Help

```
#include  "
href../../../mod/bshw1d/bshw1d_stda/bshw1d_stda_h_src.pdfbshw1d_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_BSHW)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(FD_GMWB_BSHW)(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 Bla
// model with sotchastic rate undeer Hull-White model.
// 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.
//////////
// Compute Zero-Coupon bond.
//////////
// Compute the mean reversion from Zero Coupon Bond data curve.
static double *initial_forward; //Values of initial_forward
static double *initial_derive_forward; //Values of initial_derive_forward

static int lecture_tr(char *init_tr, double *Pm, double *tm)
{
    int i;
    char ligne[30];
    char *pligne;
    double p, tt_value;
    FILE *Entrees;

    Entrees = fopen(init_tr, "r");
```

```
    if (Entrees == NULL)
    {
printf("LE FICHIER N'A PU ETRE OUVERT. VERIFIEZ LE CHEMIN.\ n");
    }


    /// i is the number of lines that has been read.
    i = 0;
    pligne = ligne;
    //   Pm = (double *)malloc(200 * sizeof(double));
    //   tm = (double *)malloc(200 * sizeof(double));

    while (1)
    {
pligne = fgets(ligne, sizeof(ligne), Entrees);
if (pligne == NULL) break;
else
{
    sscanf(ligne, "%lf t=%lf", &p, &tt_value);
    // The line read must be written "0.943290 t=0.5" where 0.943290 is a double
    Pm[i] = p; // save the price of the zero coupon.
    tm[i] = tt_value; // save the corresponding time.
    i++;
}
    }

    free(pligne);
    fclose(Entrees);
    return i;
}

static void interpole(int n_price, int imax, double *Pm, double* tm, double *ini
{
    int i, iF, j;
    double p_infinity;

    n_price--;
    i = 0;
    while (t[i] <= tm[1])
    {
        initial_yield[i] = (tm[1] - t[i]) / tm[1] + t[i] * Pm[1] / tm[1];
```

```
        i++;
    }
    for (j = 0; j < n_price; j++)
    {
        while ((i <= imax) && (t[MIN(i,imax)] < tm[j + 1]))
        {
            initial_yield[i] = (tm[j + 1] - t[i]) / (tm[j + 1] - tm[j]) * Pm[j]
            i++;
        }
    }

    if ((i <= imax) && (t[MIN(i,imax)] >= tm[n_price]))
    {
        // Extrapolation to a limit value at infinite
        for (iF = i ; iF <= imax ; iF++)
        {
            p_infinity = Pm[n_price];
            initial_yield[iF] = Pm[n_price] + (p_infinity - Pm[n_price]) / (t[im
        }
        // or extrapolation following last tendancy -> could lead to negative va
        for (iF = i ; iF <= imax ; iF++)
        {
            //            initial_yield[iF] = Pm[n_price] + (Pm[n_price] - Pm[n_
        }
    }
}


static void compute_forward(double *Price, double *Time, int size, double *Forwa
{
    //Forward = - d log(P) / dt
    //Derive_Forward = d Forward / dt
    int i;

    for (i=0; i<=size-1; i++)
    {
Forward[i] = -(log(Price[i+1])-log(Price[i]))/(Time[i+1]-Time[i]);
    }
    Forward[size] = Forward[size-1];

    for (i=0; i<=size-1; i++)
```

```
    {
Derive_Forward[i] = (Forward[i+1]-Forward[i])/(Time[i+1]-Time[i]);
    }
    Derive_Forward[size] = Derive_Forward[size-1];
}

static void extract_forward(double *Forward, double *Derive_Forward, double *Tim
{
    int i;
    int j;

    i=0;
    j=0;
    while (j<size_extract)
    {
while ((Time[i]<t_extract[j]) && (i<size))
    i++;

initial_forward[j] = Forward[i];
initial_derive_forward[j] = Derive_Forward[i];

j++;
    }
}

static double func_zero_coupon(double* tgrid, int Nt, int time_index_arg,
       double alpha_r, double sigma_r, int flat_flag, double R0_flat)
{
    // This function characterizes the mean reversion at time tgrid[time_index_a
    // dr = alpha_r ( THIS_FUNCTION(t) - r) dt + sigma_r dWt

    // Be careful, actually this function assumes that "forward" and "derive_for
    // computed/interpolated on tgrid...

    double b;
    double time;
    int time_index;

    //tgrid[time_index_arg] is the real time t.
    time = tgrid[time_index_arg];
    time_index = time_index_arg;
```

```
    // So do you use T-t ???
    // time = tgrid[Nt]-tgrid[time_index_arg]; // T-t
    // time_index = Nt-time_index_arg;

    //////////
    // Vasicek : THIS = Cst
    //return b;
    //////////

    //////////
    // Hull-White : THIS = function of time
    // An example :
    // return b+(sigma_r*sigma_r)/(2.*(alpha_r*alpha_r))*(1.-exp(-2*alpha_r*time

    // With interest curve P(r,t,T) = exp(-b(T-t))
    // return ( b*alpha_r+(sigma_r*sigma_r)/(2.*alpha_r)*(1.-exp(-2*alpha_r*time

    // With the forward rate f = -d_t log(P)
    // return ( df/dt f(0,t) + alpha_r f(0,t) + (sigma_r*sigma_r)/(2.*alpha_r)*(


    if (flat_flag==0)
    {
b = R0_flat;
//printf("return = %f, time = %f\ n",
//((sigma_r*sigma_r)/(2.*alpha_r)*(1.-exp(-2*alpha_r*time)))/alpha_r,time);
return b + ((sigma_r*sigma_r)/(2.*alpha_r)*(1.-exp(-2*alpha_r*time)))/alpha_r;
    }
    else
    {
//   printf("initial_forward= %f",initial_forward[time_index]);
//   printf("initial_derive_forward= %f",initial_derive_forward[time_index]);
b = initial_forward[time_index] + initial_derive_forward[time_index]/alpha_r;
//   printf("b= %f",b);
//   printf("return = %f",((sigma_r*sigma_r)/(2.*alpha_r)*(1.-exp(-2*alpha_r*tim
return b + ((sigma_r*sigma_r)/(2.*alpha_r)*(1.-exp(-2*alpha_r*time)))/alpha_r;
// Take care of "/alpha_r" since mean reversion is "alpha_r * ( THIS - r)" and i
    }
}
```

```
//////////
// 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_rate(double Rmax, double Center_r, double Coeff_r, d
{
    int k;
    double deltazeta;

    deltazeta = (asinh1((Rmax-Center_r)/Coeff_r)-asinh1((-Rmax-Center_r)/Coeff_r

    // Definition of uniform grid eta.
    for (k=0; k<=Nr; k++)
    {
rgrid[k] = asinh1((-Rmax-Center_r)/Coeff_r) + k * deltazeta;
    }
    // Definition of the rate grid with the uniform grid zeta.
    for (k=0; k<=Nr; k++)
    {
rgrid[k] = Center_r + Coeff_r*sinh(rgrid[k]);
    }

k = lower_index(rgrid, Nr, Center_r);
rgrid[k] = Center_r;
}

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

```
// 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_d_d, double value_d_u,
   double value_u_d, double value_u_u,
   double grid_one_d, double grid_one_u,
   double grid_two_d, double grid_two_u,
   double unknown_one, double unknown_two)
{
    double value_one_d,value_one_u;
    value_one_d = interpolation (grid_two_d, value_d_d, grid_two_u, value_d_u, u
    value_one_u = interpolation (grid_two_d, value_u_d, grid_two_u, value_u_u, u
    return interpolation (grid_one_d, value_one_d, grid_one_u, value_one_u, unkn
}

static int it_exists_stencil(int i, int Nw, int j, int Nr, int stencil)
{
    // We use i from 0 to Nw, j from 0 to Nr.
    // 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<Nr-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==Nr-1)
if (stencil== 6)
    return 0;

    if (j==Nr)
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 R0, double sigma_r, double alpha_r, int flat_flag, double R0_flat,
       double *rgrid, int Nr, int ir,
       double V0, double rho_wr,
       double *tgrid, int Nt, int time_index, int period)
{
    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 R0, double sigma_r, double alpha_r, int flat_flag, double R0_flat,
       double *rgrid, int Nr, int ir,
       double V0, double rho_wr,
       double *tgrid, int Nt, int time_index, int period)
{
    return 0.;
}

static double bc_rate_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 R0, double sigma_r, double alpha_r, int flat_flag, double R0_flat,
```

```
  double *rgrid, int Nr, int ir,
  double V0, double rho_wr,
  double *tgrid, int Nt, int time_index, int period)
{
    double time;
    double b;
    double a;
    double deriv;
    int decalage_derivative;


    time = tgrid[time_index];
    b=func_zero_coupon(tgrid, Nt, time_index, alpha_r, sigma_r, flat_flag, R0_fl
    a=alpha_r;
    deriv = (exp(-a*time)-1.)/a; // Derivative of func(r) in the exponential. Ta
    decalage_derivative = 1; // 0 or 1. Use Neumann condition at point Rmin or R

    if (1==0) // Bond
    {
if (flat_flag==0)
{
    // Assuming Vasicek model, the closed formula gives the value of the derivat
    // Take care of the sign minus in the return value since U0 = U1 - Neumann *
    return -1. * deriv*exp(-b*time + (b-rgrid[ir+decalage_derivative])/a*(1.-exp
   + sigma_r*sigma_r/(2.*a*a)*time+sigma_r*sigma_r/(4.*a*a*a)*(1.-exp(-2.*a*time
   - sigma_r*sigma_r/(a*a*a)*(1.-exp(-a*time)))
    * (rgrid[ir+1]-rgrid[ir]);
}
else
{
    // Assuming Hull-White model,
    // Bond = Price(0,t)/Price(0,0) * exp(-deriv * forward(0,t) - sigma*sigma (1
    // * exp (deriv * R)
    // Take care of the sign minus in the return value since U0 = U1 - Neumann *
    // So "d Bond / dr" = -deriv * Price(0,t)/Price(0,0) * exp(-deriv * forward(


    // Furtermore, in all cases, homogeneous Neumann should be enough...
    return 0.;
}
```

```
    }
    // Neumann = 0
    return 0.;
}

static double bc_rate_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 R0, double sigma_r, double alpha_r, int flat_flag, double R0_flat,
  double *rgrid, int Nr, int ir,
  double V0, double rho_wr,
  double *tgrid, int Nt, int time_index, int period)
{
    double time;
    double b;
    double a;
    double deriv;
    int decalage_derivative;

    time = tgrid[time_index];
    b=func_zero_coupon(tgrid, Nt, time_index, alpha_r, sigma_r, flat_flag, R0_fl
    a=alpha_r;
    deriv = (exp(-a*time)-1.)/a; // Derivative of func(r) in the exponential. Ta
    decalage_derivative = 0; // 0 or 1. Use Neumann condition at point Rmax or R


    if (1==0) // Bond
    {
if (flat_flag==0)
{
    // Assuming Vasicek model, the closed formula gives the value of the derivat
    // Take care of the sign plus in the return value since UN = UN-1 + Neumann
    return deriv*exp(-b*time + (b-rgrid[ir-decalage_derivative])/a*(1.-exp(-a*ti
     + sigma_r*sigma_r/(2.*a*a)*time+sigma_r*sigma_r/(4.*a*a*a)*(1.-exp(-2.*a*ti
     - sigma_r*sigma_r/(a*a*a)*(1.-exp(-a*time)))
    * (rgrid[ir]-rgrid[ir-1]);
}
else
{
    // Assuming Hull-White model,
```

```
    // Bond = Price(0,t)/Price(0,0) * exp(-deriv * forward(0,t) - sigma*sigma (1
    // * exp (deriv * R)
    // So "d Bond / dr" = deriv * Price(0,t)/Price(0,0) * exp(-deriv * forward(0

    // Furthermore, in all cases, homogeneous Neumann should be enough...
    return 0.;
}
    }

    // Neumann = 0
    return 0.;
}


static double supplementary_terms(double alpha_m, double Gr, double kappa,
  double *tgrid, int Nt, int time_index, int period)
{
    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 R0, double sigma_r, double alpha_r, int flat_flag, double R0_flat,
     double *rgrid, int Nr,
     double V0, double rho_wr,
     double *tgrid, int Nt, int time_index, int period,
     double ***Matrix0, double ***Matrix1, double ***Matrix2,
     double **G0, double **G1, double **G2)
{
    double actualwpoint;
    double actualvpoint; // To unify notations.
    double actualrpoint;

    double Dwi, Dwip1;
    double Drj, Drjp1;
```

```
    double convection_w, diffusion_w;
    double convection_r, diffusion_r;
    double order_0, mixted_wr;

    double *cw;
    double *dw;
    double *cr;
    double *dr;
    double *mwr;

    // 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 ralpha_jm2, ralpha_jm1, ralpha_j0;
    double rbeta_jm1, rbeta_j0, rbeta_jp1;
    //   double rgamma_j0, rgamma_jp1, rgamma_jp2;
    //   double rdelta_jm1, rdelta_j0, rdelta_jp1;

    int i, j, st;

    cw=(double*)malloc(11*sizeof(double));
    dw=(double*)malloc(11*sizeof(double));
    cr=(double*)malloc(11*sizeof(double));
    dr=(double*)malloc(11*sizeof(double));
    mwr=(double*)malloc(11*sizeof(double));

    double G_cw, G_dw, G_cr, G_dr, G_mwr;

    for(j=0;j<Nr+1;j++)
    {
for(i=0;i<Nw+1;i++)
{
    for(st=0;st<11;st++)
    {
cw[st]=0.;
dw[st]=0.;
cr[st]=0.;
dr[st]=0.;
```

```
mwr[st]=0.;
    }
    G_cw=0.;
    G_dw=0.;
    G_cr=0.;
    G_dr=0.;
    G_mwr=0.;

    actualwpoint = wgrid[i];
    actualrpoint = rgrid[j];
    actualvpoint = V0;

    convection_w = (actualrpoint - alpha_g) * actualwpoint;
    diffusion_w = actualwpoint * actualwpoint * actualvpoint/2.0;
    convection_r = alpha_r*(func_zero_coupon(tgrid, Nt, time_index, alpha_r, sig
    diffusion_r = sigma_r * sigma_r/2.0;
    order_0 = -actualrpoint;
    mixted_wr = rho_wr * sigma_r * actualwpoint * sqrt(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 grid.
    // 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 rate grid.
    // Compute Drj, copy it in Drjp1.
    // Call bc_rate_max at point j.
    // Suppress dr[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) // W=Wmin -> 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,
     R0, sigma_r, alpha_r, flat_flag, R0_flat,
     rgrid, Nr, j,
     V0, rho_wr,
     tgrid, Nt, time_index, period);

    //cw[stencil(-1,0)]=-Dwip1/(Dwi*(Dwi+Dwip1));
    cw[stencil(0,0)]=(Dwip1-Dwi)/(Dwi*Dwip1) - Dwip1/(Dwi*(Dwi+Dwip1));
    cw[stencil(1,0)]=Dwi/(Dwip1*(Dwi+Dwip1));
    G_cw += -Dwip1/(Dwi*(Dwi+Dwip1)) * bc_w_min(alpha_g,
alpha_m, Gr, kappa,
//A0, agrid, Na, k,
W0, wgrid, Nw, i,
R0, sigma_r, alpha_r, flat_flag, R0_flat,
rgrid, Nr, j,
V0, rho_wr,
tgrid, Nt, time_index, period);
}
else
{
    if (i==Nw) // W=Wmax -> 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+Dwip1));
//dw[stencil(1,0)]=2.0/(Dwip1*(Dwi+Dwip1));
G_dw = 2.0/(Dwip1*(Dwi+Dwip1)) * bc_w_max(alpha_g,
```

```
  alpha_m, Gr, kappa,
  //A0, agrid, Na, k,
  W0, wgrid, Nw, i,
  R0, sigma_r, alpha_r, flat_flag, R0_flat,
  rgrid, Nr, j,
  V0, rho_wr,
  tgrid, Nt, time_index, period);

cw[stencil(-1,0)]=-Dwip1/(Dwi*(Dwi+Dwip1));
cw[stencil(0,0)]=(Dwip1-Dwi)/(Dwi*Dwip1) + Dwi/(Dwip1*(Dwi+Dwip1));
//cw[stencil(1,0)]=Dwi/(Dwip1*(Dwi+Dwip1));
G_cw += Dwi/(Dwip1*(Dwi+Dwip1)) * bc_w_max(alpha_g,
   alpha_m, Gr, kappa,
   //A0, agrid, Na, k,
   W0, wgrid, Nw, i,
   R0, sigma_r, alpha_r, flat_flag, R0_flat,
   rgrid, Nr, j,
   V0, rho_wr,
   tgrid, Nt, time_index, period);
    }
    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 R
    ////////////////////
    //printf("Diffusion and Convection R.\ n");
    {
if (j==0) // R=Rmin
{
```

```
    Drjp1 = rgrid[j+1]-rgrid[j];
    Drj = Drjp1;
    //dr[stencil(0,-1)]=2.0/(Drj*(Drj+Drjp1));
    dr[stencil(0,0)]=-2.0/(Drj*Drjp1) + 2.0/(Drj*(Drj+Drjp1));
    dr[stencil(0,1)]=2.0/(Drjp1*(Dwi+Drjp1));
    G_dr += 2.0/(Drj*(Drj+Drjp1)) * bc_rate_min(alpha_g,
alpha_m, Gr, kappa,
//A0, agrid, Na, k,
W0, wgrid, Nw, i,
R0, sigma_r, alpha_r, flat_flag, R0_flat,
rgrid, Nr, j,
V0, rho_wr,
tgrid, Nt, time_index, period);

    //cr[stencil(0,-1)]=-Drjp1/(Drj*(Drj+Drjp1));
    cr[stencil(0,0)]=(Drjp1-Drj)/(Drj*Drjp1) - Drjp1/(Drj*(Drj+Drjp1));
    cr[stencil(0,1)]=Drj/(Drjp1*(Drj+Drjp1));
    G_cr += -Drjp1/(Drj*(Drj+Drjp1)) * bc_rate_min(alpha_g,
   alpha_m, Gr, kappa,
   //A0, agrid, Na, k,
   W0, wgrid, Nw, i,
   R0, sigma_r, alpha_r, flat_flag, R0_flat,
   rgrid, Nr, j,
   V0, rho_wr,
   tgrid, Nt, time_index, period);
}
else
{
    if (j==Nr) // R=Rmax
    {

Drj = rgrid[j]-rgrid[j-1];
Drjp1 = Drj;
dr[stencil(0,-1)]=2.0/(Drj*(Drj+Drjp1));
dr[stencil(0,0)]=-2.0/(Drj*Drjp1) + 2.0/(Drjp1*(Drj+Drjp1));
//dr[stencil(0,1)]=2.0/(Drjp1*(Drj+Drjp1));
G_dr = 2.0/(Drjp1*(Drj+Drjp1)) * bc_rate_max(alpha_g,
     alpha_m, Gr, kappa,
     //A0, agrid, Na, k,
     W0, wgrid, Nw, i,
```

```
      R0, sigma_r, alpha_r, flat_flag, R0_flat,
      rgrid, Nr, j,
      V0, rho_wr,
      tgrid, Nt, time_index, period);

cr[stencil(0,-1)]=-Drjp1/(Drj*(Drj+Drjp1));
cr[stencil(0,0)]=(Drjp1-Drj)/(Drj*Drjp1) + Drj/(Drjp1*(Drj+Drjp1));
//cr[stencil(0,1)]=Drj/(Drjp1*(Drj+Drjp1));
G_cr += Drj/(Drjp1*(Drj+Drjp1)) * bc_rate_max(alpha_g,
       alpha_m, Gr, kappa,
       //A0, agrid, Na, k,
       W0, wgrid, Nw, i,
       R0, sigma_r, alpha_r, flat_flag, R0_flat,
       rgrid, Nr, j,
       V0, rho_wr,
       tgrid, Nt, time_index, period);
     }
     else
     {
Drj = rgrid[j]-rgrid[j-1];
Drjp1 = rgrid[j+1]-rgrid[j];

dr[stencil(0,-1)]=2.0/(Drj*(Drj+Drjp1));
dr[stencil(0,0)]=-2.0/(Drj*Drjp1);
dr[stencil(0,1)]=2.0/(Drjp1*(Drj+Drjp1));

cr[stencil(0,-1)]=-Drjp1/(Drj*(Drj+Drjp1));
cr[stencil(0,0)]=(Drjp1-Drj)/(Drj*Drjp1);
cr[stencil(0,1)]=Drj/(Drjp1*(Drj+Drjp1));

     }
}
     }

     // Mixted WR
     //printf("Mixted WR.\ n");
     {
// W=Wmin or W=Wmax or R=Rmin or R=Rmax -->> Mixted WR = 0 by Neumann conditions
if ((0<i) && (i<Nw) && (0<j) && (j<Nr)) // Strict interior 0<i<Nw and 0<j<Nv
{
     Dwi = wgrid[i]-wgrid[i-1];
```

```
    Dwip1 = wgrid[i+1]-wgrid[i];
    Drj = rgrid[j]-rgrid[j-1];
    Drjp1 = rgrid[j+1]-rgrid[j];

    wbeta_im1 = -Dwip1/(Dwi*(Dwi+Dwip1));
    wbeta_i0 = (Dwip1-Dwi)/(Dwi*Dwip1);
    wbeta_ip1 = Dwi/(Dwip1*(Dwi+Dwip1));
    rbeta_jm1 = -Drjp1/(Drj*(Drj+Drjp1));
    rbeta_j0 = (Drjp1-Drj)/(Drj*Drjp1);
    rbeta_jp1 = Drj/(Drjp1*(Drj+Drjp1));

    mwr[stencil(0,0)] = wbeta_i0 * rbeta_j0;
    mwr[stencil(-1,0)] = wbeta_im1 * rbeta_j0;
    mwr[stencil(1,0)] = wbeta_ip1 * rbeta_j0;
    mwr[stencil(0,-1)] = wbeta_i0 * rbeta_jm1;
    mwr[stencil(0,1)] = wbeta_i0 * rbeta_jp1;
    mwr[stencil(-1,-1)] = wbeta_im1 * rbeta_jm1;
    mwr[stencil(1,-1)] = wbeta_ip1 * rbeta_jm1;
    mwr[stencil(-1,1)] = wbeta_im1 * rbeta_jp1;
    mwr[stencil(1,1)] = wbeta_ip1 * rbeta_jp1;
}
else // i==0 or i==Ns or j==0 or j==Nr
{
    if (j==0) // R=Rmin
    {
// i==0 or Nw -> Condition on R=Rmin compatible with condition on W=Wmin or Wmax
// 0<i<Nw -> Condition on R=Rmin.
if ((0<i) && (i<Nw))
{
    Dwi = wgrid[i]-wgrid[i-1];
    Dwip1 = wgrid[i+1]-wgrid[i];
    Drjp1 = rgrid[j+1]-rgrid[j];
    Drj = Drjp1;

    wbeta_im1 = -Dwip1/(Dwi*(Dwi+Dwip1));
    wbeta_i0 = (Dwip1-Dwi)/(Dwi*Dwip1);
    wbeta_ip1 = Dwi/(Dwip1*(Dwi+Dwip1));
    rbeta_jm1 = -Drjp1/(Drj*(Drj+Drjp1));
    rbeta_j0 = (Drjp1-Drj)/(Drj*Drjp1);
    rbeta_jp1 = Drj/(Drjp1*(Drj+Drjp1));
```

```
    mwr[stencil(-1,0)] = wbeta_im1 * rbeta_j0;
    mwr[stencil(1,0)] = wbeta_ip1 * rbeta_j0;
    //mwr[stencil(0,-1)] = wbeta_i0 * rbeta_jm1;
    G_mwr += wbeta_i0 * rbeta_jm1 * bc_rate_min(alpha_g,
alpha_m, Gr, kappa,
//A0, agrid, Na, k,
W0, wgrid, Nw, i,
R0, sigma_r, alpha_r, flat_flag, R0_flat,
rgrid, Nr, j,
V0, rho_wr,
tgrid, Nt, time_index, period);
    mwr[stencil(0,1)] = wbeta_i0 * rbeta_jp1;
    //mwr[stencil(-1,-1)] = wbeta_im1 * rbeta_jm1;
    G_mwr += wbeta_im1 * rbeta_jm1 * bc_rate_min(alpha_g,
 alpha_m, Gr, kappa,
 //A0, agrid, Na, k,
 W0, wgrid, Nw, i-1,
 R0, sigma_r, alpha_r, flat_flag, R0_flat,
 rgrid, Nr, j,
 V0, rho_wr,
 tgrid, Nt, time_index, period);
    //mwr[stencil(1,-1)] = wbeta_ip1 * rbeta_jm1;
    G_mwr += wbeta_ip1 * rbeta_jm1 * bc_rate_max(alpha_g,
 alpha_m, Gr, kappa,
 //A0, agrid, Na, k,
 W0, wgrid, Nw, i+1,
 R0, sigma_r, alpha_r, flat_flag, R0_flat,
 rgrid, Nr, j,
 V0, rho_wr,
 tgrid, Nt, time_index, period);
    mwr[stencil(-1,1)] = wbeta_im1 * rbeta_jp1;
    mwr[stencil(1,1)] = wbeta_ip1 * rbeta_jp1;
    mwr[stencil(0,0)] = wbeta_i0 * rbeta_j0
    + 1. * wbeta_im1 * rbeta_jm1 // Neumann
    + 1. * wbeta_i0 * rbeta_jm1 // Neumann
    + 1. * wbeta_ip1 * rbeta_jm1; // Neumann
}
    } // End of R=Rmin -> we do not have done i==0 and i==Nw in j==0.
    if (j==Nr) // R=Rmax
    {
```

```
// i==0 or Nw -> Condition on R=Rmax compatible with condition on W=Wmin or Wmax
// 0<i<Nw -> Condition on R=Rmax.
if ((0<i) && (i<Nw))
{
    Dwi = wgrid[i]-wgrid[i-1];
    Dwip1 = wgrid[i+1]-wgrid[i];
    Drj = rgrid[j]-rgrid[j-1];
    Drjp1 = Drj;

    wbeta_im1 = -Dwip1/(Dwi*(Dwi+Dwip1));
    wbeta_i0 = (Dwip1-Dwi)/(Dwi*Dwip1);
    wbeta_ip1 = Dwi/(Dwip1*(Dwi+Dwip1));
    rbeta_jm1 = -Drjp1/(Drj*(Drj+Drjp1));
    rbeta_j0 = (Drjp1-Drj)/(Drj*Drjp1);
    rbeta_jp1 = Drj/(Drjp1*(Drj+Drjp1));


    mwr[stencil(-1,0)] = wbeta_im1 * rbeta_j0;
    mwr[stencil(1,0)] = wbeta_ip1 * rbeta_j0;
    mwr[stencil(0,-1)] = wbeta_i0 * rbeta_jm1;
    //mwr[stencil(0,1)] = wbeta_i0 * rbeta_jp1;
    G_mwr += wbeta_i0 * rbeta_jp1 * bc_rate_max(alpha_g,
alpha_m, Gr, kappa,
//A0, agrid, Na, k,
W0, wgrid, Nw, i,
R0, sigma_r, alpha_r, flat_flag, R0_flat,
rgrid, Nr, j,
V0, rho_wr,
tgrid, Nt, time_index, period);
    mwr[stencil(-1,-1)] = wbeta_im1 * rbeta_jm1;
    mwr[stencil(1,-1)] = wbeta_ip1 * rbeta_jm1;
    //mwr[stencil(-1,1)] = wbeta_im1 * rbeta_jp1;
    G_mwr += wbeta_im1 * rbeta_jp1 * bc_rate_max(alpha_g,
 alpha_m, Gr, kappa,
 //A0, agrid, Na, k,
 W0, wgrid, Nw, i-1,
 R0, sigma_r, alpha_r, flat_flag, R0_flat,
 rgrid, Nr, j,
 V0, rho_wr,
 tgrid, Nt, time_index, period);
    //msv[stencil(1,1)] = wbeta_ip1 * rbeta_jp1;
```

```
    G_mwr += wbeta_ip1 * rbeta_jp1 * bc_rate_max(alpha_g,
 alpha_m, Gr, kappa,
 //A0, agrid, Na, k,
 W0, wgrid, Nw, i+1,
 R0, sigma_r, alpha_r, flat_flag, R0_flat,
 rgrid, Nr, j,
 V0, rho_wr,
 tgrid, Nt, time_index, period);
    mwr[stencil(0,0)] = wbeta_i0 * rbeta_j0
    + 1. * wbeta_im1 * rbeta_jp1 // Neumann
    + 1. * wbeta_i0 * rbeta_jp1 // Neumann
    + 1. * wbeta_ip1 * rbeta_jp1; // Neumann
}
    } // End of R=Rmax -> we do not have done i==0 and i==Nw in j==Nr.
    if (i==0) // W=Wmin
    {
// j==0 or Nr -> Condition on W=Wmin compatible with condition on R=Rmin or Rmax
// 0<j<Nr -> Condition on W=Wmin.
if ((0<j) && (j<Nr))
{
    Dwip1 = wgrid[i+1]-wgrid[i];
    Dwi = Dwip1;
    Drj = rgrid[j]-rgrid[j-1];
    Drjp1 = rgrid[j+1]-rgrid[j];

    wbeta_im1 = -Dwip1/(Dwi*(Dwi+Dwip1));
    wbeta_i0 = (Dwip1-Dwi)/(Dwi*Dwip1);
    wbeta_ip1 = Dwi/(Dwip1*(Dwi+Dwip1));
    rbeta_jm1 = -Drjp1/(Drj*(Drj+Drjp1));
    rbeta_j0 = (Drjp1-Drj)/(Drj*Drjp1);
    rbeta_jp1 = Drj/(Drjp1*(Drj+Drjp1));


    //mwr[stencil(-1,0)] = wbeta_im1 * rbeta_j0;
    G_mwr += wbeta_im1 * rbeta_j0 * bc_w_min(alpha_g,
     alpha_m, Gr, kappa,
     //A0, agrid, Na, k,
     W0, wgrid, Nw, i,
     R0, sigma_r, alpha_r, flat_flag, R0_flat,
     rgrid, Nr, j,
     V0, rho_wr,
```

```
     tgrid, Nt, time_index, period);
    mwr[stencil(1,0)] = wbeta_ip1 * rbeta_j0;
    mwr[stencil(0,-1)] = wbeta_i0 * rbeta_jm1;
    mwr[stencil(0,1)] = wbeta_i0 * rbeta_jp1;
    //mwr[stencil(-1,-1)] = wbeta_im1 * rbeta_jm1;
    G_mwr += wbeta_im1 * rbeta_jm1 * bc_w_min(alpha_g,
      alpha_m, Gr, kappa,
      //A0, agrid, Na, k,
      W0, wgrid, Nw, i,
      R0, sigma_r, alpha_r, flat_flag, R0_flat,
      rgrid, Nr, j-1,
      V0, rho_wr,
      tgrid, Nt, time_index, period);
    mwr[stencil(1,-1)] = wbeta_ip1 * rbeta_jm1;
    //mwr[stencil(-1,1)] = wbeta_im1 * rbeta_jp1;
    G_mwr += wbeta_im1 * rbeta_jp1 * bc_w_min(alpha_g,
      alpha_m, Gr, kappa,
      //A0, agrid, Na, k,
      W0, wgrid, Nw, i,
      R0, sigma_r, alpha_r, flat_flag, R0_flat,
      rgrid, Nr, j+1,
      V0, rho_wr,
      tgrid, Nt, time_index, period);
    mwr[stencil(1,1)] = wbeta_ip1 * rbeta_jp1;
    mwr[stencil(0,0)] = wbeta_i0 * rbeta_j0
    + 1. * wbeta_im1 * rbeta_jm1 // Neumann
    + 1. * wbeta_im1 * rbeta_j0 // Neumann
    + 1. * wbeta_im1 * rbeta_jp1; // Neumann
}
    } // End of W=Wmin -> we do not have done j==0 or j==Nr in i==0.
    if (i==Nw) // W=Wmax
    {
// j==0 or Nr -> Condition on W=Wmax compatible with condition on R=Rmin or Rmax
// 0<j<Nr -> Condition on W=Wmax.
if ((0<j) && (j<Nr))
{
    Dwi = wgrid[i]-wgrid[i-1];
    Dwip1 = Dwi;
    Drj = rgrid[j]-rgrid[j-1];
    Drjp1 = rgrid[j+1]-rgrid[j];
```

```
wbeta_im1 = -Dwip1/(Dwi*(Dwi+Dwip1));
wbeta_i0 = (Dwip1-Dwi)/(Dwi*Dwip1);
wbeta_ip1 = Dwi/(Dwip1*(Dwi+Dwip1));
rbeta_jm1 = -Drjp1/(Drj*(Drj+Drjp1));
rbeta_j0 = (Drjp1-Drj)/(Drj*Drjp1);
rbeta_jp1 = Drj/(Drjp1*(Drj+Drjp1));


mwr[stencil(-1,0)] = wbeta_im1 * rbeta_j0;
//mwr[stencil(1,0)] = wbeta_ip1 * rbeta_j0;
G_mwr += wbeta_ip1 * rbeta_j0 * bc_w_max(alpha_g,
 alpha_m, Gr, kappa,
 //A0, agrid, Na, k,
 W0, wgrid, Nw, i,
 R0, sigma_r, alpha_r, flat_flag, R0_flat,
 rgrid, Nr, j,
 V0, rho_wr,
 tgrid, Nt, time_index, period);
mwr[stencil(0,-1)] = wbeta_i0 * rbeta_jm1;
mwr[stencil(0,1)] = wbeta_i0 * rbeta_jp1;
mwr[stencil(-1,-1)] = wbeta_im1 * rbeta_jm1;
//mwr[stencil(1,-1)] = wbeta_ip1 * rbeta_jm1;
G_mwr += wbeta_ip1 * rbeta_jm1 * bc_w_max(alpha_g,
  alpha_m, Gr, kappa,
  //A0, agrid, Na, k,
  W0, wgrid, Nw, i,
  R0, sigma_r, alpha_r, flat_flag, R0_flat,
  rgrid, Nr, j-1,
  V0, rho_wr,
  tgrid, Nt, time_index, period);
mwr[stencil(-1,1)] = wbeta_im1 * rbeta_jp1;
//mwr[stencil(1,1)] = wbeta_ip1 * rbeta_jp1;
G_mwr += wbeta_ip1 * rbeta_jp1 * bc_w_max(alpha_g,
  alpha_m, Gr, kappa,
  //A0, agrid, Na, k,
  W0, wgrid, Nw, i,
  R0, sigma_r, alpha_r, flat_flag, R0_flat,
  rgrid, Nr, j+1,
  V0, rho_wr,
  tgrid, Nt, time_index, period);
mwr[stencil(0,0)] = wbeta_i0 * rbeta_j0
```

```
    + 1. * wbeta_ip1 * rbeta_jm1 // Neumann
    + 1. * wbeta_ip1 * rbeta_j0 // Neumann
    + 1. * wbeta_ip1 * rbeta_jp1; // Neumann
}
    } // End of W=Wmax -> we do not have done j==0 or j==Nr in i==Nw.
    // Do the corners now !

    //###### FIX ME
    //###### FIX ME
    //###### FIX ME
    //###### FIX ME

    if ((i==0) && (j==0)) // Corner ! The two Neumann conditions are compatible.
    {
Dwip1 = wgrid[i+1]-wgrid[i];
Dwi = Dwip1;

Drjp1 = rgrid[j+1]-rgrid[j];
Drj = Drjp1;

wbeta_im1 = -Dwip1/(Dwi*(Dwi+Dwip1));
wbeta_i0 = (Dwip1-Dwi)/(Dwi*Dwip1);
wbeta_ip1 = Dwi/(Dwip1*(Dwi+Dwip1));
rbeta_jm1 = -Drjp1/(Drj*(Drj+Drjp1));
rbeta_j0 = (Drjp1-Drj)/(Drj*Drjp1);
rbeta_jp1 = Drj/(Drjp1*(Drj+Drjp1));


//mwr[stencil(-1,0)] = wbeta_im1 * rbeta_j0;
G_mwr += wbeta_im1 * rbeta_j0 * bc_w_min(alpha_g,
 alpha_m, Gr, kappa,
 //A0, agrid, Na, k,
 W0, wgrid, Nw, i,
 R0, sigma_r, alpha_r, flat_flag, R0_flat,
 rgrid, Nr, j,
 V0, rho_wr,
 tgrid, Nt, time_index, period);
mwr[stencil(1,0)] = wbeta_ip1 * rbeta_j0;
//mwr[stencil(0,-1)] = wbeta_i0 * rbeta_jm1;
G_mwr += wbeta_i0 * rbeta_jm1 * bc_rate_min(alpha_g,
    alpha_m, Gr, kappa,
```

```
    //A0, agrid, Na, k,
    W0, wgrid, Nw, i,
    R0, sigma_r, alpha_r, flat_flag, R0_flat,
    rgrid, Nr, j,
    V0, rho_wr,
    tgrid, Nt, time_index, period);
mwr[stencil(0,1)] = wbeta_i0 * rbeta_jp1;
//mwr[stencil(-1,-1)] = wbeta_im1 * rbeta_jm1;
G_mwr += wbeta_im1 * rbeta_jm1 * bc_w_min(alpha_g,
  alpha_m, Gr, kappa,
  //A0, agrid, Na, k,
  W0, wgrid, Nw, i,
  R0, sigma_r, alpha_r, flat_flag, R0_flat,
  rgrid, Nr, j-1,
  V0, rho_wr,
  tgrid, Nt, time_index, period);
//mwr[stencil(1,-1)] = wbeta_ip1 * rbeta_jm1;
G_mwr += wbeta_ip1 * rbeta_jm1 * bc_rate_min(alpha_g,
     alpha_m, Gr, kappa,
     //A0, agrid, Na, k,
     W0, wgrid, Nw, i+1,
     R0, sigma_r, alpha_r, flat_flag, R0_flat,
     rgrid, Nr, j,
     V0, rho_wr,
     tgrid, Nt, time_index, period);
//mwr[stencil(-1,1)] = wbeta_im1 * rbeta_jp1;
G_mwr += wbeta_im1 * rbeta_jp1 * bc_w_min(alpha_g,
  alpha_m, Gr, kappa,
  //A0, agrid, Na, k,
  W0, wgrid, Nw, i,
  R0, sigma_r, alpha_r, flat_flag, R0_flat,
  rgrid, Nr, j+1,
  V0, rho_wr,
  tgrid, Nt, time_index, period);
mwr[stencil(1,1)] = wbeta_ip1 * rbeta_jp1;
mwr[stencil(0,0)] = wbeta_i0 * rbeta_j0
+ 1. * wbeta_im1 * rbeta_jm1 // Neumann
+ 1. * wbeta_im1 * rbeta_j0 // Neumann
+ 1. * wbeta_im1 * rbeta_jp1 // Neumann
+ 1. * wbeta_i0 * rbeta_jm1 // Neumann
+ 1. * wbeta_ip1 * rbeta_jm1; // Neumann
```

```
    }
    if ((i==0) && (j==Nr)) // Corner ! The two Neumann conditions are compatible
    {
Dwip1 = wgrid[i+1]-wgrid[i];
Dwi = Dwip1;

Drj = rgrid[j]-rgrid[j-1];
Drjp1 = Drj;

wbeta_im1 = -Dwip1/(Dwi*(Dwi+Dwip1));
wbeta_i0 = (Dwip1-Dwi)/(Dwi*Dwip1);
wbeta_ip1 = Dwi/(Dwip1*(Dwi+Dwip1));
rbeta_jm1 = -Drjp1/(Drj*(Drj+Drjp1));
rbeta_j0 = (Drjp1-Drj)/(Drj*Drjp1);
rbeta_jp1 = Drj/(Drjp1*(Drj+Drjp1));


//mwr[stencil(-1,0)] = wbeta_im1 * rbeta_j0;
G_mwr += wbeta_im1 * rbeta_j0 * bc_w_min(alpha_g,
 alpha_m, Gr, kappa,
 //A0, agrid, Na, k,
 W0, wgrid, Nw, i,
 R0, sigma_r, alpha_r, flat_flag, R0_flat,
 rgrid, Nr, j,
 V0, rho_wr,
 tgrid, Nt, time_index, period);
mwr[stencil(1,0)] = wbeta_ip1 * rbeta_j0;
mwr[stencil(0,-1)] = wbeta_i0 * rbeta_jm1;
//mwr[stencil(0,1)] = wbeta_i0 * rbeta_jp1;
G_mwr += wbeta_i0 * rbeta_jp1 * bc_rate_max(alpha_g,
    alpha_m, Gr, kappa,
    //A0, agrid, Na, k,
    W0, wgrid, Nw, i,
    R0, sigma_r, alpha_r, flat_flag, R0_flat,
    rgrid, Nr, j,
    V0, rho_wr,
    tgrid, Nt, time_index, period);
//mwr[stencil(-1,-1)] = wbeta_im1 * rbeta_jm1;
G_mwr += wbeta_im1 * rbeta_jm1 * bc_w_min(alpha_g,
  alpha_m, Gr, kappa,
  //A0, agrid, Na, k,
```

```
  W0, wgrid, Nw, i,
  R0, sigma_r, alpha_r, flat_flag, R0_flat,
  rgrid, Nr, j-1,
  V0, rho_wr,
  tgrid, Nt, time_index, period);
mwr[stencil(1,-1)] = wbeta_ip1 * rbeta_jm1;
//mwr[stencil(-1,1)] = wbeta_im1 * rbeta_jp1;
G_mwr += wbeta_im1 * rbeta_jp1 * bc_w_min(alpha_g,
  alpha_m, Gr, kappa,
  //A0, agrid, Na, k,
  W0, wgrid, Nw, i,
  R0, sigma_r, alpha_r, flat_flag, R0_flat,
  rgrid, Nr, j+1,
  V0, rho_wr,
  tgrid, Nt, time_index, period);
//mwr[stencil(1,1)] = wbeta_ip1 * rbeta_jp1;
G_mwr += wbeta_ip1 * rbeta_jp1 * bc_rate_max(alpha_g,
     alpha_m, Gr, kappa,
     //A0, agrid, Na, k,
     W0, wgrid, Nw, i,
     R0, sigma_r, alpha_r, flat_flag, R0_flat,
     rgrid, Nr, j,
     V0, rho_wr,
     tgrid, Nt, time_index, period);
mwr[stencil(0,0)] = wbeta_i0 * rbeta_j0
+ 1. * wbeta_im1 * rbeta_jm1 // Neumann
+ 1. * wbeta_im1 * rbeta_j0 // Neumann
+ 1. * wbeta_im1 * rbeta_jp1 // Neumann
+ 1. * wbeta_i0 * rbeta_jp1 // Neumann
+ 1. * wbeta_ip1 * rbeta_jp1; // Neumann
    }
    if ((i==Nw) && (j==0)) // Corner ! The two Neumann conditions are compatible
    {
Dwi = wgrid[i]-wgrid[i-1];
Dwip1 = Dwi;

Drjp1 = rgrid[j+1]-rgrid[j];
Drj = Drjp1;

wbeta_im1 = -Dwip1/(Dwi*(Dwi+Dwip1));
wbeta_i0 = (Dwip1-Dwi)/(Dwi*Dwip1);
```

```
wbeta_ip1 = Dwi/(Dwip1*(Dwi+Dwip1));
rbeta_jm1 = -Drjp1/(Drj*(Drj+Drjp1));
rbeta_j0 = (Drjp1-Drj)/(Drj*Drjp1);
rbeta_jp1 = Drj/(Drjp1*(Drj+Drjp1));


mwr[stencil(-1,0)] = wbeta_im1 * rbeta_j0;
//mwr[stencil(1,0)] = wbeta_ip1 * rbeta_j0;
G_mwr += wbeta_ip1 * rbeta_j0 * bc_w_max(alpha_g,
 alpha_m, Gr, kappa,
 //A0, agrid, Na, k,
 W0, wgrid, Nw, i,
 R0, sigma_r, alpha_r, flat_flag, R0_flat,
 rgrid, Nr, j,
 V0, rho_wr,
 tgrid, Nt, time_index, period);
//mwr[stencil(0,-1)] = wbeta_i0 * rbeta_jm1;
G_mwr += wbeta_i0 * rbeta_jm1 * bc_rate_min(alpha_g,
    alpha_m, Gr, kappa,
    //A0, agrid, Na, k,
    W0, wgrid, Nw, i,
    R0, sigma_r, alpha_r, flat_flag, R0_flat,
    rgrid, Nr, j,
    V0, rho_wr,
    tgrid, Nt, time_index, period);
mwr[stencil(0,1)] = wbeta_i0 * rbeta_jp1;
//mwr[stencil(-1,-1)] = wbeta_im1 * rbeta_jm1;
G_mwr += wbeta_im1 * rbeta_jm1 * bc_rate_min(alpha_g,
     alpha_m, Gr, kappa,
     //A0, agrid, Na, k,
     W0, wgrid, Nw, i-1,
     R0, sigma_r, alpha_r, flat_flag, R0_flat,
     rgrid, Nr, j,
     V0, rho_wr,
     tgrid, Nt, time_index, period);
//mwr[stencil(1,-1)] = wbeta_ip1 * rbeta_jm1;
G_mwr += wbeta_ip1 * rbeta_jm1 * bc_w_max(alpha_g,
  alpha_m, Gr, kappa,
  //A0, agrid, Na, k,
  W0, wgrid, Nw, i,
  R0, sigma_r, alpha_r, flat_flag, R0_flat,
```

```
  rgrid, Nr, j-1,
  V0, rho_wr,
  tgrid, Nt, time_index, period);
mwr[stencil(-1,1)] = wbeta_im1 * rbeta_jp1;
//mwr[stencil(1,1)] = wbeta_ip1 * rbeta_jp1;
G_mwr += wbeta_ip1 * rbeta_jp1 * bc_w_max(alpha_g,
  alpha_m, Gr, kappa,
  //A0, agrid, Na, k,
  W0, wgrid, Nw, i,
  R0, sigma_r, alpha_r, flat_flag, R0_flat,
  rgrid, Nr, j+1,
  V0, rho_wr,
  tgrid, Nt, time_index, period);

mwr[stencil(0,0)] = wbeta_i0 * rbeta_j0
+ 1. * wbeta_ip1 * rbeta_jm1 // Neumann
+ 1. * wbeta_ip1 * rbeta_j0 // Neumann
+ 1. * wbeta_ip1 * rbeta_jp1 // Neumann
+ 1. * wbeta_i0 * rbeta_jm1 // Neumann
+ 1. * wbeta_im1 * rbeta_jm1; // Neumann
    }
    if ((i==Nw) && (j==Nr)) // Corner ! The two Neumann conditions are compatibl
    {

Dwi = wgrid[i]-wgrid[i-1];
Dwip1 = Dwi;

Drj = rgrid[j]-rgrid[j-1];
Drjp1 = Drj;

wbeta_im1 = -Dwip1/(Dwi*(Dwi+Dwip1));
wbeta_i0 = (Dwip1-Dwi)/(Dwi*Dwip1);
wbeta_ip1 = Dwi/(Dwip1*(Dwi+Dwip1));
rbeta_jm1 = -Drjp1/(Drj*(Drj+Drjp1));
rbeta_j0 = (Drjp1-Drj)/(Drj*Drjp1);
rbeta_jp1 = Drj/(Drjp1*(Drj+Drjp1));


mwr[stencil(-1,0)] = wbeta_im1 * rbeta_j0;
//mwr[stencil(1,0)] = wbeta_ip1 * rbeta_j0;
G_mwr += wbeta_ip1 * rbeta_j0 * bc_w_max(alpha_g,
```

```
 alpha_m, Gr, kappa,
 //A0, agrid, Na, k,
 W0, wgrid, Nw, i,
 R0, sigma_r, alpha_r, flat_flag, R0_flat,
 rgrid, Nr, j,
 V0, rho_wr,
 tgrid, Nt, time_index, period);
mwr[stencil(0,-1)] = wbeta_i0 * rbeta_jm1;
//mwr[stencil(0,1)] = wbeta_i0 * rbeta_jp1;
G_mwr += wbeta_i0 * rbeta_jp1 * bc_rate_max(alpha_g,
    alpha_m, Gr, kappa,
    //A0, agrid, Na, k,
    W0, wgrid, Nw, i,
    R0, sigma_r, alpha_r, flat_flag, R0_flat,
    rgrid, Nr, j,
    V0, rho_wr,
    tgrid, Nt, time_index, period);
mwr[stencil(-1,-1)] = wbeta_im1 * rbeta_jm1;
//mwr[stencil(1,-1)] = wbeta_ip1 * rbeta_jm1;
G_mwr += wbeta_ip1 * rbeta_jm1 * bc_w_max(alpha_g,
  alpha_m, Gr, kappa,
  //A0, agrid, Na, k,
  W0, wgrid, Nw, i,
  R0, sigma_r, alpha_r, flat_flag, R0_flat,
  rgrid, Nr, j-1,
  V0, rho_wr,
  tgrid, Nt, time_index, period);
//mwr[stencil(-1,1)] = wbeta_im1 * rbeta_jp1;
G_mwr += wbeta_im1 * rbeta_jp1 * bc_rate_max(alpha_g,
     alpha_m, Gr, kappa,
     //A0, agrid, Na, k,
     W0, wgrid, Nw, i-1,
     R0, sigma_r, alpha_r, flat_flag, R0_flat,
     rgrid, Nr, j,
     V0, rho_wr,
     tgrid, Nt, time_index, period);
//mwr[stencil(1,1)] = wbeta_ip1 * rbeta_jp1;
G_mwr += wbeta_ip1 * rbeta_jp1 * bc_w_max(alpha_g,
  alpha_m, Gr, kappa,
  //A0, agrid, Na, k,
  W0, wgrid, Nw, i,
```

```
  R0, sigma_r, alpha_r, flat_flag, R0_flat,
  rgrid, Nr, j+1,
  V0, rho_wr,
  tgrid, Nt, time_index, period);

mwr[stencil(0,0)] = wbeta_i0 * rbeta_j0
+ 1. * wbeta_ip1 * rbeta_jm1 // Neumann
+ 1. * wbeta_ip1 * rbeta_j0 // Neumann
+ 1. * wbeta_ip1 * rbeta_jp1 // Neumann
+ 1. * wbeta_i0 * rbeta_jp1 // Neumann
+ 1. * wbeta_im1 * rbeta_jp1; // Neumann
    }
    //*/

    //###### FIX ME
    //###### FIX ME
    //###### FIX ME
    //###### FIX ME

}
    }

    // Central point for order_0
    Matrix0[i][j][0] = mixted_wr * mwr[0];
    Matrix1[i][j][0] = order_0 * 0.5 + convection_w * cw[0] + diffusion_w * dw[0
    Matrix2[i][j][0] = order_0 * 0.5 + convection_r * cr[0] + diffusion_r * dr[0

    for (st=1;st<11;st++)
    {
Matrix0[i][j][st] = mixted_wr * mwr[st];
Matrix1[i][j][st] = convection_w * cw[st] + diffusion_w * dw[st];
Matrix2[i][j][st] = convection_r * cr[st] + diffusion_r * dr[st];
    }

    // Second members
    G0[i][j] = mixted_wr * G_mwr;
    G1[i][j] = convection_w * G_cw + diffusion_w * G_dw;
    G2[i][j] = convection_r * G_cr + diffusion_r * G_dr;

    G1[i][j] += supplementary_terms(alpha_m, Gr, kappa, tgrid, Nt, time_index, p
```

```
}
    }

    free(mwr);
    free(dr);
    free(cr);
    free(dw);
    free(cw);
}

static void compute_explicit_syslin_all_matrix(double coeff,
       double *wgrid, int Nw, double *rgrid, int Nr,
       double ***Matrix0, double ***Matrix1, double ***Matrix2,
       double **G0nm1, double **G1nm1, double **G2nm1,
       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<Nr+1; j++)
{
    val = Unm1[i][j];
    for (st=0; st<11; st++)
    {
if (it_exists_stencil(i, Nw, j, Nr, 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, double *rgrid, int Nr,
    double ***Matrix0, double ***Matrix1, double ***Matrix2,
    double **G0nm1, double **G1nm1, double **G2nm1,
    double **Unm1, double **Y0, double **Sortie)
{
    int i, j, st;
    double val;
    int istencil, jstencil;

    val=0.;

    for (j=0; j<Nr+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, Nr, 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, double *rgrid, int Nr,
    double ***Matrix0, double ***Matrix1, double ***Matrix2,
    double **G0, double **G1, double **G2,
    double **rhs, double **lhs)
```

```
{
    int i, j;

    // Only points from i=0 to i=Nw.
    // Only points from j=0 to j=Nr.
    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<Nr+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_rate_matrix(double coeff,
    double *wgrid, int Nw, double *rgrid, int Nr,
    double ***Matrix0, double ***Matrix1, double ***Matrix2,
    double **G0nm1, double **G1nm1, double **G2nm1,
    double **Unm1, double **Y1, double **Sortie)
{
    int i, j, st;
    double val;
    int istencil, jstencil;

    val=0.;

    for (i=0; i<Nw+1; i++)
    {
for (j=0; j<Nr+1; j++)
{
    val = Y1[i][j];
    for (st=0; st<11; st++)
    {
if (it_exists_stencil(i, Nw, j, Nr, 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_rate_matrix(double coeff,
    double *wgrid, int Nw, double *rgrid, int Nr,
    double ***Matrix0, double ***Matrix1, double ***Matrix2,
    double **G0, double **G1, double **G2,
    double **rhs, double **lhs)
{
    int i, j;

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

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

    for (i=0; i<Nw+1; i++)
    {
Working_Matrix = pnl_band_mat_create(Nr+1,Nr+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<Nr-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, Nr-1, Nr-1-2, Matrix2[i][Nr-1][3]);
pnl_band_mat_set (Working_Matrix, Nr-1, Nr-1-1, Matrix2[i][Nr-1][4]);
pnl_band_mat_set (Working_Matrix, Nr-1, Nr-1+0, Matrix2[i][Nr-1][0]);
pnl_band_mat_set (Working_Matrix, Nr-1, Nr-1+1, Matrix2[i][Nr-1][5]);
//pnl_band_mat_set (Working_Matrix, Nr-1, Nr-1+2, Matrix2[i][Nr-1][6]);
pnl_band_mat_set (Working_Matrix, Nr, Nr-2, Matrix2[i][Nr][3]);
pnl_band_mat_set (Working_Matrix, Nr, Nr-1, Matrix2[i][Nr][4]);
pnl_band_mat_set (Working_Matrix, Nr, Nr+0, Matrix2[i][Nr][0]);
//pnl_band_mat_set (Working_Matrix, Nr, Nr+1, Matrix2[i][Nr][5]);
//pnl_band_mat_set (Working_Matrix, Nr, Nr+2, Matrix2[i][Nr][6]);

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

// Build the vectors.
for (j=0; j<Nr+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<Nr+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 R0, double sigma_r, double alpha_r, int flat_flag, double R0_flat,
       double *rgrid, int Nr,
       double V0, double rho_wr,
       double *tgrid, int Nt,
       double t_begin, double t_end, int index_t_begin, int period, int n_step_b
       double theta, int scheme,
       // 2-vectors
       double **Unm1, double **Utmp, double **Y0, double **Y1, //double **Y2, //
       double **G0nm1, double **G1nm1, double **G2nm1, //double **G3nm1,
       //double **G0n;
       double **G1n, double **G2n, //double **G3n,
       double **GSecondMember,
       // Matrix (=3-vectors)
       double ***Matrix0nm1, double ***Matrix1nm1, double ***Matrix2nm1, //doubl
       //double ***Matrix0n;
       double ***Matrix1n, double ***Matrix2n) //double ***Matrix3n)
{
    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<Nr+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",period,
 printf("It corresponds to the temporal interval [%f,%f[ which contains %f.\ n",
 t_begin, t_end, tgrid[time_index]);
 //*/

if (scheme==0) // Douglas scheme
{
    //print_2vector(Unm1, Nw, Nr);


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

 alpha_m, Gr, kappa,

 W0, wgrid, Nw,
 R0, sigma_r, alpha_r, flat_flag, R0_flat, rgrid, Nr,
 V0, rho_wr,
 tgrid, Nt, time_index-1, period,
 Matrix0nm1, Matrix1n, Matrix2n,
 G0nm1, G1n, G2n);
//printf("Build matrix at step time_index (in t-present=tau-present)\ n");
// Compute the elements at time step time_index.
build_all_matrix(alpha_g,

 alpha_m, Gr, kappa,
```

```
W0, wgrid, Nw,
R0, sigma_r, alpha_r, flat_flag, R0_flat, rgrid, Nr,
V0, rho_wr,
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+G2+G3)[n-
   //Y0 = U[n-1] + Dt (A0+A1+A2+A3)[n-1] * U[n-1] + Dt (G0+G1+G2+G3)[n-1]
   compute_explicit_syslin_all_matrix(deltat,
      wgrid, Nw, rgrid, Nr,
      Matrix0nm1, Matrix1nm1, Matrix2nm1,
      G0nm1, G1nm1, G2nm1,
      Unm1, Y0);
   //print_2vector(Y0, Nw, Nr);

   //printf("Utmp = Y0 - theta*Dt A1[n-1] * U[n-1] - theta*Dt G1[n-1].\ n");
   //Utmp = Y0 - theta*Dt A1[n-1] * U[n-1] - theta*Dt G1[n-1]
   computation_explicit_syslin_spot_matrix(theta * deltat,
   wgrid, Nw, rgrid, Nr,
   Matrix0nm1, Matrix1nm1, Matrix2nm1,
   G0nm1, G1nm1, G2nm1,
   Unm1, Y0, Utmp);
   //print_2vector(Utmp, Nw, Nr);

   //printf("Y1 = (Id - theta*Dt A1[n])^-1 ( Utmp + theta*Dt G1[n] ).\ n");
   //Y1 = (Id - theta*Dt A1[n])^-1 ( Utmp + theta*Dt G1[n] )
   computation_implicit_syslin_spot_matrix(theta * deltat,
   wgrid, Nw, rgrid, Nr,
   Matrix0nm1, Matrix1n, Matrix2n,
   G0nm1, G1n, G2n,
   Utmp, Y1);
   //print_2vector(Y1, Nw, Nr);

   //printf("Utmp = Y1 - theta*Dt A2[n-1] * U[n-1] - theta*Dt G2[n-1].\ n");
   //Utmp = Y1 - theta*Dt A2[n-1] * U[n-1] - theta*Dt G2[n-1]
   computation_explicit_syslin_rate_matrix(theta * deltat,
   wgrid, Nw, rgrid, Nr,
```

```
    Matrix0nm1, Matrix1nm1, Matrix2nm1,
    G0nm1, G1nm1, G2nm1,
    Unm1, Y1, Utmp);
    //print_2vector(Utmp, Nw, Nr);

    //printf("Y2 = (Id - theta*Dt A2[n])^-1 ( Utmp + theta*Dt G2[n] ).\ n");
    //Y2 = (Id - theta*Dt A2[n])^-1 ( Utmp + theta*Dt G2[n] )
    computation_implicit_syslin_rate_matrix(theta * deltat,
    wgrid, Nw, rgrid, Nr,
    Matrix0nm1, Matrix1n, Matrix2n,
    G0nm1, G1n, G2n,
    Utmp, Unm1); // /!\  with Y2=Un which becomes Unm1 for the next loop.
    //print_2vector(Unm1, Nw, 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<Nr+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 R0, double sigma_r, double alpha_r, int flat_flag, double R0_flat,
    double *rgrid, int Nr,
    double V0, double rho_wr,
    double *tgrid, int number_of_periods, int n_step_by_period, int Nt,
    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, IndexR;
    double price, delta;
    double price_wd_rd, price_wd_ru;
```

```
    double price_wu_rd, price_wu_ru;

    // 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((Nr+1) * sizeof(double));
    G1nm1[i] = (double*) malloc((Nr+1) * sizeof(double));
    G2nm1[i] = (double*) malloc((Nr+1) * sizeof(double));
    //G3nm1[i] = (double*) malloc((Nr+1) * sizeof(double));
    //G0n[i] = (double*) malloc((Nr+1) * sizeof(double));
    G1n[i] = (double*) malloc((Nr+1) * sizeof(double));
    G2n[i] = (double*) malloc((Nr+1) * sizeof(double));
    //G3n[i] = (double*) malloc((Nr+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((Nr+1) * sizeof(double));
    U_new[i] = (double*) malloc((Nr+1) * sizeof(double));
    Y0[i] = (double*) malloc((Nr+1) * sizeof(double));
    Y1[i] = (double*) malloc((Nr+1) * sizeof(double));
    //Y2[i] = (double*) malloc((Nr+1) * sizeof(double));
    //Y3[i] = (double*) malloc((Nr+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((Nr+1) * sizeof(double));
U_3_new[k][i] = (double*) malloc((Nr+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((Nr+1) * sizeof(double*));
    Matrix1nm1[i] = (double**) malloc((Nr+1) * sizeof(double*));
    Matrix2nm1[i] = (double**) malloc((Nr+1) * sizeof(double*));
    //Matrix3nm1[i] = (double**) malloc((Nr+1) * sizeof(double*));
    //Matrix0n[i] = (double**) malloc((Nr+1) * sizeof(double*));
    Matrix1n[i] = (double**) malloc((Nr+1) * sizeof(double*));
    Matrix2n[i] = (double**) malloc((Nr+1) * sizeof(double*));
    //Matrix3n[i] = (double**) malloc((Nr+1) * sizeof(double*));
    for (j=0; j<Nr+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<Nr+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<Nr+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<Nr+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<Nr+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((Nr+1) * sizeof(double));
    }
    for (i=0; i<Nw+1; i++)
    {
for (j=0; j<Nr+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<Nr+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]-Gr,0.));
// U_3_old[k][i][j] = MAX(wgrid[i],agrid[k]-kappa*MAX(agrid[k]-Gr,0.));
    }
}
    }


    // 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_step_by_per
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 and sub-ac
    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<Nr+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;indx_gamma++)
    {
// Define the strategy corresponding to gamma_i.
```

```
gamma=gamma_vect[indx_gamma];
withdrawal = gamma*Gr;

//if (j==0) printf("\ t gamma=%f, withdrawal=%f-> agrid[k]=%f\ n",gamma, withdra

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

    // Withdrawal not exceeding contract amount.
    if (gamma<=1.)
    {
a_value=agrid[k]-withdrawal; // 0 <= a_value <= agrid[i]
// Find the a_value in the agrid, then start at beginning.
indx_agrid=0;
// Move but never pass agrid[i].
while ((agrid[indx_agrid]<a_value)&&(indx_agrid<Na)) indx_agrid++;
// There is no interpolation for agrid, since it contains all Gr multiples.

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

//if (j==0) printf("\ t a_value=%f, indx_agrid=%d, w_value=%f, indx_wgrid=%d \ n


if (indx_wgrid==0) // w_value <= wgrid[0] thus w_value==0.
{
    new_U=U_3_new[indx_agrid][0][j];
    //if (j==0) printf("\ t indw_wgrid=0 -------->new_U=%f\ n",new_U);

}
else
{
    //if(indx_wgrid>Nw) // Impossible...
    //{
    //printf("Out of range in withdrawal event.");
```

```
    //printf("%d %f %f", indx_wgrid, w_value, wgrid[indx_wgrid]);
    //new_U=U_tmp[Nw][j][k];
    //}
    //else
    //{
    val_next=U_3_new[indx_agrid][indx_wgrid][j];
    val_prev=U_3_new[indx_agrid][indx_wgrid-1][j];
    new_U=val_next+(val_next-val_prev)
    *(w_value-wgrid[indx_wgrid])/(wgrid[indx_wgrid]-wgrid[indx_wgrid-1]);
    //if (j==0) printf("\ t indx_wgrid>0 -------->val_prev=%f, val_next=%f,\ n",

    //}
}

//if (j==0) printf("\ t -------->new_U=%f, test_<_value=%f\ n",new_U, new_U+with


if (worst_value <= new_U + withdrawal)
{
    worst_value = new_U + withdrawal;
    indx_gamma_of_worst_value = indx_gamma;
    //if (j==0) printf("Withdrawal event chosen, new value = %f\ n",worst_value)
}
    }//End of withdrawal event.

    // Partial or full surrender event.
    else // i.e. (gamma>1.)
    {
a_value=agrid[k]-withdrawal; // 0 <= a_value <= agrid[i]
// Find the a_value in the agrid, then start at beginning.
indx_agrid=0;
// Move but never pass agrid[i].
while ((agrid[indx_agrid]<a_value)&&(indx_agrid<Na)) indx_agrid++;
// There is no interpolation for agrid, since it contains all Gr multiples.

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

```
//if (j==0) printf("\ t a_value=%f, indx_agrid=%d, w_value=%f, indx_wgrid=%d \ n



if (indx_wgrid==0)
{
    new_U=U_3_new[indx_agrid][0][j];
    //if (j==0) printf("\ t indw_wgrid=0 -------->new_U=%f\ n",new_U);
}
else
{
    val_next=U_3_new[indx_agrid][indx_wgrid][j];
    val_prev=U_3_new[indx_agrid][indx_wgrid-1][j];
    new_U=val_next+(val_next-val_prev)
    *(w_value-wgrid[indx_wgrid])/(wgrid[indx_wgrid]-wgrid[indx_wgrid-1]);
    //if (j==0) printf("\ t indx_wgrid>0 -------->val_prev=%f, val_next=%f,\ n",
}

//if (j==0) printf("\ t -------->new_U=%f, test_>_value=%f\ n",new_U, new_U + (1

if (worst_value <= new_U + (1.-kappa) * withdrawal +  kappa * Gr)
{
    worst_value = new_U + (1.-kappa) * withdrawal +  kappa * Gr;
    indx_gamma_of_worst_value = indx_gamma;
    //if (j==0) printf("Surrender event chosen, new value = %f\ n",worst_value);
}
    }//End of partial or full surrender event.
} // End of test if Withdrawal or surrender not exceeding amount in benefit base
//else
//if (j==0) printf("\ t Withdrawal exceeds benefit base\ n");
    }//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],agrid[k]-kappa*MAX(ag
    //U_3_old[k][i][j] = MAX(worst_value,0.); // To avoid negative prices. ???
    if (period==number_of_periods)
    {
U_3_old[k][i][j] = MAX(wgrid[i],agrid[k]-kappa*MAX(agrid[k]-Gr,0.));
    }
    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,Nr);
//printf("-----------------------------------\ n");


// Compute the evolution between period and period-1.
//printf("Compute the evolution between period %d and period %d.\ n",period,peri
///*
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<Nr+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,
      R0, sigma_r, alpha_r, flat_flag, R0_flat,
      rgrid, Nr,
      V0, rho_wr,
      tgrid, Nt, t_begin, t_end, index_t_begin, period, n_step_by_period,
      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, //Matrix0n,
      Matrix1n, Matrix2n); //MatrixA3n);

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

//printf("-----------------------------------\ n");
//print_3vector(U_3_new,Na,Nw,Nr);
//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 rgrid corresponding to the rate R0 of the asset.
    IndexR = lower_index(rgrid,Nr+1,R0);

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

```
    //printf("Value = %f\ n",U_3_new[IndexA][IndexW][IndexR]);

    // First compute the delta (using price as temporary variable). We do an int
    price_wd_rd = U_3_new[IndexA][IndexW+1][IndexR];
    price_wd_ru = U_3_new[IndexA][IndexW+1][IndexR+1];
    price_wu_rd = U_3_new[IndexA][IndexW+2][IndexR];
    price_wu_ru = U_3_new[IndexA][IndexW+2][IndexR+1];

    price=double_interpolation(price_wd_rd, price_wd_ru, price_wu_rd, price_wu_r
       wgrid[IndexW+1], wgrid[IndexW+2],
       rgrid[IndexR], rgrid[IndexR+1],
       wgrid[IndexW+1], V0);

    price_wd_rd = U_3_new[IndexA][IndexW][IndexR];
    price_wd_ru = U_3_new[IndexA][IndexW][IndexR+1];
    price_wu_rd = U_3_new[IndexA][IndexW+1][IndexR];
    price_wu_ru = U_3_new[IndexA][IndexW+1][IndexR+1];
    //printf("price_wd_rd=%f, price_wd_ru=%f, price_wu_rd=%f, price_wu_ru=%f\ n"

    delta =(price - double_interpolation(price_wd_rd, price_wd_ru, price_wu_rd,
 wgrid[IndexW], wgrid[IndexW+1],
 rgrid[IndexR], rgrid[IndexR+1],
 wgrid[IndexW], R0) )
    /(wgrid[IndexW+1] - wgrid[IndexW]);

    // Next compute the price. We do an interpolation.
    price=double_interpolation(price_wd_rd, price_wd_ru, price_wu_rd, price_wu_r
       wgrid[IndexW], wgrid[IndexW+1],
       rgrid[IndexR], rgrid[IndexR+1],
       W0, R0);
    //printf("price=%f\ n", price);


    // Memory desallocations.
    {
// Memory desallocation of matrix (=3-vectors).
for (i=0; i<Nw+1; i++)
{
    for (j=0; j<Nr+1; j++)
    {
free(MatrixOnm1[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_tabular,

 double A0, double *agrid, int Na,
 double W0, double *wgrid, int Nw,
 double R0, double sigma_r, double alpha_r, int flat_flag, double R0_flat,
 double *rgrid, int Nr,
 double V0, double rho_wr,
 double *tgrid, int number_of_periods, int n_step_by_period, int Nt,
 double *gamma_vect, int number_of_control_values,

 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,
   R0, sigma_r, alpha_r, flat_flag, R0_flat, rgrid, Nr,
   V0, rho_wr,
   local_tgrid, number_of_periods, local_n_step_by_period, local_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,
   R0, sigma_r, alpha_r, flat_flag, R0_flat, rgrid, Nr,
   V0, rho_wr,
   local_tgrid, number_of_periods, local_n_step_by_period, local_Nt,
   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,
      R0, sigma_r, alpha_r, flat_flag, R0_flat, rgrid, Nr,
      V0, rho_wr,
      local_tgrid, number_of_periods, local_n_step_by_period, local_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++;

    }

    pr_old = compute_price(alpha_old,

   alpha_m, Gr, kappa_tabular,

   A0, agrid, Na,
   W0, wgrid, Nw,
   R0, sigma_r, alpha_r, flat_flag, R0_flat, rgrid, Nr,
   V0, rho_wr,
   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,
   R0, sigma_r, alpha_r, flat_flag, R0_flat, rgrid, Nr,
   V0, rho_wr,
   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,
      R0, sigma_r, alpha_r, flat_flag, R0_flat, rgrid, Nr,
      V0, rho_wr,
      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;

}
free(local_tgrid);
    return alpha;

}



static int FD_gmwb_bshw(double W0, double maturity, double R0, double V0,int fla
{
   // 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;
    // Compute the mean reversion from Zero Coupon Bond data curve.
    int NinterpolZCB;
    int n_price;
    char *init_tr;
    double *Pm;
    double *tm;
    double *initial_yield;
    double *t_interpol;
    double *forward;
    double *derive_forward;

    // Numerical parameters
    int Na, Nt;
    double *agrid, *wgrid, *rgrid, *tgrid;
    double Amax, Aleft, Aright, Coeff_a;
    double Wmax, Wleft, Wright, Coeff_w;
    double Rmax, Center_r, Coeff_r;
    int n_step_by_period;
    int scheme;
    double theta;
    // Variables for method of pricing
    double alpha_old, alpha_new;
double R0_flat;
    init_tr = curve;

    A0=W0;
    V0=SQR(V0);
    // 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 <= 5)
    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
    R0_flat = R0;


    // 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
    // Rate
    Rmax = 10.*R0;
    Center_r = R0;
    Coeff_r = Rmax/400.;
    // Scheme numerical parameters.
    theta=0.5;//Theta schema
    scheme=0.;//Douglas Scheme
```

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




    ////////////////////////////////////
    // Compute agrid, wgrid and rgrid //
    ////////////////////////////////////
    // Memory allocation of 1-vectors.
    agrid=(double *)malloc((Na+1)*sizeof(double));
    wgrid=(double *)malloc((Nw+1)*sizeof(double));
    rgrid=(double *)malloc((Nr+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_rate(Rmax, Center_r, Coeff_r, rgrid, Nr);


    ////////////////////////////////////////////////////////////////
    // Compute tgrid, initial_forward and initial_derive_forward. //
    ////////////////////////////////////////////////////////////////
    // Compute the mean reversion from Zero Coupon Bond data curve.
    // Calibration on the zero coupon bond data table.

    // Read the values of pm and tm.
    Pm = (double *)malloc(200 * sizeof(double));
    tm = (double *)malloc(200 * sizeof(double));
    n_price = lecture_tr(init_tr, Pm, tm);
    // We search in initialyield.dat the biggest value before time T.
    if (maturity > tm[n_price - 1])
    {
printf("\ nError : Maturity bigger than the last time value entered in %s\ n\ n"
    }
    init_tr = (char*) malloc (sizeof (char));
    free(init_tr);

    // Compute interpolation on a very fine grid given pm, tm and n_price. Retur
    // Moins que 1.000.000 mais quand même plus que n_step_by_period.
    NinterpolZCB = MAX(Nt,-MAX(-n_step_by_period*1000,-1000000));
    NinterpolZCB = tm[n_price - 1] * NinterpolZCB;
```

```
    initial_yield = (double *)malloc((NinterpolZCB + 1) * sizeof(double));
    t_interpol = (double *)malloc((NinterpolZCB + 1) * sizeof(double));
    for (i = 0; i <= NinterpolZCB; i++)
    {
initial_yield[i] = 0.;
t_interpol[i] = i * (maturity/NinterpolZCB);
    }
    interpole(n_price, NinterpolZCB, Pm, tm, initial_yield, t_interpol);
    free(Pm);
    free(tm);

    // Compute the forward rate and the derivative of the forward rate given a v
    forward = (double *)malloc((NinterpolZCB + 1) * sizeof(double));
    derive_forward = (double *)malloc((NinterpolZCB + 1) * sizeof(double));
    for (i = 0; i <= NinterpolZCB; i++)
    {
forward[i] = 0.;
derive_forward[i] = 0.;
    }
    compute_forward(initial_yield, t_interpol, NinterpolZCB, forward, derive_for

    // We obtain the forward and derivative of forward rate on a very fine grid.
    // We should now define the rate on the real temporal grid.
    initial_forward = (double *)malloc((Nt + 1) * sizeof(double));
    initial_derive_forward = (double *)malloc((Nt + 1) * sizeof(double));
    tgrid = (double *)malloc((Nt + 2) * sizeof(double));
    for (i = 0; i <= Nt; i++)
    {
initial_forward[i] = 0.;
initial_derive_forward[i] = 0.;
    }
    for (i = 0; i <= Nt+1; i++)
    {
tgrid[i] = i * maturity/(double)Nt;
    }
    extract_forward(forward, derive_forward, t_interpol, NinterpolZCB, tgrid, Nt

    // Desallocation of temporary arrays.
    free(derive_forward);
    free(forward);
    free(t_interpol);
```

```
    free(initial_yield);



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

    *ptprice=compute_fair_price(alpha_old, alpha_new,

   alpha_m, Gr, kappa_tabular,

   A0, agrid, Na,
   W0, wgrid, Nw,
   R0, sigma_r, alpha_r, flat_flag, R0_flat, rgrid, Nr,
   V0, rho_wr,
   tgrid, number_of_periods, n_step_by_period, Nt,
   gamma_vect, number_of_control_values,

   theta, scheme);


    free(agrid);
    free(wgrid);
    free(rgrid);
    free(tgrid);
    free(initial_derive_forward);
    free(initial_forward);
    free(gamma_vect);
    free(Kt);
    free(kappa_tabular);

  return OK;
}



int CALC(FD_GMWB_BSHW)(void *Opt, void *Mod, PricingMethod *Met)
{
  TYPEOPT *ptOpt = (TYPEOPT *)Opt;
  TYPEMOD *ptMod = (TYPEMOD *)Mod;
```

```
  return FD_gmwb_bshw(ptMod->S0.Val.V_PDOUBLE, ptOpt->Maturity.Val.V_DATE - ptMo
 ptMod->kr.Val.V_PDOUBLE,
 ptMod->Sigmar.Val.V_PDOUBLE,
 ptMod->RhoSr.Val.V_PDOUBLE,ptOpt->Alpha_m.Val.V_PDOUBLE, ptOpt->MaximumWithdraw
}

static int CHK_OPT(FD_GMWB_BSHW)(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_bshw";
      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_BSHW) =
{
  "FD_GMWB_BSHW",
  { {"Timestep_for_year", INT2, {100}, ALLOW}, {"SpaceStepNumber W", INT2, {100}
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(FD_GMWB_BSHW),
  { {"Fair fee", DOUBLE, {100}, FORBID},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
```

```
  CHK_OPT(FD_GMWB_BSHW),
  CHK_split,
  MET(Init)
};
```