Help

```
#include  "
href../../../mod/merhes1d_default/merhes1d_default_stdr/merhes1d_default_stdr_h_
#include "pnl/pnl_mathtools.h"
#include "pnl/pnl_tridiag_matrix.h"
#include "pnl/pnl_band_matrix.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2020+2) //The "#els
static int CHK_OPT(FD_FiniteElement_HestonCVA)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(FD_FiniteElement_HestonCVA)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else

static double interpolation_quadrangle(double *xy, PnlVect *V, double* mesh,
                                       int nnod, int nodx, int nody, double deltax, dou
{


    int nx, ny, nodinfleft, nodinfright, nodsupleft, nodsupright; //nodinfleft n

    //double xn, yn, zn;
    double x1,x2,x3,y1,y2,y3,z1,z2,z3,x4,y4,z4;
    //double z5;
    double Vinfleft,Vinfright,Vsupleft,Vsupright;
    double t,s;

    nx = xy[0]/deltax;
    ny = xy[1]/deltay;

    nodinfleft = (ny*nodx + (nx+1))-1; //-1 because we start in zero
    nodinfright = (nodinfleft+1);
    nodsupleft  = (nodinfleft + nodx);
    nodsupright = (nodsupleft + 1);

    // Modification Ludo BUG HERE, could leave the domain
```

```
// Border cases...
if (nx+1==nodx) // translate left the quadrangle
{
    nodinfleft = nodinfleft-1;
    nodinfright = nodinfright-1;
    nodsupleft = nodsupleft-1;
    nodsupright = nodsupright-1;
}
if (ny+1==nody) // translate down the quadrangle
{
    nodinfleft = nodinfleft-nodx;
    nodinfright = nodinfright-nodx;
    nodsupleft = nodsupleft-nodx;
    nodsupright = nodsupright-nodx;

}
if ((nx+1==nodx)&&(ny+1==nody)) // translate left and down the quadrangle
{
    nodinfleft = nodinfleft-nodx-1;
    nodinfright = nodinfright-nodx-1;
    nodsupleft = nodsupleft-nodx-1;
    nodsupright = nodsupright-nodx-1;
}


Vinfleft = pnl_vect_get(V,nodinfleft);
Vinfright = pnl_vect_get(V,nodinfright);
Vsupleft = pnl_vect_get(V,nodsupleft);
Vsupright = pnl_vect_get(V,nodsupright);

x1 = mesh[nodinfleft];
y1 = mesh[nodinfleft + nnod];
z1 = Vinfleft;
x2 = mesh[nodinfright];
y2 = mesh[nodinfright + nnod];
z2 = Vinfright;
x3 = mesh[nodsupleft];
y3 = mesh[nodsupleft + nnod];
z3 = Vsupleft;
x4 = mesh[nodsupright];
y4 = mesh[nodsupright + nnod];
```

```
    z4 = Vsupright;

    t=(xy[0]-x1)/(x4-x1);
    s=(xy[1]-y1)/(y4-y1);
    return (z1 * (1.-t) + z2 * t) * (1.-s) + (z3 * (1.-t) + z4 * t) * s;

}


//-------------------------------------------------------------------------
//Classical functions, which could be optimized...
//-------------------------------------------------------------------------

static void normalCDF(double *N,  double value){
    /*
     void normalCDF(double *N,  double value)
     function which computes standar normal cumulative distribution
     input parameter:
     N: standar normal cumulative distribution
     value:value where the standar normal cumulative distribution is evaluated
     */
    float MSQRT12 = sqrt(0.5);

    (*N)= 0.5 * erfc(-value * MSQRT12);
}

//-------------------------------------------------------------------------

static double exact_call(double tau, double S, double strike, double sigma, doub
    /*
     void exact_call(double *BS_sol, double tau, double S, double strike, double
     function which computes the analytical solution for a European call option
     input parameter:
     BS_sol: analytical Black--Scholes solution
     tau:instant of time
     S: asset price
     strike: strike of the option
     sigma, D0, r: parameter of the problems
     ssup: maximum value of S in the original domain
     */
```

```
    double d1, d2;
    double Nd1;
    double Nd2;

    //points where the standard normal cumulative distribution is evaluated
    d1 = (log(S / strike) + (r - d + 0.5 * sigma*sigma) * tau)/ (sigma * sqrt(ta
    d2 = d1 - sigma * sqrt(tau);

    //standard normal cumulative distribution
    normalCDF(&Nd1,d1);
    normalCDF(&Nd2,d2);


    return S * exp( -d * (tau)) * Nd1 - strike * exp ( -r * tau) * Nd2;

}

//------------------------------------------------------------------------------

static double exact_put(double tau, double S, double strike, double sigma, doubl
    /*
     void exact_put(double *BS_sol, double tau, double S, double strike, double
     function which computes the analytical solution for a European put option
     input parameter:
     BS_sol: analytical Black--Scholes solution
     tau:instant of time
     S: asset price
     strike: strike of the option
     sigma, D0, r: parameter of the problems
     ssup: maximum value of S in the original domain
     */

    double d1, d2;
    double Nd1;
    double Nd2;

    //points where the standar normal cumulative distribution is evaluated
    d1 = (log(S / strike) + (r - d + 0.5 * sigma*sigma) * tau)/ (sigma * sqrt(ta
    d2 = d1 - sigma * sqrt(tau);

    //standard normal cumulative distribution
```

```
    normalCDF(&Nd1,-d1);
    normalCDF(&Nd2,-d2);

    return strike * exp ( -r * tau) * Nd2 - S * exp( -d * (tau)) * Nd1;

}

//-----------------------------------------------------------------------------

static void exact_call_1d(PnlVect *V, double t, double *meshS, double strike, do
    /*
     void exact_call_1d(double *V, double t, double *meshS, double strike, doubl
     function which computes the analytical solution for a European call option
     input parameter:
     V: analytical Black--Scholes solution
     t:instant of time
     meshS: mesh of asset price
     strike: strike of the option
     Tmaturity: maturity time
     sigma, gammas, qs, r: parameter of the problems
     nnod: number of nodes
     */

    double D0 = dividend + rfree - rate;
    double d1, d2;
    double Nd1;
    double Nd2;
    int i;
    for (i = 0;i<nnod;i++){
        //points where the standar normal cumulative distribution is evaluated
        d1 = (log(meshS[i] / strike) + (rfree - D0 + 0.5 * sigma*sigma) * (Tmatu
        d2 = d1 - sigma * sqrt((Tmaturity-t));

        //standar normal cumulative distributio
        normalCDF(&Nd1,d1);
        normalCDF(&Nd2,d2);
        pnl_vect_set(V,i, meshS[i] * exp( -D0 * (Tmaturity-t)) * Nd1 - strike *
    }

}
```

```
//-----------------------------------------------------------------------

static void exact_put_1d(PnlVect *V, double t, double *meshS, double strike, dou
    /*
     void exact_put_1d(double *V, double t, double *meshS, double strike, double
     function which computes the analytical solution for a European put option f
     input parameter:
     V: analytical Black--Scholes solution
     t:instant of time
     meshS: mesh of asset price
     strike: strike of the option
     Tmaturity: maturity time
     sigma, gammas, qs, r: parameter of the problems
     nnod: number of nodes
     */

    double D0 = dividend + rfree - rate;
    double d1, d2;
    double Nd1;
    double Nd2;
    int i;
    for (i = 0;i<nnod;i++){
        //points where the standar normal cumulative distribution is evaluated
        d1 = (log(meshS[i] / strike) + (rfree - D0 + 0.5 * sigma*sigma) * (Tmatu
        d2 = d1 - sigma * sqrt((Tmaturity-t));

        //standar normal cumulative distributio
        normalCDF(&Nd1,-d1);
        normalCDF(&Nd2,-d2);
        pnl_vect_set(V,i, strike * exp( -rfree * (Tmaturity-t)) * Nd2 - meshS[i]
    }
}



//-----------------------------------------------------------------------


static void characteristic2d( double *chi, double *xy, double deltatau, double m
    /*
     void characteristic2d( double *chi, double *xy, double deltatau, double mus
```

```
     function which computes the value of a node through the characteristic curv
     input parameter:
     chi: characteristic value
     xy: node to be evaluated
     deltatau: time step
     musup: maximum value of the volatility in the original domain
     rho, gamma, rR, kappa, theta: parameter of the problem
     smin, smax: minimum and maximum value of S in the reescalated domain
     mumin, mumax: minimum and maximum value of the volatility in the reescalate
     */
    chi[0] = xy[0] * exp(-(xy[1] * musup + (rho * gamma) / 2 - rR) * deltatau);

    if (chi[0]>smax){
        chi[0] = smax;

        chi[1] =  1 / (0.5 * rho * gamma + kappa) * (0.5 * gamma*gamma / musup -
        if (chi[1]<mumin){
            chi[1] = mumin;
        }
        else{
            if(chi[1]>mumax){
                chi[1] = mumax;
            }
        }
    }
    else{
        chi[1] = - 1 / (0.5 * rho * gamma + kappa) * (0.5 * gamma*gamma / musup
        if(chi[1]<mumin){
            chi[1] = mumin;

            chi[0] = xy[0] * pow(((mumin + 1/(gamma*rho*0.5 + kappa)*(0.5*gamma*
        }
    }

}

//-------------------------------------------------------------------------

static void min_0_esc(double *finalvalue, double V){
    /*
     void min_0_esc(double *finalvalue, double V)
```

```
     function which computes the negative part function of an scalar, minimum be
     input parameter:
     finalvalue: minimum value
     V: value to be compared with zero
     */

    if(V<0){
        (*finalvalue) = V;
    }
    else{
        (*finalvalue) = 0;
    }

}


//-------------------------------------------------------------------------

static void LU_crout_LUDO(PnlVect* x, PnlMat* a, PnlVect* b, int n)
{
    pnl_mat_syslin (x, a, b);
}



//-------------------------------------------------------------------------


static void max_0(PnlVect *finalvalue, PnlVect *V, int length){
    /*
     void max_0(double *finalvalue, double *V, int length)
     function which computes the positive part function of a vector, maximum bet
     input parameter:
     finalvalue: maximum value
     V: vector whose elements are compared with zero
     length: length of the vector V
     */
    int i;
    for( i = 0; i<length; i++){
        pnl_vect_set(finalvalue, i, MAX(pnl_vect_get(V,i),0.));
    }
```

```
}

//-------------------------------------------------------------------------

static void max_0_esc(double *finalvalue, double V){
    /*
     void max_0_esc(double *finalvalue, double V)
     function which computes the positive part function of an scalar, maximum be
     input parameter:
     finalvalue: maximum value
     V: value to be compared with zero
     */

    if(V>0){
        (*finalvalue) = V;
    }
    else{
        (*finalvalue) = 0;
    }

}

//-------------------------------------------------------------------------
// Functions related to matrix, vector, finite elements, etc.
//-------------------------------------------------------------------------

static void assembled_mat(PnlMat *Ah, double *aelt, int *vertix, int nnod){
    /*
     void assembled_mat(double *Ah, double *aelt, int *vertix, int nnod)
     function which enssambles the submatrix in the global matrix of the system
     input parameter:
     Ah: global matrix
     aelt: submatrix
     vertix: nodes associated with the finite element where aelt has been comput
     nnod: number of nods in the mesh
     */

    int i, j, ii, jj;

    for (i  = 0;i<3;i++){
```

```
        ii = vertix[i];
        for (j = 0;j<3;j++){
            jj = vertix[j];
            pnl_mat_set(Ah,ii,jj, pnl_mat_get(Ah,ii,jj) + aelt[j+i*3]);
        }

    }

}

//-----------------------------------------------------------------------------

static void mesh2d(double *nodes, int xmax, int ymax, int xmin, int ymin, int no
    /*
     void mesh2d(double *nodes, int xmax, int ymax, int xmin, int ymin, int nodx
     function which builds a uniform two dimensional mesh
     input parameter:
     nodes: matrix where the mesh is built. First row with x coordinate, second
     xmax, xmin: maximum and minimum value of the x domain
     ymax, ymin: maximu and minimum value of the y domain
     nodx, nody: numer of nodes in the x and y direction respectively
     */

    double deltax = (double) (xmax - xmin) / (nodx - 1);
    double deltay = (double)(ymax - ymin) / (nody - 1);
    int nnod = nodx * nody;
    double* vectory;
    vectory = (double*) malloc(sizeof(double)*nody);
    double xx;
    int j,i;
    int k = 0;
    double yy = ymin - deltay;




    for (j = 0; j < nody; j++) {
        yy = yy + deltay;
        vectory[j] = yy;
        xx = xmin - deltax;
        for (i = 0; i < nodx; i++) {
```

```
            xx = xx + deltax;
            nodes[k + 0*nnod] = xx;
            nodes[k + 1*nnod] = yy;
            k = k+1;
        }
    }
    free(vectory);

}



//-------------------------------------------------------------------------


static void transformation(double *xy, double x, double y, double a1, double a2,
    /*
     void transformation(double *xy, double x, double y, double a1, double a2, d
     function which transforms a node from the reference element into a node of
     input parameter:
     xy: transformed values
     x, y: coordinates of the reference triangles
     a1,a2;coordinate x and y respectively for the first node of the element
     b1,b2;coordinate x and y respectively for the second node of the element
     c1,c2;coordinate x and y respectively for the third node of the element
     */

    /*
    double mat1[2*2];

    mat1[0] = b1 - a1;
    mat1[1] = c1 - a1;
    mat1[2] = b2 - a2;
    mat1[3] = c2 - a2;



    double vec1[2];

    vec1[0] = x;
```

```
    vec1[1] = y;


    double vec2[2];

    vec2[0] = a1;
    vec2[1] = a2;

    double Prod[2];
    int i,j,k;
    for (i = 0;i<2; i++){
        Prod[i] = 0.;
    }


    for (i = 0; i<2;i++){//row
        for (j = 0; j<1;j++){//columns
            for (k = 0;k<2;k++){
                Prod[j + i*1] = Prod[j + i*1] + mat1[k + i*2]*vec1[j + k*1];
            }

            xy[j + i*1] = Prod[j + i*1] + vec2[j + i*1];

        }

    }
    */

    xy[0]=(b1-a1) * x + (c1-a1) * y + a1;
    xy[1]=(b2-a2) * x + (c2-a2) * y + a2;



}


//-------------------------------------------------------------------------


static void matrixA(double *diffusion, double musup, double rho, double gamma, d
    /*
```

```
     void matrixA(double *diffusion, double musup, double rho, double gamma, dou
     function which builds the diffusion matrix of the equation
     input parameter:
     diffusion: where the matrix iis defined
     musup: maximum value of volatility in the original domain
     rho, gamma:parameter data
     x,y: coordinate of the node where the matrix is evaluated
     */

    diffusion[0] = 0.5 * y * x * x * musup;
    diffusion[1] = 0.5 * rho * y * x * gamma;
    diffusion[2] = 0.5 * rho * y * x * gamma;
    diffusion[3] = 0.5 * gamma*gamma * y / musup; // BUG Here, should be *musup

}



//-------------------------------------------------------------------------

static void mat02(double *aelt, double a1, double a2, double b1, double b2, doub
    /*
     void mat02(double *aelt, double a1, double a2, double b1, double b2, double
     function which builds a submatrix of Ah associated with the divergencial te
     input parameter:
     aelt: submatrix to build
     a1,a2;coordinate x and y respectively for the first node of the element
     b1,b2;coordinate x and y respectively for the second node of the element
     c1,c2;coordinate x and y respectively for the third node of the element
     nodesg, weight: nodes and weigth for the Gauss quadrature formulae
     msup: maximum value of the volatility in the original domain
     gamma, rho: parameter of the model
     Dphi:transformation matrix from the reference element
     deter: determinant of Dphi
     */

    double Dphiinv[2*2];
    double Dphiinvtras[2*2];
    int i,j,k,l;
    int DPhat[2*3];
    int DPhattras[3*2];
```

```
double diffusion[2*2];
double Prod1[3*2];
double Prod2[2*3];
double Prod3[3*2];
double Prod4[3*3];
double xy[2];



Dphiinv[0] = Dphi[0]/deter;
Dphiinv[1] = - Dphi[1]/deter;
Dphiinv[2] = - Dphi[2]/deter;
Dphiinv[3] = Dphi[3]/deter;


for (i = 0;i<2;i++){
    for (j=0;j<2;j++){
        Dphiinvtras[j + i*2] = Dphiinv[i + j*2];
    }
}



DPhat[0] = -1;
DPhat[1] = 1;
DPhat[2] = 0;
DPhat[3] = -1;
DPhat[4] = 0;
DPhat[5] = 1;

for (i = 0;i<3;i++){
    for (j=0;j<2;j++){
        DPhattras[j + i*2] = DPhat[i + j*3];
    }
}

for (i=0; i<3*3; i++) {
    aelt[i] = 0;
}
```

```
//product of matrix


for (i = 0; i<3;i++){//row
    for (j = 0; j<2;j++){//columns
        Prod1[j + i*2] =0;
    }
}


for (i = 0; i<3;i++){//row
    for (j = 0; j<2;j++){//columns
        for (k = 0;k<2;k++){
            Prod1[j + i*2] =  Prod1[j + i*2] +DPhattras[k + i*2]*Dphiinvtras
        }
    }
}

for (i = 0; i<2;i++){//row
    for (j = 0; j<3;j++){//columns
        Prod2[j + i*3] =0;
    }
}

for (i = 0; i<2;i++){//row
    for (j = 0; j<3;j++){//columns
        for (k = 0;k<2;k++){
            Prod2[j + i*3] = Prod2[j + i*3] + Dphiinv[k + i*2]*DPhat[j + k*3
        }
    }
}



for (l = 0; l<3;l++){ //loop through the integration nodes

    transformation(xy, nodesg[l], nodesg[l+3], a1, a2, b1, b2, c1, c2);


    matrixA(diffusion, musup, rho, gamma, xy[0], xy[1]);
```

```
        for (i = 0; i<3;i++){//row
            for (j = 0; j<2;j++){//columns
                Prod3[j + i*2] =0;
            }
        }
        for (i = 0; i<3;i++){//row
            for (j = 0; j<2;j++){//columns
                for (k = 0;k<2;k++){
                    Prod3[j + i*2] = Prod3[j + i*2] + Prod1[k + i*2]*diffusion[j
                }
            }
        }


        for (i = 0; i<3;i++){//row
            for (j = 0; j<3;j++){//columns
                Prod4[j + i*3] =0;
            }
        }
        for (i = 0; i<3;i++){//row
            for (j = 0; j<3;j++){//columns
                for (k = 0;k<2;k++){
                    Prod4[j + i*3] = Prod4[j + i*3] + Prod3[k + i*2]*Prod2[j + k
                }
            }
        }


        for (j = 0;j<3*3; j++){
            aelt[j] +=  weightg[k]*Prod4[j];
        }

    }
    for (j = 0;j<3*3; j++){
        aelt[j] =  deter*aelt[j];

    }

}
```

```
//-------------------------------------------------------------------------

static void polynomial(double *P, double x, double y){
    /*
     void polynomial(double *P, double x, double y)
     function which builds the base function
     input parameter:
     P: vector where the base function is defined
     x,y: coordinates where the base function is evaluated
     */

    P[0] = 1 - x - y;
    P[1] = x;
    P[2] = y;

}



//-------------------------------------------------------------------------

static void mat01(double *aelt, double *nodesg, double *weightg){
    /*
     void mat01(double *aelt, double *nodesg, double *weightg)
     function which builds a submatrix of Ah associated with the time derivative
     input parameter:
     aelt: submatrix to build
     nodesg, weight: nodes and weigth for the Gauss quadrature formulae
     */

    int i,j,l;
    double *P=(double*) malloc(3 * sizeof(double));
    double *PP=(double*) malloc(3 * 3 * sizeof(double));


    for (i=0; i<3*3; i++) {
        aelt[i] = 0.0;
    }
```

```
    for (i = 0; i<3; i++){

        polynomial(P,nodesg[i], nodesg[i + 3]);

        for (j=0; j<3;j++){
            for (l=0; l<3;l++){
                PP[l +j*3] = P[j]*P[l];

            }
        }
        for (j = 0;j<3*3;j++){
            aelt[j] = aelt[j] + weightg[i] * PP[j];
        }

    }

    free(P);
    free(PP);

}

//-------------------------------------------------------------------------

static void connectivity(int *connect, int nx, int ny, int nelt){
    /*
     void connectivity(int *connect, int nx, int ny, int nelt)
     function which builds a matrix containing the conectivity of the finite ele
     row i of the matrix contain the nodes of the finite element i
     input parameter:
     connect: matrix to build
     nx, ny: nodes in x and y direction respectively
     nelt: number of elements in the mesh
     */

    int k = 0;
    int j, i, n1, n2, n3, n4;

    for (j = 0; j<ny-1; j++){
        n1 = j*nx;
        for (i = 0; i<nx-1; i++){
```

```
            n2 = n1 + 1;
            n3 = n2 + nx;
            n4 = n1 + nx;

            connect[0+k*3] = n4;
            connect[1+k*3] = n1;
            connect[2+k*3] = n3;

            k = k+1;

            connect[0+k*3] = n2;
            connect[1+k*3] = n3;
            connect[2+k*3] = n1;

            k = k+1;
            n1 = n1+1;
        }
    }


}



//-------------------------------------------------------------------------


static void localiza(int *nodes_contain, double *smint, int nodx, int nody, doub
    /*
     void localiza(int *nodes_contain, double *smint, int nodx, int nody, double
     function which finds the element where a point is localized
     input parameter:
     nodes_contain: nodes of the element
     smint: value to be localize in an element
     nodx, nody: number of nodes in x and y direction respectively
     deltax, deltay: spatial steps
     */

    int nx, ny, nodinfleft, nodinfright, nodsupleft, nodsupright; //nodinfleft n

    nx = smint[0]/deltax;
```

```
    ny = smint[1]/deltay;

    nodinfleft = (ny*nodx + (nx+1))-1; //-1 because we start in zero
    nodinfright = (nodinfleft+1);
    nodsupleft  = (nodinfleft + nodx);
    nodsupright = (nodsupleft + 1);

    if ((smint[0]-nx*deltax)>(smint[1] - ny*deltay)){

        nodes_contain[0] = nodinfright;
        nodes_contain[1] = nodsupright;
        nodes_contain[2] = nodinfleft;
    }
    else{
        nodes_contain[0] = nodsupleft;
        nodes_contain[1] = nodinfleft;
        nodes_contain[2] = nodsupright;
    }


    // (BUG HERE) Modification L.Goudenege : triangle could leave the domain
    // Consider the border cases...
    if (nx+1==nodx) // translate left the down-right triangle
    {
        nodes_contain[0] = nodinfright-1;
        nodes_contain[1] = nodsupright-1;
        nodes_contain[2] = nodinfleft-1;
    }
    if (ny+1==nody) // translate down the up-left triangle
    {
        nodes_contain[0] = nodsupleft-nodx;
        nodes_contain[1] = nodinfleft-nodx;
        nodes_contain[2] = nodsupright-nodx;
    }
    if ((nx+1==nodx)&&(ny+1==nody)) // translate left and down the up-left trian
    {
        nodes_contain[0] = nodsupleft-nodx-1;
        nodes_contain[1] = nodinfleft-nodx-1;
        nodes_contain[2] = nodsupright-nodx-1;
    }
}
```

```
//-------------------------------------------------------------------------

static double interpolation_interval(int jb, double a, double b, PnlVect *uold,
    /*
     void interpolation_interval(double *Uinterpol, int jb, double a, double b,
     function which computes the interpolation value in a middle point of an int
     value in the extrems of the interval
     input parameter:
     Uinterpol: interpolation value
     jb: node of right extrem of the interval where shoul be interpolated
     a, b: coordinates of the extremes of the interval
     uold: xva in the previous instant of time
     Sint: value where the function should be interpolated
     */

    return (pnl_vect_get(uold,jb)-pnl_vect_get(uold,jb-1))*(Sint-a)/(b-a) + pnl_

}


//-------------------------------------------------------------------------


static double interpolation_triangle(double *Velem, double *meshnodes, double *s
    /*
     void interpolation_triangle(double *Vinterp, double *Velem, double *meshnod
     function which obtains an interpolationvalue in a triangle, knowing the fun
     input parameter:
     Vinterp:  the interpolation value
     Velem: vector with the functio value at each node of the triangles
     meshnodes: nodes of the triangles
     smint: coordinate where the interpolated value is computed
     */

    /*
     double dv1, dv2, dv3, w1, w2, w3; //distances from each vertix of the trian
```

```
        dv1 = sqrt((meshnodes[0]-smint[0])*(meshnodes[0]-smint[0]) + (meshnodes[0+3
        dv2 = sqrt((meshnodes[1]-smint[0])*(meshnodes[1]-smint[0]) + (meshnodes[1+3
        dv3 = sqrt((meshnodes[2]-smint[0])*(meshnodes[2]-smint[0]) + (meshnodes[2+3
        printf("dv = %f %f %f\ n",dv1,dv2,dv3);

        w1 = 1/dv1;
        w2 = 1/dv2;
        w3 = 1/dv3;
        printf("w = %f %f %f\ n",w1,w2,w3);

        return (w1 * Velem[0] + w2 * Velem[1] + w3 * Velem[2] ) /(w1 + w2 + w3);
        */

       double xn, yn, zn;
       double x1,x2,x3,y1,y2,y3,z1,z2,z3,x4,y4;
       x1 = meshnodes[0];
       y1 = meshnodes[3];
       z1 = Velem[0];
       x2 = meshnodes[0+1];
       y2 = meshnodes[3+1];
       z2 = Velem[0+1];
       x3 = meshnodes[0+2];
       y3 = meshnodes[3+2];
       z3 = Velem[0+2];

       x4 = smint[0];
       y4 = smint[1];

       xn=(y2-y1)*(z3-z1) - (y3-y1)*(z2-z1);
       yn=(x3-x1)*(z2-z1) - (x2-x1)*(z3-z1);
       zn=(x2-x1)*(y3-y1) - (x3-x1)*(y2-y1);

       return (x1*xn + y1*yn + z1*zn - x4*xn - y4*yn) / zn;



}

//-------------------------------------------------------------------------
```

```
static void vect01(double *belt, PnlVect *Vold,double *mesh,
            double a1, double a2, double b1, double b2, double c1, double c2,
            double *nodesg, double *weightg, double deter,
            double musup, double rho, double gamma,
            double rR, double kappa, double theta,
            double deltatau, int smin, int smax, int mumin, int mumax,
            int nnod, int nods, int nodmu, double deltas, double deltamu){
    /*
     void vect01(double *belt, double *Vold,double *mesh, double a1, double a2,
     function which computes the subvector associated with the characteristic te
     input parameter:
     belt: vector in each elemento
     Vold: derivative value in the previous instant of time
     mesh: mesh of the problem
     a1,a2;coordinate x and y respectively for the first node of the element
     b1,b2;coordinate x and y respectively for the second node of the element
     c1,c2;coordinate x and y respectively for the third node of the element
     nodesg, weightg: nodes and weight of the Gauss wuadrature formulae
     deter:determinant of DPhi
     musup: maximum value of the volatility in the original domain
     rho, gamma, rR, kappa, theta: parameter of the problem
     deltatau: time step
     smin, smax: minimum and maximum value of S in the reescalated domain
     mumin, mumax: minimum and maximum value of the volatility in the reescalate
     nods, nodmu: nodes in the S and volatily direction respectively
     deltas, deltamu: spatial steps
     */

    double smint[2];
    //double P=[3];
    double *P=(double*) malloc(3 * sizeof(double));
    double xy[2];
    int nodes_cont[3]; //node of the triangle which contain the smint point
    int i, j, numnod;


    double Vinterp;// interpolation value in the point smint
    double Velem[3];  // derivative value in the triangle of the mesh where smin
    double meshnodes[2*3]; //coordinate of the nodes_cont
```

```
    for (j = 0;j<3;j++){
        belt[j] = 0;
    }

    for (j = 0;j<3;j++){
        //transformation of the integration nodes into the original mesh
        transformation(xy, nodesg[j], nodesg[j+3], a1, a2, b1, b2, c1, c2);
        //obtaining of the value of the nodes through the characteristic curve
        characteristic2d(smint, xy, deltatau, musup, rho, gamma, rR, kappa, thet

        localiza(nodes_cont, smint, nods, nodmu, deltas, deltamu);

        for (i = 0; i <3; i++){
            numnod = nodes_cont[i];
            Velem[i] = pnl_vect_get(Vold,numnod);
            meshnodes[i] = mesh[numnod];
            meshnodes[i + 3] = mesh[numnod + nnod];
        }

        //interpolation to evaluate the funtion in smint
        Vinterp = interpolation_triangle(Velem, meshnodes, smint);
        // Better interpolation made by following call :
        Vinterp = interpolation_quadrangle(smint, Vold, mesh, nnod, nods, nodmu,

        polynomial(P, nodesg[j], nodesg[j+3]);

        belt[0] = belt[0] + weightg[j] * Vinterp * P[0];
        belt[1] = belt[1] + weightg[j] * Vinterp * P[1];
        belt[2] = belt[2] + weightg[j] * Vinterp * P[2];
    }

    for (j = 0;j<3;j++){
        belt[j] = deter*belt[j];
    }

    free(P);

}
```

```
//-------------------------------------------------------------------------


static void matbuilt(PnlMat *Ah_V, PnlMat *Ah_XVA, double *nodesg, double *weigh
    /*
     void matbuilt(double *Ah_V, double *Ah_XVA, double *nodesg, double *weightg
     function which builds the matrix of the final system for the Heston model a
     input parameter:
     Ah_V, Ah_XVA: matrix of the final linear system for the Heston and XVA mode
     nodesg, weigth: nodes and weight for the Gauss quadrature formulae
     rho, gamma, r, lambdaB, lambdaC: parameter of the model
     musup: maximum value of the volatility in the real domain
     connect: connectivity matrix
     nelt,nnod: number of finite elements and nodes of the mesh respectively
     mesh: mesh of the problem
     deltatau: step of time
     */

    double aelt01[3*3];

    double Dphi[2*2];
    int vertix[3];
    double deter;
    double aelt02[3*3];//matrix in each element
    double aeltV[3*3]; //matrix in each element
    double aeltXVA[3*3]; //matrix in each element
    int j, k, i, aa, bb, cc;
    double a1, a2, b1, b2, c1, c2;

    mat01(aelt01, nodesg, weightg);


    for (i = 0;i<nnod*nnod;i++){
    //            Ah_V[i] = 0.0;
    //        Ah_XVA[i] = 0.0;
    }


    //loop through the finite elements

    for(k = 0; k<nelt; k++){
```

```
    for (i = 0;i<3;i++){
        vertix[i] = connect[i + k*3];
    }


    //node associated with each vertix
    aa = vertix[0];
    bb = vertix[1];
    cc = vertix[2];

    //Coordinate of the vertix in the mesh2d

    a1 = mesh[aa];
    a2 = mesh[aa + nnod];
    b1 = mesh[bb];
    b2 = mesh[bb + nnod];
    c1 = mesh[cc];
    c2 = mesh[cc + nnod];


    //transformation from the reference element
    Dphi[0] = b1-a1;
    Dphi[1] = c1-a1;
    Dphi[2] = b2-a2;
    Dphi[3] = c2-a2;

    deter = Dphi[0]*Dphi[3]-Dphi[1]*Dphi[2];

    mat02(aelt02, a1, a2, b1, b2, c1, c2, nodesg, weightg, musup, gamma, rho

    for (j = 0;j<3*3; j++){
        aeltV[j] =  (1 + r * deltatau) * deter * aelt01[j] + deltatau * aelt
        aeltXVA[j] = (1 + (r + lambdaB + lambdaC) * deltatau) * deter * aelt
    }

    assembled_mat(Ah_V, aeltV, vertix, nnod);

    assembled_mat(Ah_XVA, aeltXVA, vertix, nnod);

}
```

```
}


//------------------------------------------------------------------------


static void payoff(PnlVect *finalvalue, double *meshS, double strike, double ssu
    /*
     void payoff(double *finalvalue, double *meshS, double strike, double ssup,
     function which computes the payoff of the option
     input parameter:
     finalvalue: where the payoff is computed
     meshS: vector with the x coordinate of the mesh
     strike: strike of the option
     ssup: maximum value of the asset price iin the original domain
     opt: kind of option, 0 = call or 1 =vput
     nnod: number of nodes in the mesh
     */
    int i;

    if (opt==0){
        for (i = 0;i<nnod;i++){
            pnl_vect_set(finalvalue,i, MAX(ssup*meshS[i] - strike,0.));
        }
    }
    else{
        for (i = 0;i<nnod;i++){
            pnl_vect_set(finalvalue,i, MAX(strike - ssup*meshS[i],0.));
        }
    }

}



//------------------------------------------------------------------------

static void assembled_vect(PnlVect *bh, double *belt, int *vertix){
    /*
     void assembled_vect(double *bh, double *belt, int *vertix)
```

```
     function which introduces the subvectors associated with each element in th
     input parameter:
     bh: global vector
     belt: subvector associated with an element
     vertix: nodes of the element associated with belt
     */

    int i, ii;

    for (i = 0;i<3;i++){
        ii = vertix[i];
        pnl_vect_set(bh,ii, pnl_vect_get(bh,ii) + belt[i]);
    }
}




//-----------------------------------------------------------------------------

static void vectbuilt(PnlVect *bh_V, int nnod, int nelt, double *mesh, int *conn
    /*
     void vectbuilt(double *bh_V, int nnod, int nelt, double *mesh, int *connect
     function which builds the vector of the right hand side of the llinear syst
     input parameter:
     bh_V:  vcetor fo the system
     nnod, nelt: number of nodes and elements of the mesh respectively
     mesh: mesh of the problem
     connect: connectivity matrix
     nodesg, weightg: nodes and weight of the Gauss wuadrature formulae
     deltatau: time step
     musup: maximum value of the volatility in the original domain
     rho, gamma, rR, kappa, theta: parameter of the problem
     smin, smax: minimum and maximum value of S in the reescalated domain
     mumin, mumax: minimum and maximum value of the volatility in the reescalate
     Vold: derivative value in the previous instant of time
     nods, nodmu: nodes in the S and volatily direction respectively
     deltas, deltamu: spatial steps
     */

    double Dphi[2*2];
```

```
double belt[3];
int vertix[3];

float deter, a1, a2, b1, b2, c1, c2;
int k,i, aa, bb, cc;

for(i = 0; i<nnod; i++){
    pnl_vect_set(bh_V,i, 0.);
}

for (k = 0;k<nelt; k++){

    for (i = 0;i<3;i++){
        vertix[i] = connect[i + k*3];
    }

    //node associated with each vertix
    aa = vertix[0];
    bb = vertix[1];
    cc = vertix[2];

    //Coordinate of the vertix in the mesh2d

    a1 = mesh[aa];
    a2 = mesh[aa + nnod];
    b1 = mesh[bb];
    b2 = mesh[bb + nnod];
    c1 = mesh[cc];
    c2 = mesh[cc + nnod];

    //transformation of the reference elementos
    Dphi[0] = b1-a1;
    Dphi[1] = c1-a1;
    Dphi[2] = b2-a2;
    Dphi[3] = c2-a2;


    deter = Dphi[0]*Dphi[3]-Dphi[1]*Dphi[2];

    vect01(belt, Vold, mesh, a1, a2, b1, b2, c1, c2,
           nodesg, weightg, deter,
```

```
                musup, rho, gamma,
                rR, kappa, theta,
                deltatau, smin, smax,
                mumin, mumax,
                nnod, nods, nodmu, deltas, deltamu);
        assembled_vect(bh_V, belt, vertix);

    }


}



//-------------------------------------------------------------------------

static double functionXVA(double Vinterp, double sf, double lambdaB, double lamb
    /*
     void functionXVA(double *FXVA, double Vinterp, double sf, double lambdaB, d
     function which evaluates the f function of the right hand side equation
     input parameter:
     FXVA: the evaluated function
     Vinterp: the derivated value interpolated in the point where the function s
     sf, lambdaB, lambdaC, RB, RC: parameters of the problem
     */

    return (sf + lambdaC * (1 - RC)) * MAX(Vinterp,0.) + lambdaB * (1 - RB) * MI

}


//-------------------------------------------------------------------------

static void vect02(double *belt, PnlVect* V, double *mesh, double a1, double a2,
    /*
     void vect02(double *belt, double *V, double *mesh, double a1, double a2, do
     function which computes the subvector associated with the f function in eac
     input parameter:
     belt: subvector associated with each element
     V: derivative value
     mesh: mesh of the problem
```

```
 a1,a2;coordinate x and y respectively for the first node of the element
 b1,b2;coordinate x and y respectively for the second node of the element
 c1,c2;coordinate x and y respectively for the third node of the element
 nodesg, weightg: nodes and weight of the Gauss wuadrature formulae
 deter:determinant of Dphi
 nods, nodmu: nodes in the S and volatily direction respectively
 nnod: number of nodes in the mesh
 sf, lambdaB, lambdaC, RB, RC: parameters of the problem
 deltas, deltamu: spatial steps
 */

double xy[2];;
int nodes_cont[3];
int i, j, numnod;
double FXVA;


double Velem[3];  // derivative value in the triangle of the mesh where smin
double meshnodes[2*3]; //coordinate of the nodes_cont
double Vinterp;// interpolation value  in the integration node

double *P=malloc(3 * sizeof(double));  // polynomial in the triangles nodeS
for(i = 0; i<3; i++){
    belt[i] = 0.;
}
for (j = 0;j<3;j++){

    //transformation of the integration nodes into the original mesh
    transformation(xy, nodesg[j], nodesg[j+3], a1, a2, b1, b2, c1, c2);

    localiza(nodes_cont, xy, nods, nodmu, deltas, deltamu);

    for (i = 0; i <3; i++){
        numnod = nodes_cont[i];
        Velem[i] = pnl_vect_get(V,numnod);
        meshnodes[i] = mesh[numnod];
        meshnodes[i + 3] = mesh[numnod + nnod];
    }
    //interpolation to evaluate the funtion in smint
    Vinterp = interpolation_triangle(Velem, meshnodes, xy);
    // Better interpolation made by the following call :
```

```
        Vinterp = interpolation_quadrangle(xy, V, mesh, nnod, nods, nodmu, delta

        polynomial(P, nodesg[j], nodesg[j+3]);

        FXVA = functionXVA(Vinterp, sf, lambdaB, lambdaC, RB, RC);

        belt[0] = belt[0] + weightg[j] * FXVA * P[0];
        belt[1] = belt[1] + weightg[j] * FXVA * P[1];
        belt[2] = belt[2] + weightg[j] * FXVA * P[2];

    }
    // (BUG HERE) Modification L.Goudenege : index j was i so that nothing was d
    for (j = 0;j<3;j++){
        belt[j] = deter*belt[j];
    }

    free(P);

}


//-------------------------------------------------------------------------

static void vectbuilt_XVA(PnlVect *bh_XVA, int nnod, int nelt, double *mesh, int
    /*
     void vectbuilt_XVA(double *bh_XVA, int nnod, int nelt, double *mesh, int *c
     function which computes the vector of the linear system
     input parameter:
     bh_XVA: global vector
     nnod, nelt: number of nodes and elements of the mesh
     mesh: mesh of the problem
     connect: conectivity matrix
     nodesg, weightg: nodes and weight of the Gauss wuadrature formulae
     deltatau: time step
     musup: maximum value of the volatility in the original domain
     sf,rho, gamma, rR, lambdaB, lambdaC, RB, RC, kappa, theta: parameters of th
     smin, smax: minimum and maximum value of S in the reescalated domain
     mumin, mumax: minimum and maximum value of the volatility in the reescalate
     XVAold: XVA value in the previous instant of time
     V: derivative value
     nods, nodmu: nodes in the S and volatily direction respectively
```

```
 deltas, deltamu: spatial steps
 */


double Dphi[2*2];
double belt01[3];  //characteristic member
double belt02[3];  //right side function
double beltXVA[3];  //global vector associadted with a finite element


int vertix[3];

float deter, a1, a2, b1, b2, c1, c2;
int k,i, aa, bb, cc;

for(i = 0; i<nnod; i++){
    pnl_vect_set(bh_XVA,i, 0.);
}


for (k = 0;k<nelt; k++){

    for (i = 0;i<3;i++){
        vertix[i] = connect[i + k*3];
    }

    //node associated with each vertix
    aa = vertix[0];
    bb = vertix[1];
    cc = vertix[2];

    //Coordinate of the vertix in the mesh2d
    a1 = mesh[aa];
    a2 = mesh[aa + nnod];
    b1 = mesh[bb];
    b2 = mesh[bb + nnod];
    c1 = mesh[cc];
    c2 = mesh[cc + nnod];

    //transformation of the reference elementos
    Dphi[0] = b1-a1;
```

```
        Dphi[1] = c1-a1;
        Dphi[2] = b2-a2;
        Dphi[3] = c2-a2;

        deter = Dphi[0]*Dphi[3]-Dphi[1]*Dphi[2];

        vect01(belt01, XVAold, mesh, a1, a2, b1, b2, c1, c2, nodesg, weightg, de

        vect02(belt02, V, mesh, a1, a2, b1, b2, c1, c2, nodesg, weightg, deter,


        for (i = 0;i<3;i++){
            beltXVA[i] = belt01[i] - deltatau * belt02[i];
        }

        assembled_vect(bh_XVA,beltXVA,vertix);

    }

}



//-------------------------------------------------------------------------

static void min_0(double *finalvalue, double *V, int length){
    /*
     void mix_0(double *finalvalue, double *V, int length)
     function which computes the negative part function of a vector, minimum bet
     input parameter:
     finalvalue: minimum value
     V: vector whose elements are compared with zero
     length: length of the vector V
     */

    int i;
    for( i = 0; i<length; i++){
        if(V[i]<0){
            finalvalue[i] = V[i];
        }
        else{
```

```
            finalvalue[i] = 0;
        }
    }


}


//-------------------------------------------------------------------------

static void conditionsV(PnlMat* A, PnlVect* b,
                double tau, int *boundary1, int *boundary2, int *boundary3,
                PnlVect* Vold, int nnod, int nods, int nodmu,
                double deltatau, double musup, double ssup, double *mesh,
                double rfree, double kappa, double theta, double rR, double str
                int smin, int smax, int mumin, int mumax,
                int opt, int feller, PnlVect *V_no_bc, double deltamu, double d
    /*
     void conditionsV(double *A, double *b, double tau, int *boundary1, int *bou
     function which imposes the condition of the problem
     input parameter:
     A: matrix of the system
     b: vector of the system
     tau: instant of time
     boundary1, boundary2, boundary3: boundary where a condition should be impos
     Vold: derivative value in the previous instant of time
     nnod: number of nodes in the mesh
     nods, nodmu: nodes in the S and mu direction respectively
     deltatau: time step
     musup, ssup: maximum value of mu and S respectiveley in the original domain
     r, kappa, theta, rR, strike: parameter of the problem
     smin, smax: minimum and maximum valur of S in the reescalated domain
     mumin, mumax: minimum and maximum value of mu in the reescalated domain
     opt: kind of option
     feller: indicator of the Feller conditions is satisfied
     V_no_bc: derivative value without imposing boundary condition
     deltamu, deltas: spatial steps
     */

    int i, j;
    double *meshS;
```

```
meshS = (double*)(malloc(sizeof(double)*nods));
double* meshS_0;
meshS_0 = (double*)(malloc(sizeof(double)*(nods-2)));
double coef1, coef2, coef3, V0;

//double BS_sol[1];
double BS_sol=0.;
double D0 = rfree-rR;
double x, y, Vm;

PnlVect* bb;
PnlVect* Vold_boundary;
PnlVect* boundary_value;
PnlVect* diag1;
PnlVect* diag2;
PnlMat* AA;

bb = pnl_vect_create(nods-2);
Vold_boundary = pnl_vect_create(nods-2);
boundary_value = pnl_vect_create(nods-2);
diag1 = pnl_vect_create(nods-2);
diag2 = pnl_vect_create(nods-2);
AA = pnl_mat_create(nods-2, nods-2);



for (i = 0; i<nods; i++){
    j = boundary3[i];
    meshS[i]=mesh[j];
    pnl_mat_set(A,j,j,1.e+12) ;

    if(opt==0){ //call option
        BS_sol = exact_call(tau, meshS[i]*ssup, strike, sqrt(musup), D0, rfr
        pnl_vect_set(b,j, 1.e+12 * BS_sol);
    }
    else{//put option
        BS_sol = exact_put(tau, meshS[i]*ssup, strike, sqrt(musup), D0, rfre
        pnl_vect_set(b,j, 1.e+12 * BS_sol);
    }

}
```

```
//boundary S = Smax
for (i = 0; i<nodmu; i++){
    j = boundary2[i];
    x = mesh[j]; //x = Ssup non... x=1.
    y = mesh[j + nnod];
    pnl_mat_set(A,j,j, 1.e+12);
    if(opt==0){ //call option
        BS_sol = exact_call(tau, x*ssup, strike, sqrt(y*musup), D0, rfree);
        pnl_vect_set(b,j, 1.e+12 * BS_sol);
    }
    else{//put option
        BS_sol = exact_put(tau, x*ssup, strike, sqrt(y*musup), D0, rfree);
        pnl_vect_set(b,j, 1.e+12 * BS_sol);
    }
    if(i ==0){
        Vm = BS_sol;
    }
}

// boundary mu = 0;

if (feller == 0){


    coef1 = 1 + deltatau * rfree + deltatau * kappa * theta / deltamu ;
    coef2 = -0.5 * rR * deltatau / deltas;
    coef3 = -deltatau * kappa * theta / deltamu;

    if(opt==0){
        V0 = 0.;
    }
    else{
        V0 = strike*exp(-rfree*tau);
    }

    for(i = 0;i<nods-2;i++){
        j = boundary1[i+1];
        meshS_0[i] = mesh[j];
        pnl_vect_set(bb,i , -coef3 * pnl_vect_get(V_no_bc,j));
        pnl_vect_set(Vold_boundary,i, pnl_vect_get(Vold,j));
```

```
}

for (i =0;i<nods-2;i++){
    pnl_vect_set(diag1,i, coef1);
    pnl_vect_set(diag2,i, coef2*meshS_0[i]);
}

for(i = 0;i<nods-2;i++){
    pnl_mat_set(A,i,i,pnl_vect_get(diag1,i));
}

// (BUG HERE) Modification L.Goudenege : index i-1 with i=0 for matrix.
// Maybe confusion between line and columns also
for(i = 0;i<nods-3;i++){
    //pnl_mat_set(AA,i+1,i,pnl_vect_get(diag2,i)); // confusion line/col
    pnl_mat_set(AA,i,i+1,pnl_vect_get(diag2,i));
}
for(i = 1;i<nods-2;i++){
    //pnl_mat_set(AA,i-1,i,-pnl_vect_get(diag2,i)); // confusion line/co
    pnl_mat_set(AA,i,i-1,-pnl_vect_get(diag2,i));
}

// (BUG HERE) Modification L.Goudenege : Index starts at 0 not 1 like Ma
//pnl_vect_set(bb,1, pnl_vect_get(bb,1) + pnl_vect_get(diag2,1)*V0);
//pnl_vect_set(bb,nods-2, pnl_vect_get(bb,nods-2)- pnl_vect_get(diag2,no
//for(i = 1;i<nods-2;i++){ pnl_vect_set(bb,i,pnl_vect_get(bb,i) + pnl_ve
pnl_vect_set(bb,0, pnl_vect_get(bb,0) + pnl_vect_get(diag2,0)*V0);
pnl_vect_set(bb,nods-3, pnl_vect_get(bb,nods-3) - pnl_vect_get(diag2,nod
for(i = 0;i<nods-2;i++){
    pnl_vect_set(bb,i,pnl_vect_get(bb,i) + pnl_vect_get(Vold_boundary,i)
}

pnl_mat_syslin(boundary_value, AA, bb);


// (BUG HERE) Modification L.Goudenege : index i starts at 0
for(i = 0;i<nods-2;i++){
    j = boundary1[i+1];
    pnl_vect_set(b,j,1.e+12 * pnl_vect_get(boundary_value,i));
    pnl_mat_set(A,j,j,1.e+12);
}
```

```
    }
    free(meshS);
    free(meshS_0);
    pnl_vect_free(&bb);
    pnl_vect_free(&Vold_boundary);
    pnl_vect_free(&boundary_value);
    pnl_vect_free(&diag1);
    pnl_vect_free(&diag2);
    pnl_mat_free(&AA);

}



//-----------------------------------------------------------------------------

static void matrixU(PnlMat *Ah, double *nodes, int nelt, double deltatau, double
    /*
     void matrixU(double *Ah, double *nodes, int nelt, double deltatau, double r
     function which builds the matrix of the linear system for the Black-Scholes
     input parameter:
     Ah: matrix of the system
     nodes: mesh of the problem
     nelt: number of element in the one dimensional mesh
     deltatau: time step
     r, sigma, lambdaB, lambdaC: parameter of the problems
     */
    int i,j,k;

    //weight and node of integration, Gauss formulae
    int wn = 3;
    double s, w0, w1;
    s = sqrt(0.6);
    w0 = 5.0 / 9.0;
    w1 = 8.0 / 9.0;
    double coef = 0.0;


    double gx[3];
    gx[0] = -s;
    gx[1] = 0.0;
```

```
gx[2] = s;
double gw[3];
gw[0] = w0;
gw[1] = w1;
gw[2] = w0;

double h,xm;
double aelt1[2*2];
double aelt2[2*2];
int aux1[2*2];
aux1[0] = 2;
aux1[1] = 1;
aux1[2] = 1;
aux1[3] = 2;
int aux2[2*2];
aux2[0] = 1;
aux2[1] = -1;
aux2[2] = -1;
aux2[3] = 1;

//int nnod = nelt+1;
pnl_mat_set_zero(Ah);


//    for (i = 0;i<nnod*nnod;i++){
//        Ah[i] = 0;
//    }




for (i = 0; i<nelt;i++){
    h = nodes[i+1]-nodes[i];
    xm = 0.5 * (nodes[i] + nodes[i+1]);

    for (j = 0; j<2;j++ ){
        for (k = 0; k<2;k++ ){
            aelt1[k + j*2] =  (1. + deltatau * (r + lambdaB + lambdaC)) * (h
        }
    }
```

```
        coef = 0.0;
        for (k = 0;k<wn;k++){
            coef = coef + gw[k] * pow(xm + 0.5 * h * gx[k],2);
        }

        coef = 0.5 * coef / h; //Gauss formulae
        for (j = 0; j<2;j++ ){
            for (k = 0; k<2;k++ ){
                aelt2[k + j*2] = deltatau * 0.5 * sigma*sigma * coef * aux2[k +
            }
        }


        //Assembled matrix1
        pnl_mat_set(Ah,i,i,pnl_mat_get(Ah,i,i) + aelt1 [0 + 0*2] + aelt2 [0 + 0*
        pnl_mat_set(Ah,i,i+1, pnl_mat_get(Ah,i,i+1) + aelt1 [1 + 0*2] + aelt2[1
        pnl_mat_set(Ah,i+1,i, pnl_mat_get(Ah,i+1,i)   + aelt1 [0 + 1*2] + aelt2
        pnl_mat_set(Ah,i+1,i+1, pnl_mat_get(Ah,i+1,i+1) + aelt1 [1 + 1*2] + aelt

    }
}




//-------------------------------------------------------------------------

static double characteristicU (double x, double deltatau, double sigma, double g
    /*
     void characteristicU (double *Sint, double x, double deltatau, double sigma
     funtion which evaluates the characteristic curve in a particula node
     input argument:
     Sint: characteristic value
     x: value to be evaluated
     deltatau: time step
     sigma, gammas, qs: parameters of the problem
     nelt: number of elements
     */

    return x*exp((qs - gammas - sigma*sigma)*deltatau);
```

```
}


//-------------------------------------------------------------------------

static void interval_located(int *jb, double x, double *meshS, int nnod){
    /*
     void interval_located(int *jb, double x, double *meshS, int nnod)
     funtion which finds the interval where a pint is located
     input parameter:
     jb: right extrem of the interval where the point is located
     x: point
     meshS: one dimensional mesh
     nnod: number of nodes in the mesh
     */

    int j;

    if(x>meshS[nnod-1]){
        (*jb) = nnod-1;
    }
    else if(x<meshS[0]){
        (*jb) = 1;
    }
    else{
        for (j = 0;j<nnod-1;j++){
            if(x>=meshS[j] && x<=meshS[j+1]){
                (*jb) = j+1;
                j = nnod;
            }
        }
    }


}



//-------------------------------------------------------------------------
```

```
static double functionUV(double V, double RB, double RC, double lambdaB, double
    /*
     void functionUV(double *value_f, double V, double RB, double RC, double lam
     function which evaluates the right hand side of the equation, (fucntion f)
     input parameter:
     value_f: fucntion value
     V: value where the function should be evaluated
     RB, RC, lambdaB, lambdaC, sF: parameter of the problems
     */

    return (1-RB) * lambdaB * MIN(V,0.) + ((1-RC) * lambdaC + sF) * MAX(V,0.);

}



//-------------------------------------------------------------------------

static void termindep(PnlVect *bh, double *meshS, int nelt, int *jb,double *Sint
    /*
     void termindep(double *bh, double *meshS, int nelt, int *jb,double *Sint, d
     function which computes the vector of the linear system for Black--Scholes
     input parameter:
     bh: vector of the system
     meshS: mesh of the problem
     nelt: number of elements
     jb: vector  with the node of right extreme of the interval where the charac
     Sint: characteristic value
     uold: xva in the previous instant of time
     V: derivative value
     deltatau: time step
     RB, RC, lambdaB, lambdaC, sF: parameter of the problem
     */

    int j;
    double a, b, h;
    double Uinterpol;

    double bhaux1[2];
    double bhaux2[2];
```

```
    double value_f;

    for(j =0;j<nelt+1;j++){
        pnl_vect_set(bh,j, 0);
    }

    for(j = 0;j<nelt;j++){ //loop throught the finite element

        a = meshS[jb[j]-1];
        b = meshS[jb[j]];
        h = b-a;



        Uinterpol = interpolation_interval(jb[j], a, b, uold,Sint[j]);

        bhaux1[0] = 0.5 * h * Uinterpol ;
        bhaux1[1] = 0.5 * h * Uinterpol ;

        value_f = functionUV(pnl_vect_get(V,j), RB, RC, lambdaB, lambdaC, sF);
        bhaux2[0] = 0.5 * h * value_f;
        value_f = functionUV(pnl_vect_get(V,j+1), RB, RC, lambdaB, lambdaC, sF);
        bhaux2[1] = 0.5 * h * value_f;

        //Assembled

        pnl_vect_set(bh,j,pnl_vect_get(bh,j) + bhaux1 [0] - deltatau * bhaux2[0]
        pnl_vect_set(bh,j+1,  pnl_vect_get(bh,j+1) + bhaux1 [1] - deltatau * bha

    }

}



//-------------------------------------------------------------------------

static void conditionsU(PnlMat *Ah, PnlVect *bh, PnlVect *uold, PnlVect *V, doub
    /*
     void conditionsU(double *Ah, double *bh, double *uold, double *V, double *U
     function which imposes the boundary condition in the Black-Scholes problem
```

```
 input parameter:
 Ah: matrix of the system
 bh: vector of the system
 uold: xva value in the previous instant of times
 V: derivative value
 U_0:XVA value in the previous fixed point iteration
 r: parameter of the problem
 deltatau: time step
 nnod: number of nodes of mesh
 meshS: 1 dimensional mesh
 RB, RC, lambdaB, lambdaC, sF, qs, gammas: parameters of the problem
 */

double a, b, Sint, g, f;
double Uinterpol;
double term1;
double term2;
double value_f;

// (BUG HERE) Modification L.Goudenege :
//    a = meshS[nnod-1];
//    b = meshS[nnod];
a = meshS[nnod-2];
b = meshS[nnod-1];

// (BUG HERE) Modification L.Goudenege :
//Sint = meshS[nnod]*exp((qs-gammas)*deltatau); //value through the characte
Sint = b*exp((qs-gammas)*deltatau); //value through the characteristic curve

// (BUG HERE) Modification L.Goudenege :
//Uinterpol = interpolation_interval(nnod, a, b, uold, Sint);
Uinterpol = interpolation_interval(nnod-1, a, b, uold, Sint);

min_0_esc(&term1, pnl_vect_get(V,0));
max_0_esc(&term2, pnl_vect_get(V,0));
//    f = (1 - RB) * lambdaB * MIN(V[0],0.) + ((1 - RC) * lambdaC + sF) * MA
f = (1 - RB) * lambdaB * term1 + (1 - RC) * lambdaC * term2 + sF * term2;

pnl_vect_set(bh,0, 1.e+8 * (pnl_vect_get(uold,0) - deltatau * f)/(1. + (r +

value_f = functionUV(pnl_vect_get(V,nnod-1), RB, RC, lambdaB, lambdaC, sF);
```

```
    g = (Uinterpol - deltatau * value_f)/(1. + (r + lambdaB + lambdaC) * deltata
    pnl_vect_set(bh,nnod-1,1.e+8 * g);

    pnl_mat_set(Ah,0,0, 1.e+8);
    pnl_mat_set(Ah,nnod-1,nnod-1, 1.e+8);
}



//-----------------------------------------------------------------------------

static void solutionU(PnlVect* U1d, PnlVect* uold, PnlVect* V, int it, int nt, d
    /*
     void solutionU(double *U1d, double *uold, double *V, int it, int nt, double
     function which builds the vector of the Black--Schole system and obtain the
     input parameter:
     U1d: XVA for the S mesh for the whole time discretization
     uold: xva value in the previous instant of time
     V: derivative value
     it: time iterator
    nt: number  of time step
    meshS: one dimensional mesh for the asset price
    nelt, nnod: number of element and nodes in the one dimensional mesh, respect
    jb: vector with the right extrem of the interval where the nodes through the
    Sint: characteristic value
    Ah: matrix of the system
    deltatau: time step
    r, RB, RC, lambdaB, lambdaC, sF, qs, gammas: parameter data
     */

    double tau;
    int i;
    tau =  deltatau*it;
    PnlVect *bh;
    PnlVect *U1d_S;

    bh=pnl_vect_create(nnod);

    termindep(bh,meshS,nelt,jb,Sint,uold,V,deltatau, RB, RC, lambdaB, lambdaC, s
```

```
    conditionsU(Ah,bh,uold,V,r,deltatau,nnod,meshS, RB, RC, lambdaB, lambdaC, sF

    U1d_S=pnl_vect_create(nnod);

    LU_crout_LUDO(U1d_S, Ah,bh, nnod);

    for (i = 0;i<nnod;i++){
        pnl_vect_set(U1d,i + it*nnod, pnl_vect_get(U1d_S,i));
        pnl_vect_set(uold,i, pnl_vect_get(U1d_S,i));
    }

    pnl_vect_free(&bh);
    pnl_vect_free(&U1d_S);

}


//-----------------------------------------------------------------------------

static void solver_BS_risk(PnlVect *U1d,
                    int Tin, double Tmaturity, int nt,
                    int nnod, double *meshS,
                    double rfree, double sigma, double kappa, double rate, doubl
                    double sF, double lambdaB, double lambdaC, double RB, double
                    double Smin, double Smax, double strike, int opt){
    /*
     void solver_BS_risk(double *U1d, int Tin, double Tmaturity, int nt, int nno
    funtion which computes the XVA of the associated Black--Scholes model
    input parameter:
    U1d: XVA for the S mesh for the whole time discretization
    Tin: initial time
    Tmaturity: maturity time
    nt: number  of time step
    nnod: number of nodes in the one dimensional mesh
    meshS: one dimensional mesh for the asset price
    r, sigma, rate, dividend, sF, lambdaB, lambdaC, RB, RC: parameter data
    Smax, Smin: minimum and maximum value for the aaset price
    strike: strike of the options
    opt: kind of option call = 0; put = 1
     */
    double deltatau  = (Tmaturity - Tin) / nt; // step of time
```

```
    int i;

    int nelt = nnod - 1;

    double D0, rR;

    PnlVect* uold;
    PnlMat* Ah;

//    double Sint[nelt]; //vector which the S value through the characteristic c
//    int jb[nelt];//right limit of the interval where Sint is located
    double* Sint;
    Sint = (double*) malloc(sizeof(double)*nelt);
    int* jb;
    jb = (int*) malloc(sizeof(int)*nelt);



    int j;
    double charac;

    int jb_1;

    PnlVect* Vrisk;
    PnlVect* V;
    int it;
    double tau,t;



    D0  = dividend + rfree - rate;
    rR = rate - dividend;


    for (i = 0;i<nnod*nt;i++){
        pnl_vect_set(U1d,i, 0.);
    }

    // (BUG HERE) Modification L.Goudenege : final condition for the XVA not USE
    //double Ufin[nnod];
```

```
//for(i = 0;i<nnod;i++){
//    Ufin[i] = 0.;
//}

// Solution in the previous instant of time
uold = pnl_vect_create(nnod);

for(i = 0;i<nnod;i++){
    pnl_vect_set(uold,i, 0.);
}


Ah = pnl_mat_create(nnod,nnod);

//System matrix
matrixU(Ah, meshS, nelt, deltatau, rfree, sigma,lambdaB, lambdaC);

for(j = 0;j<nelt;j++){
    charac = characteristicU (0.5*(meshS[j]+ meshS[j+1]),deltatau, sigma, di
    Sint[j] = charac;
    interval_located(&jb_1, Sint[j], meshS,nnod);
    jb[j] = jb_1;
}

Vrisk = pnl_vect_create(nnod*nt);
V = pnl_vect_create(nnod);


for (it = 0; it<nt; it++){
    tau = (it+1)*deltatau; //forward time
    t = Tmaturity - tau; //backward time

    //Analytical Black Scholes solution

    if(opt == 0){
        exact_call_1d(V, t, meshS, strike, Tmaturity, sigma, dividend, rate,
    }
    else{
        exact_put_1d(V, t, meshS, strike, Tmaturity, sigma, dividend, rate,
    }
```

```
        solutionU(U1d, uold, V, it, nt, meshS, nelt, nnod, jb, Sint, Ah, deltata

        for (i = 0;i<nnod;i++){
            pnl_vect_set(Vrisk,i + it*nnod, pnl_vect_get(U1d,i + it*nnod) + pnl_
        }
    }
    free(jb);
    free(Sint);
    pnl_vect_free(&uold);
    pnl_mat_free(&Ah);
    pnl_vect_free(&Vrisk);
    pnl_vect_free(&V);


}
//-----------------------------------------------------------------------------

static void conditionsXVA(PnlMat *A, PnlVect *b,
                  double tau, int *boundary1, int *boundary2, int *boundary3,
                  PnlVect *V, PnlVect *XVAold, int nnod, int nods, int nodmu,
                  double deltatau, double musup, double *mesh,
                  double r, double rho, double kappa, double gamma, double thet
                  double sF, double strike, double lambdaB, double lambdaC, dou
                  int smin, int smax, int mumin, int mumax,
                  PnlVect *XVA_solution_t, int feller, PnlVect *XVA_no_bc, doub
    /*
     void conditionsXVA(double *A, double *b, double tau, int *boundary1, int *b
     function which imposes the condition of the XVA problem
     input parameter:
     A: matrix of the system
     b: vector of the system
     tau: instant of time
     boundary1, boundary2, boundary3: boundary where a condition should be impos
     V: derivative value
     XVAold: XVA value in the previous instant of time
     XVA_0: XVA value in the previous fixed point iteration
     nnod: number of nodes in the mesh
     nods, nodmu: nodes in the S and mu direction respectively
     deltatau: time step
     musup maximum value of mu respectiveley in the original domain
```

```
 mesh: mesh of the problem
 r, rho, kappa, gamma, theta, rR, sF, strike, lambdaB, lambdaC, RB, RC: para
 smin, smax: minimum and maximum valur of S in the reescalated domain
 mumin, mumax: minimum and maximum value of mu in the reescalated domain
 XVA_solution_t: XVA value at the current instant of time for the Black-Scho
 feller: indicator of the Feller conditions is satisfied
 XVA_no_bc: XVA value without imposing boundary condition
 deltamu, deltas: spatial steps
 */

//boundary S = Smax
double x, y, XVAm;
int i,j;
int XVA0;
double* meshS_0;
meshS_0 = (double*) malloc(sizeof(double)*(nods-2));

double coef1, coef2, coef3;
PnlVect* bb;
PnlVect* XVAold_boundary;
PnlVect* boundary_value;
PnlVect* value_f;
double fXVA;
PnlMat* AA;
PnlVect* diag1;
PnlVect* diag2;

bb = pnl_vect_create(nods-2);
XVAold_boundary = pnl_vect_create(nods-2);
boundary_value = pnl_vect_create(nods-2);
value_f = pnl_vect_create(nods-2);

diag1 = pnl_vect_create(nods-2);
diag2 = pnl_vect_create(nods-2);
AA = pnl_mat_create(nods-2,nods-2);


for (i = 0; i<nodmu; i++){
    j = boundary2[i];
    x = mesh[j]; //x = Ssup
    y = mesh[j + nnod];
```

```
    pnl_mat_set(A,j,j, 1.e+12);

    // (BUG HERE) Modification L.Goudenege :
    //pnl_vect_set(b,j, 1.e+12 * pnl_vect_get(XVA_solution_t,i + nods*nodmu)
    pnl_vect_set(b,j, 1.e+12 * pnl_vect_get(XVA_solution_t,i + (nods-1)*nodm

    if(i==0){
        // (BUG HERE) Modification L.Goudenege :
        //XVAm = pnl_vect_get(XVA_solution_t,i + nods*nodmu);
        XVAm = pnl_vect_get(XVA_solution_t,i + (nods-1)*nodmu);
    }
}

//boundary mu = mumax;

for (i = 0; i<nods; i++){
    j = boundary3[i];

    pnl_mat_set(A,j,j, 1.e+12);
    pnl_vect_set(b,j, 1.e+12 * pnl_vect_get(XVA_solution_t,nodmu-1 + i*nodmu
}

// boundary mu = 0;

if (feller == 0){
    XVA0 = 0;

    coef1 = 1 + deltatau * (r + lambdaB + lambdaC) + deltatau * kappa * thet
    coef2 = -0.5 * rR * deltatau / deltas;
    coef3 = -deltatau * kappa * theta / deltamu;


    for(i = 0;i<nods-2;i++){
        j = boundary1[i+1];
        meshS_0[i] = mesh[j];
        pnl_vect_set(bb,i, -coef3 * pnl_vect_get(XVA_no_bc,j));
        pnl_vect_set(XVAold_boundary,i, pnl_vect_get(XVAold,j));
        fXVA = functionXVA (pnl_vect_get(V,j), sF, lambdaB, RB, lambdaC, RC)
        pnl_vect_set(value_f,i, fXVA);
    }
```

```
    for (i =0;i<nods-2;i++){
        pnl_vect_set(diag1,i, coef1);
        pnl_vect_set(diag2,i, coef2*meshS_0[i]);
    }


    for(i = 0;i<nods-2;i++){
        pnl_mat_set(AA,i,i, pnl_vect_get(diag1,i));
    }

    // (BUG HERE) Modification L.Goudenege : index i-1 with i=0 for matrix.
    // Maybe confusion between line and columns also
    for(i = 0;i<nods-3;i++){
        //pnl_mat_set(AA,i+1,i,pnl_vect_get(diag2,i)); // confusion line/col
        pnl_mat_set(AA,i,i+1, pnl_vect_get(diag2,i));
    }
    for(i = 1;i<nods-2;i++){
        //pnl_mat_set(AA,i-1,i, -pnl_vect_get(diag2,i+1)); // confusion line
        pnl_mat_set(AA,i,i-1, -pnl_vect_get(diag2,i));
    }

    // (BUG HERE) Modification L.Goudenege : Index starts at 0 not 1 like Ma
    //pnl_vect_set(bb,1, pnl_vect_get(bb,1) + pnl_vect_get(diag2,1)*XVA0);
    //pnl_vect_set(bb,nods-2, pnl_vect_get(bb,nods-2) - pnl_vect_get(diag2,n
    //for(i = 1;i<nods-2;i++){ pnl_vect_set(bb,i, pnl_vect_get(bb,i) + pnl_v
    pnl_vect_set(bb,0, pnl_vect_get(bb,0) + pnl_vect_get(diag2,0)*XVA0);
    pnl_vect_set(bb,nods-3, pnl_vect_get(bb,nods-3) - pnl_vect_get(diag2,nod
    for(i = 0;i<nods-2;i++){
        pnl_vect_set(bb,i, pnl_vect_get(bb,i) + pnl_vect_get(XVAold_boundary
    }

    LU_crout_LUDO(boundary_value, AA, bb,nods-2);

    // (BUG HERE) Modification L.Goudenege : index i starts at 0 and finish
    for(i = 0;i<nods-2;i++){
        j = boundary1[i+1];
        pnl_vect_set(b,j, 1.e+12 * pnl_vect_get(boundary_value,i));
        pnl_mat_set(A,j,j, 1.e+12);
    }
}
```

```
    free(meshS_0);
    pnl_vect_free(&bb);
    pnl_vect_free(&XVAold_boundary);
    pnl_vect_free(&boundary_value);
    pnl_vect_free(&value_f);
    pnl_vect_free(&diag1);
    pnl_vect_free(&diag2);
    pnl_mat_free(&AA);

}

//-----------------------------------------------------------------------------
int FDFiniteElement_CVA(double S0, NumFunc_1  *p, double Tmaturity, double strik
{
  int opt;
  int i,j;
    // Time data

    double Tin = 0.; // initial time
    double deltatau  = Tmaturity/(double)nt; // time step


    // Parameter data
    double qs = R0; //interest rate
    double ds = dividend; //dividend
    double rR = qs - ds;
    double r = R0; //free interest rate
    //double D0 = r - rR;
    double lambdaB = 0.;//0.04; //intensity of default from counterparty B
    double lambdaC = 0.; //intensity of default from counterparty C
    double RB = 1.;//0.3; //Recovery rate of B
    double RC = 1.; //Recovery rate of C

    double sF = (1.-Recovery)*delta_PD;//(1.-RC)*delta_PD;//(1 - RB) * lambdaB;


    //spatial discretization
    int nnod = nods * nodmu; // total nodes in the mesh
    int nelt = (nods - 1) * (nodmu - 1) * 2; // number of elements in the mesh
```

```
int it;
double tau,t;

int feller;

int* connect;
int* boundary1;
int* boundary2;
int* boundary3;
int* boundary4;
double* meshS_1d;
double* meshmu;
double* U1dtotal;
double* mesh;
double* meshS;


PnlMat *Ah_V;
PnlMat *Ah_XVA;
PnlMat *Ah_V_ini;
PnlMat *Ah_XVA_ini;

PnlVect *Vfin;
PnlVect *Vold;
PnlVect *V;
PnlVect *V_no_bc;
PnlVect *Vrisky;


PnlVect *XVAfin;
PnlVect *XVAold;
PnlVect *XVA;
PnlVect *XVA_no_bc;

PnlVect* U1d;

PnlVect* bh_V;
PnlVect* bh_XVA;
PnlVect* XVA_solution_t;
```

```
  //--------------------reescalate domain ------------------------------
  double mumin = 0.; //minimum value of volatility in the reescalate domain
  double mumax = 1.; //maximum value of volatility in the reescalate domain
  double smin = 0.; //minimum value of asset price in the reescalate domain
  double smax = 1.; //maximum value of asset price in the reescalate domain
  //-------------------original domain-----------------------------------
  double muinf = 0.;//sqrt(var0); //minimum value of the volatility in the ori
  double musup = 1.; //maximum value of the volatility in the original domain
  double ssup = 4*strike; //maximum value of the asset price in the original d
  double sinf = 0.; //minimum value of the asset price in the original domain

  //------------------------------------------------------------------------
  double deltas = (double) (smax - smin) / (nods - 1);
  double deltamu = (double) (mumax - mumin) / (nodmu - 1);

  //nodes and weight associated with the integral in the reference triangle. C
  double nodesg[6];
  double weightg[3];

  double xy[2];


if ((p->Compute) == &Call)
 opt = 0;
else
  opt = 1;

  // build the mesh
  //------------uniform mesh----------------------------

  mesh = (double*) malloc(sizeof(double)*2*nnod);

  mesh2d(mesh, smax, mumax, smin, mumin, nods, nodmu);

  meshS = (double*) malloc(sizeof(double)*nnod);
  for (i = 0; i < nnod; i++) {
      meshS[i]=mesh[i] ;
  }


  //-----------------------------------------------
```

```
//Feller condition
if (2*kappa*theta>gamma*gamma){
    feller = 1;
    //printf("Feller condition is satisfied\ n");
}
else{
    feller = 0;
    //printf("Feller condition is violated (should be fixed -> now set felle
    feller = 1;
}

//-------------------------------------------------

//definition of the elements in the mesh

connect = (int*) malloc(sizeof(int)*3*nelt);
for (i = 0; i < 3*nelt; i++) {
    connect[i]=0.;
}

connectivity(connect, nods, nodmu, nelt);


//-----------------------------------------------

//Definition of the boundaries

boundary1 = (int*) malloc(sizeof(int)*nods);
boundary2 = (int*) malloc(sizeof(int)*nodmu);
boundary3 = (int*) malloc(sizeof(int)*nods);
boundary4 = (int*) malloc(sizeof(int)*nodmu);


for (i= 0; i<nods; i++){
    boundary1[i] = i;
    boundary3[i] = nodmu*( nods-1)+i;
}

for (i= 0; i<nodmu; i++){
    boundary2[i] = (nods-1) + i*nods;
    boundary4[i] =  i*nods;
```

```
}

//-------------------------------------------------------------------------
//Building of system matrix

// Create full matrix but can use sparse matrix
Ah_V = pnl_mat_create(nnod, nnod);
Ah_XVA = pnl_mat_create(nnod, nnod);
Ah_V_ini = pnl_mat_create(nnod, nnod);
Ah_XVA_ini = pnl_mat_create(nnod, nnod);


nodesg[0] = (float)1./2.;
nodesg[1] = 0.;
nodesg[2] = (float)1./2.;
nodesg[3] = 0.;
nodesg[4] = (float)1./2.;
nodesg[5] = (float)1./2.;


weightg[0] = (float)1./6.;
weightg[1] = (float)1./6.;
weightg[2] = (float)1./6.;

matbuilt(Ah_V, Ah_XVA, nodesg, weightg, rho, gamma, musup,
         connect, nelt, mesh, r, lambdaB, lambdaC, deltatau, nnod);


pnl_mat_clone (Ah_XVA_ini, Ah_XVA);
pnl_mat_clone (Ah_V_ini, Ah_V);


Vfin = pnl_vect_create(nnod);
Vold = pnl_vect_create(nnod);
V = pnl_vect_create(nnod);
V_no_bc = pnl_vect_create(nnod);
Vrisky = pnl_vect_create(nnod);
pnl_vect_set_zero(Vfin);
pnl_vect_set_zero(Vold);
pnl_vect_set_zero(V);
pnl_vect_set_zero(V_no_bc);
```

```
pnl_vect_set_zero(Vrisky);


payoff(Vfin, meshS, strike, ssup, opt, nnod);
for(i = 0;i<nnod;i++){
    pnl_vect_set(Vold,i, pnl_vect_get(Vfin,i));
}

XVAfin = pnl_vect_create(nnod);
XVAold = pnl_vect_create(nnod);
XVA = pnl_vect_create(nnod);
XVA_no_bc = pnl_vect_create(nnod);
pnl_vect_set_zero(XVAfin);
pnl_vect_set_zero(XVAold);
pnl_vect_set_zero(XVA);
pnl_vect_set_zero(XVA_no_bc);

meshS_1d = (double*) malloc(sizeof(double)*nods);

for(i =0;i<nods;i++) {
    j = boundary3[i];
    meshS_1d[i] = ssup * mesh[j];
}

meshmu = (double*) malloc(sizeof(double)*nodmu);
for(i =0;i<nodmu;i++) {
    j = boundary2[i];
    meshmu[i] = (musup-muinf)*mesh[j + 1*nnod] + muinf;
}

U1dtotal = (double*) malloc(sizeof(double)*nods*nt*nodmu);
for (i = 0;i<nods*nt*nodmu;i++){
    U1dtotal[i]=0.;
}


U1d = pnl_vect_create(nods*nt);
pnl_vect_set_zero(U1d);


for (i = 0;i<nodmu;i++){
```

```
    solver_BS_risk(U1d, Tin, Tmaturity, nt,
                   nods, meshS_1d,
                   r, sqrt(meshmu[i]), kappa, qs, ds,
                   sF, lambdaB, lambdaC, RB, RC,
                   sinf, ssup, strike, opt);
    for(j = 0;j<nods*nt;j++){
        U1dtotal[j + i*nods*nt] = pnl_vect_get(U1d,j);
    }
}

bh_V = pnl_vect_create(nnod);
bh_XVA = pnl_vect_create(nnod);
XVA_solution_t = pnl_vect_create(nnod);

pnl_vect_set_zero(bh_V);
pnl_vect_set_zero(bh_XVA);
pnl_vect_set_zero(XVA_solution_t);


// Temporal loop
for (it = 0; it<nt;it++){

    tau = (it+1) * deltatau; //forward time

    t = Tmaturity - tau; //real time

    vectbuilt(bh_V, nnod, nelt, mesh, connect, nodesg, weightg,
              deltatau, musup, rho, gamma,
              rR, kappa, theta, smin, smax, mumin, mumax, Vold, nods, nodmu,

    LU_crout_LUDO(V_no_bc, Ah_V, bh_V, nnod);

    conditionsV(Ah_V, bh_V,
                tau, boundary1, boundary2, boundary3,
                Vold, nnod, nods, nodmu,
                deltatau, musup, ssup, mesh,
                r, kappa, theta, rR, strike,
                smin, smax, mumin, mumax,
                opt, feller, V_no_bc, deltamu, deltas);
```

```
LU_crout_LUDO(V, Ah_V,bh_V, nnod);


// Copy new values in old values (with max 0 or absolute values)
for(i = 0;i<nnod;i++){
    pnl_vect_set(Vold,i,pnl_vect_get(V,i));
}


vectbuilt_XVA(bh_XVA, nnod, nelt, mesh, connect,
              nodesg, weightg, deltatau, musup, sF,
              rho, gamma, rR, lambdaB, lambdaC, RB, RC,
              kappa, theta, smin, smax, mumin, mumax,
              XVAold, V, nods, nodmu, deltas, deltamu );

LU_crout_LUDO(XVA_no_bc, Ah_XVA,bh_XVA, nnod);

//XVA one dimensional solution at each instant of time
for (i = 0;i<nods;i++){
    for(j = 0;j<nodmu;j++){
        pnl_vect_set(XVA_solution_t,j+i*nodmu, U1dtotal[i + it*nods + j*
    }
}


conditionsXVA(Ah_XVA, bh_XVA,
              tau, boundary1, boundary2, boundary3,
              V, XVAold, nnod, nods, nodmu, deltatau, musup, mesh,
              r, rho, kappa, gamma, theta, rR,
              sF, strike, lambdaB, lambdaC, RB, RC,
              smin, smax, mumin, mumax,
              XVA_solution_t, feller, XVA_no_bc, deltamu, deltas);

LU_crout_LUDO(XVA, Ah_XVA,bh_XVA, nnod);


for(i = 0;i<nnod;i++){
    pnl_vect_set(XVAold,i , pnl_vect_get(XVA,i));
}
```

```
    for (i =0; i<nnod; i++){
        for (j = 0;j<nnod; j++){
            pnl_mat_set(Ah_XVA, i, j , pnl_mat_get(Ah_XVA_ini, i, j));
            pnl_mat_set(Ah_V, i, j , pnl_mat_get(Ah_V_ini, i, j));
        }
    }
}


xy[0]=S0/ssup;
xy[1]=V0/musup;

*CVA =-interpolation_quadrangle(xy, XVA, mesh, nnod, nods, nodmu, deltas, de


free(connect);
free(boundary1);
free(boundary2);
free(boundary3);
free(boundary4);
free(meshS_1d);
free(meshmu);
free(U1dtotal);
free(mesh);
free(meshS);



pnl_mat_free(&Ah_V);
pnl_mat_free(&Ah_XVA);
pnl_mat_free(&Ah_V_ini);
pnl_mat_free(&Ah_XVA_ini);


pnl_vect_free(&Vfin);
pnl_vect_free(&Vold);
pnl_vect_free(&V);
pnl_vect_free(&V_no_bc);
pnl_vect_free(&Vrisky);

pnl_vect_free(&XVAfin);
```

```
    pnl_vect_free(&XVAold);
    pnl_vect_free(&XVA);
    pnl_vect_free(&XVA_no_bc);


    pnl_vect_free(&U1d);
    pnl_vect_free(&bh_V);
    pnl_vect_free(&bh_XVA);
    pnl_vect_free(&XVA_solution_t);

  return OK;

}

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

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

  return FDFiniteElement_CVA(ptMod->S0.Val.V_PDOUBLE,
                  ptOpt->PayOff.Val.V_NUMFUNC_1,
                  ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,ptOpt->PayOf
                  r,
                  divid, ptMod->Sigma0.Val.V_PDOUBLE
                  , ptMod->MeanReversion.Val.V_PDOUBLE,
                  ptMod->LongRunVariance.Val.V_PDOUBLE,
                  ptMod->Sigma.Val.V_PDOUBLE,
                  ptMod->Rho.Val.V_PDOUBLE, ptMod->Recovery.Val.V_DOUBLE,ptMod-

}

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

  return  WRONG;
```

```
}
#endif //PremiaCurrentVersion

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

     Met->HelpFilenameHint = "FD_FiniteElement1_HestonCVA";
     Met->Par[0].Val.V_PINT = 100;
     Met->Par[1].Val.V_PINT = 25;
     Met->Par[2].Val.V_PINT = 25;
    }


  return OK;
}


PricingMethod MET(FD_FiniteElement_HestonCVA) =
{
  "FD_FiniteElement_HestonCVA",
  { { "Number of Time Steps", INT, {100}, ALLOW},{"Number of Space Steps", LONG,
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(FD_FiniteElement_HestonCVA),
   { {"CVA", DOUBLE, {100}, FORBID},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(FD_FiniteElement_HestonCVA),
  CHK_mc,
  MET(Init)
};
```