[i],agrid[k]*(1.-kappa));
            // After first event time, it should be :
            // U_3_new[k][i][j] = MAX(wgrid[i],agrid[k]-kappa*MAX(agrid[k]-G
```

```
            // U_3_old[k][i][j] = MAX(wgrid[i],agrid[k]-kappa*MAX(agrid[k]-G
        }
    }
}


// Loop over the periods
for (period=number_of_periods; period>=1; period--)
{
    //printf("We are in the period %d.\ n",period);
    index_t_begin = Nt - (number_of_periods-period)*n_step_by_period - n_ste
    t_begin = tgrid[index_t_begin];
    t_end = tgrid[index_t_begin+n_step_by_period];

    //printf("-----------------------------------\ n");
    //print_3vector(U_3_new,Na,Nw,Nv);
    //printf("-----------------------------------\ n");

    // Penalty depends on the period
    kappa = kappa_tabular[period];

    // Time events.
    {

        // Loop over the values of the product depending on account value an
        for (k=0; k<Na+1; k++) {
            //printf("[k=%d]\ n",k);
            for (i=0; i<Nw+1; i++) {
                //printf("[k=%d][i=%d]\ n",k,i);
                for (j=0; j<Nv+1; j++) {


                    // Linear research between withdrawal and surrender.
                    // Worst_value
                    worst_value=-100.;
                    indx_gamma_of_worst_value = -1;
                    // Loop over all the strategies.
                    for (indx_gamma=0;indx_gamma<=number_of_control_values;i
                    {
                        // Define the strategy corresponding to gamma_i.
                        gamma=gamma_vect[indx_gamma];
```

```
withdrawal = gamma*Gr;

//if (j==0) printf("\ t gamma=%f, withdrawal=%f-> ag

if (agrid[k]-withdrawal>=0.)
    // Withdrawal or surrender not exceeding amount
    // This test should be verified at least one tim
    // This is always the case if gamma_vect contain
{

    // Withdrawal not exceeding contract amount.
    if (gamma<=1.)
    {
        a_value=agrid[k]-withdrawal; // 0 <= a_value
        // Find the a_value in the agrid, then start
        indx_agrid=0;
        // Move but never pass agrid[i].
        while ((agrid[indx_agrid]<a_value)&&(indx_ag
        // There is no interpolation for agrid, sinc

        w_value=MAX(wgrid[i]-withdrawal,0.); // 0 <=
        // Find the w_value in the wgrid, then start
        indx_wgrid=0;
        // Move but never pass wgrid[i].
        while ((wgrid[indx_wgrid]<w_value)&&(indx_wg

        //if (j==0) printf("\ t a_value=%f, indx_agr


        if (indx_wgrid==0) // w_value <= wgrid[0] th
        {
            new_U=U_3_new[indx_agrid][0][j];
            //if (j==0) printf("\ t indw_wgrid=0 ---

        }
        else
        {
            //if(indx_wgrid>Nw) // Impossible...
            //{
            //printf("Out of range in withdrawal eve
            //printf("%d %f %f", indx_wgrid, w_value
```

```
            //new_U=U_tmp[Nw][j][k];
            //}
            //else
            //{
            val_next=U_3_new[indx_agrid][indx_wgrid]
            val_prev=U_3_new[indx_agrid][indx_wgrid-
            new_U=val_next+(val_next-val_prev)
            *(w_value-wgrid[indx_wgrid])/(wgrid[indx
            //if (j==0) printf("\ t indx_wgrid>0 ---

            //}
        }

        //if (j==0) printf("\ t -------->new_U=%f, t


        if (worst_value <= new_U + withdrawal)
        {
            worst_value = new_U + withdrawal;
            indx_gamma_of_worst_value = indx_gamma;
            //if (j==0) printf("Withdrawal event cho
        }
    }//End of withdrawal event.

    // Partial or full surrender event.
    else // i.e. (gamma>1.)
    {
        a_value=agrid[k]-withdrawal; // 0 <= a_value
        // Find the a_value in the agrid, then start
        indx_agrid=0;
        // Move but never pass agrid[i].
        while ((agrid[indx_agrid]<a_value)&&(indx_ag
        // There is no interpolation for agrid, sinc

        w_value=MAX(wgrid[i]-withdrawal,0.); // 0 <=
        // Find the w_value in the wgrid, then start
        indx_wgrid=0;
        // Move but never pass wgrid[i].
        while ((wgrid[indx_wgrid]<w_value)&&(indx_wg

        //if (j==0) printf("\ t a_value=%f, indx_agr
```

```
                if (indx_wgrid==0)
                {
                    new_U=U_3_new[indx_agrid][0][j];
                    //if (j==0) printf("\ t indw_wgrid=0 ---
                }
                else
                {
                    val_next=U_3_new[indx_agrid][indx_wgrid]
                    val_prev=U_3_new[indx_agrid][indx_wgrid-
                    new_U=val_next+(val_next-val_prev)
                    *(w_value-wgrid[indx_wgrid])/(wgrid[indx
                    //if (j==0) printf("\ t indx_wgrid>0 ---
                }

                //if (j==0) printf("\ t -------->new_U=%f, t

                if (worst_value <= new_U + (1.-kappa) * with
                {
                    worst_value = new_U + (1.-kappa) * withd
                    indx_gamma_of_worst_value = indx_gamma;
                    //if (j==0) printf("Surrender event chos
                }
            }//End of partial or full surrender event.
        } // End of test if Withdrawal or surrender not exce
        //else
        //if (j==0) printf("\ t Withdrawal exceeds benefit b
    }//End of loop over gamma_i.

    //if (j==0) printf("worst_value=%f\ n",worst_value );
    //if (j==0) printf("expected value=%f\ n",MAX(wgrid[i],a
    //U_3_old[k][i][j] = MAX(worst_value,0.); // To avoid ne
    if (period==number_of_periods)
    {
        U_3_old[k][i][j] = MAX(wgrid[i],agrid[k]-kappa*MAX(a
    }
    else
    {
        U_3_old[k][i][j] = worst_value;
```

```
                }


            }//End of loop over j.
        }// End of loop over i.
    }//End of loop over k.
}

//printf("-----------------------------------\ n");
//print_3vector(U_3_old,Na,Nw,Nv);
//printf("-----------------------------------\ n");


// Compute the evolution between period and period-1.
//printf("Compute the evolution between period %d and period %d.\ n",per
///*
for (k=0; k<Na+1; k++) {

    // Copy U_3_old[k][.][.] in U_old.
    for (i=0; i<Nw+1; i++) {
        for (j=0; j<Nv+1; j++) {
            U_old[i][j]=U_3_old[k][i][j];
        }
    }

    // Make adi cycle with fixed k.
    adi_cycle(alpha_g,

              alpha_m, Gr, kappa,

              W0, wgrid, Nw,
              V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
              R0, rho_wv,
              tgrid, Nt, t_begin, t_end, index_t_begin, period, n_step_b
              theta, scheme,
              // 2-vectors
              U_old, U_new, Y0, Y1, //Y2, //Y3,
              G0nm1, G1nm1, G2nm1, //G3nm1, //G0n,
              G1n, G2n, //G3n,
              GSecondMember,
              // Matrix (=3-vectors)
```

```
                    Matrix0nm1, Matrix1nm1, Matrix2nm1, //MatrixA3nm1, //Matri
                    Matrix1n, Matrix2n); //MatrixA3n);

        // Copy U_new in U_3_new[k][.][.].
        for (i=0; i<Nw+1; i++) {
            for (j=0; j<Nv+1; j++) {
                U_3_new[k][i][j]=U_new[i][j];
            }
        }
    }
    //*/

    //printf("-----------------------------------\ n");
    //print_3vector(U_3_new,Na,Nw,Nv);
    //printf("-----------------------------------\ n");



}//End of loop over periods.

// U_new <- initial data ;
// Loop starts here
// Ratchet(U_new) -> U_new=U_old ;
// Event_Time(U_new) -> U_old ;
// Death_Benefit(U_old) -> U_old
// During the adi cycle : ADI(U_old) -> U_new ;
// Do loop


// Index in the domain for the price.
// Find the index in wgrid corresponding to the value A0 of the account.
// Must be exact. No interpolation here.
IndexA = lower_index(agrid,Na+1,A0);
// Find the index in wgrid corresponding to the price W0 of the asset.
IndexW = lower_index(wgrid,Nw+1,W0);
// Find the index in vgrid corresponding to the variance V0 of the asset.
IndexV = lower_index(vgrid,Nv+1,V0);

//printf("IndexA= %d, IndexW= %d, IndexV=%d\ n",IndexA,IndexW,IndexV);
//printf("A ~ %f, W ~ %f V ~ %f\ n",agrid[IndexA],wgrid[IndexW],vgrid[IndexV
//printf("Value = %f\ n",U_3_new[IndexA][IndexW][IndexV]);
```

```
// First compute the delta (using price as temporary variable). We do an int
price_wd_vd = U_3_new[IndexA][IndexW+1][IndexV];
price_wd_vu = U_3_new[IndexA][IndexW+1][IndexV+1];
price_wu_vd = U_3_new[IndexA][IndexW+2][IndexV];
price_wu_vu = U_3_new[IndexA][IndexW+2][IndexV+1];

price=double_interpolation(price_wd_vd, price_wd_vu, price_wu_vd, price_wu_v
                           wgrid[IndexW+1], wgrid[IndexW+2],
                           vgrid[IndexV], vgrid[IndexV+1],
                           wgrid[IndexW+1], V0);

price_wd_vd = U_3_new[IndexA][IndexW][IndexV];
price_wd_vu = U_3_new[IndexA][IndexW][IndexV+1];
price_wu_vd = U_3_new[IndexA][IndexW+1][IndexV];
price_wu_vu = U_3_new[IndexA][IndexW+1][IndexV+1];
//printf("price_wd_vd=%f, price_wd_vu=%f, price_wu_vd=%f, price_wu_vu=%f\ n"

delta =(price - double_interpolation(price_wd_vd, price_wd_vu, price_wu_vd,
                                     wgrid[IndexW], wgrid[IndexW+1],
                                     vgrid[IndexV], vgrid[IndexV+1],
                                     wgrid[IndexW], V0)     )
/(wgrid[IndexW+1] - wgrid[IndexW]);

// Next compute the price. We do an interpolation.
price=double_interpolation(price_wd_vd, price_wd_vu, price_wu_vd, price_wu_v
                           wgrid[IndexW], wgrid[IndexW+1],
                           vgrid[IndexV], vgrid[IndexV+1],
                           W0, V0);
//printf("price=%f\ n", price);


// Memory desallocations.
{
    // Memory desallocation of matrix (=3-vectors).
    for (i=0; i<Nw+1; i++)
    {
        for (j=0; j<Nv+1; j++)
        {
            free(Matrix0nm1[i][j]);
            free(Matrix1nm1[i][j]);
```

```
            free(Matrix2nm1[i][j]);
            //free(MatrixA3nm1[i][j]);
            //free(Matrix0n[i][j]);
            free(Matrix1n[i][j]);
            free(Matrix2n[i][j]);
            //free(MatrixA3n[i][j]);
        }
        free(Matrix0nm1[i]);
        free(Matrix1nm1[i]);
        free(Matrix2nm1[i]);
        //free(MatrixA3nm1[i]);
        //free(Matrix0n[i]);
        free(Matrix1n[i]);
        free(Matrix2n[i]);
        //free(MatrixA3n[i]);
    }
    free(Matrix0nm1);
    free(Matrix1nm1);
    free(Matrix2nm1);
    //free(MatrixA3nm1);
    //free(Matrix0n);
    free(Matrix1n);
    free(Matrix2n);
    //free(MatrixA3n);

    // Memory desallocation of 3-vectors.
    for (k=0; k<Na+1; k++)
    {
        for (i=0; i<Nw+1; i++)
        {
            free(U_3_old[k][i]);
            free(U_3_new[k][i]);
        }
        free(U_3_old[k]);
        free(U_3_new[k]);
    }
    free(U_3_old);
    free(U_3_new);

    // Memory desallocation of 2-vectors.
    for (i=0; i<Nw+1; i++)
```

```
        {
            free(U_old[i]);
            free(U_new[i]);
            free(Y0[i]);
            free(Y1[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(U_old);
        free(U_new);
        free(Y0);
        free(Y1);
        //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 the price.
    return price;
}
```

```
static double compute_fair_price(double alpha_old, double alpha_new,

                                 double alpha_m, double Gr, double* kappa_tabula

                                 double A0, double *agrid, int Na,
                                 double W0, double *wgrid, int Nw,
                                 double V0, double sigma_v, double alpha_v, doub
                                 double *vgrid, int Nv,
                                 double R0, double rho_wv,
                                 double *tgrid, int number_of_periods, int n_ste
                                 double *gamma_vect, int number_of_control_value

                                 double theta,  int scheme)
{
    int cpt=0;
    double pr_old, pr_new;
    double alpha;
    double err;
    double price;

    int i;
    double* local_tgrid;
    int local_n_step_by_period, local_Nt;


    local_n_step_by_period = n_step_by_period>=20 ? n_step_by_period/20 : 1;
    local_Nt = number_of_periods*local_n_step_by_period;
    local_tgrid=(double *)malloc((local_Nt+1)*sizeof(double));

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

    pr_old = compute_price(alpha_old,

                           alpha_m, Gr, kappa_tabular,

                           A0, agrid, Na,
                           W0, wgrid, Nw,
                           V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
```

```
                        R0, rho_wv,
                        local_tgrid, number_of_periods, local_n_step_by_perio
                        gamma_vect, number_of_control_values,
                        theta, scheme);
cpt++;
pr_new = compute_price(alpha_new,

                        alpha_m, Gr, kappa_tabular,

                        A0, agrid, Na,
                        W0, wgrid, Nw,
                        V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                        R0, rho_wv,
                        local_tgrid, number_of_periods, local_n_step_by_perio
                        gamma_vect, number_of_control_values,
                        theta, scheme);
cpt++;
err = W0 - pr_new;

while ((ABS(err) > 0.00000001) && (cpt<=4))
{
    // Secante method.
    // New point
    alpha = alpha_new - (pr_new-W0) * (alpha_new-alpha_old)/(pr_new-pr_old);
    //printf("alpha = %f\ n",alpha);
    price = compute_price(alpha,

                            alpha_m, Gr, kappa_tabular,

                            A0, agrid, Na,
                            W0, wgrid, Nw,
                            V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                            R0, rho_wv,
                            local_tgrid, number_of_periods, local_n_step_by_pe
                            gamma_vect, number_of_control_values,
                            theta, scheme);
    err = W0 - price;
    // Update variables.
    alpha_old = alpha_new;
    alpha_new = alpha;
    pr_old = pr_new;
```

```
    pr_new = price;
    cpt++;
}

pr_old = compute_price(alpha_old,

                       alpha_m, Gr, kappa_tabular,

                       A0, agrid, Na,
                       W0, wgrid, Nw,
                       V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                       R0, rho_wv,
                       tgrid, number_of_periods, n_step_by_period, Nt,
                       gamma_vect, number_of_control_values,
                       theta, scheme);
cpt++;
pr_new = compute_price(alpha_new,

                       alpha_m, Gr, kappa_tabular,

                       A0, agrid, Na,
                       W0, wgrid, Nw,
                       V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                       R0, rho_wv,
                       tgrid, number_of_periods, n_step_by_period, Nt,
                       gamma_vect, number_of_control_values,
                       theta, scheme);
cpt++;
err = W0 - pr_new;

while (ABS(err) > 0.00000001)
{


    // Secante method.
    // New point
    alpha = alpha_new - (pr_new-W0) * (alpha_new-alpha_old)/(pr_new-pr_old);
    //printf("alpha = %f\ n",alpha);
    price = compute_price(alpha,

                          alpha_m, Gr, kappa_tabular,
```

```
                                A0, agrid, Na,
                                W0, wgrid, Nw,
                                V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                                R0, rho_wv,
                                tgrid, number_of_periods, n_step_by_period, Nt,
                                gamma_vect, number_of_control_values,
                                theta, scheme);
        err = W0 - price;
        // Update variables.
        alpha_old = alpha_new;
        alpha_new = alpha;
        pr_old = pr_new;
        pr_new = price;
        cpt++;
    }
    free(local_tgrid);
    return alpha;

}

static int FD_gmwb_hes(double W0, double maturity, double R0, double V0, double
{
   // For loop index.
    int i,j;

    // New variables given by PREMIA
    double* Kt;

    // Product's features
    int number_of_withdrawals_per_periods;
    int number_of_periods;
    double* kappa_tabular; // Penalty charges
    int number_of_control_values;
    double *gamma_vect;
    double A0;
double Gr;
    // Numerical parameters
    int Na, Nt;
    double *agrid, *wgrid, *vgrid, *tgrid;
    double Amax, Aleft, Aright, Coeff_a;
```

```
double Wmax, Wleft, Wright, Coeff_w;
double Vmax, Coeff_v;
int n_step_by_period;
int scheme;
double theta;
// Variables for method of pricing
double alpha_old, alpha_new;

A0=W0;

// Fees, management fees, max withdrawal rate and penalty.
number_of_withdrawals_per_periods = (int)(Nmonit/maturity);
number_of_periods = Nmonit; // For instance a period is one year or one seme
// Gr is a percentage of A0
Gr = Gr_fixed/number_of_withdrawals_per_periods; // This is equivalent to Gr

number_of_control_values = (int)(A0/Gr);
gamma_vect=(double *)malloc((number_of_control_values+1)*sizeof(double));
for (i=0; i<=number_of_control_values; i++)
    gamma_vect[i] =  i;

Nt = maturity * n_step_for_year;
n_step_by_period = (int) (Nt / Nmonit);

//Surrender charges Kt, maximum of 5 changes.
Kt = (double*)malloc((Nt + 1) * sizeof(double));
for (i = 0; i <= Nt; i++)
{
    j = 0;
    while ((pnl_vect_get(timesKt_fixed,j) < (double)i / n_step_for_year) &&
        j++;

    Kt[i] = pnl_vect_get(Kt_fixed,j);
}

kappa_tabular = (double*)malloc((number_of_periods + 1) * sizeof(double));
for (i = 0; i <= number_of_periods; i++)
{
    kappa_tabular[i] = Kt[(int)((i*Nt)/number_of_periods)];
}
```

```
// Space numerical parameters.
Na = number_of_periods; // Do not use Nj, since it is useless to refine agri

// Account
Amax = A0;
Aleft = 0.3*A0;
Aright = 0.7*A0;
Coeff_a = A0/20.; // Environ entre 2 et 5 ou fraction de Na/2
Wmax = 5.*maturity*W0;//5.*multi*W0;
Wleft = 0.8*W0;
Wright = 1.2*W0;
Coeff_w = W0/20.; // Environ entre 2 et 5 ou fraction de Nw/2
// Variance
Vmax = 5.;//-MAX(-MAX(5.*multi*V0,1.),-5.);
Coeff_v = Vmax/500.;
// Scheme numerical parameters.
theta=0.5;//Theta schema
scheme=0.;//Douglas Scheme

alpha_old = 0./10000.;
alpha_new = 200./10000.;


//////////////////////////////////////////
// Compute agrid, wgrid, vgrid and tgrid. //
//////////////////////////////////////////
// Memory allocation of 1-vectors.
agrid=(double *)malloc((Na+1)*sizeof(double));
wgrid=(double *)malloc((Nw+1)*sizeof(double));
vgrid=(double *)malloc((Nv+1)*sizeof(double));
tgrid=(double *)malloc((Nt+1)*sizeof(double));
grid_generation_guarantee(Aleft, Aright, Amax, Coeff_a, agrid, Na);
grid_generation_sub_account(Wleft, Wright, Wmax, Coeff_w, wgrid, Nw);
grid_generation_variance(Vmax, V0, Coeff_v, vgrid, Nv);

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

```
    ////////////////////////
    // Compute the fair fee. //
    ////////////////////////

    *ptprice=compute_fair_price(alpha_old, alpha_new,
                        alpha_m, Gr, kappa_tabular,
                        A0, agrid, Na,
                        W0, wgrid, Nw,
                        V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                        R0, rho_wv,
                        tgrid, number_of_periods, n_step_by_period, Nt,
                        gamma_vect, number_of_control_values,
                        theta, scheme);


    free(agrid);
    free(wgrid);
    free(vgrid);
    free(tgrid);
    free(gamma_vect);
    free(Kt);
    free(kappa_tabular);

  return OK;
}

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

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

  return FD_gmwb_hes(ptMod->S0.Val.V_PDOUBLE, ptOpt->Maturity.Val.V_DATE - ptMod
                                ptMod->MeanReversion.Val.V_PDOUBLE,
                                ptMod->LongRunVariance.Val.V_PDOUBLE,
                                ptMod->Sigma.Val.V_PDOUBLE,
   ptMod->Rho.Val.V_PDOUBLE, ptOpt->Alpha_m.Val.V_PDOUBLE, ptOpt->MaximumWithdra
}
```

```
static int CHK_OPT(FD_GMWB_HES)(void *Opt, void *Mod)
{
  if ((strcmp(((Option *)Opt)->Name, "GMWB") == 0))
    return OK;
  else
    return WRONG;
}

#endif //PremiaCurrentVersion

static int MET(Init)(PricingMethod *Met, Option *Opt)
{
  if (Met->init == 0)
    {
      Met->init = 1;
      Met->HelpFilenameHint = "fd_gmwb_hes";
      Met->Par[0].Val.V_INT2 = 10;
      Met->Par[1].Val.V_INT2 = 150;
      Met->Par[2].Val.V_INT2 = 10;
    }

  return OK;
}

PricingMethod MET(FD_GMWB_HES) =
{
  "FD_GMWB_HES",
  { {"Timestep_for_year", INT2, {100}, ALLOW}, {"SpaceStepNumber W", INT2, {100}
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(FD_GMWB_HES),
  { {"Fair fee", DOUBLE, {100}, FORBID},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(FD_GMWB_HES),
  CHK_split,
  MET(Init)
};
```