Scilab Code Manual

Authors: G. Allaire, A. Karrman, G. Michailidis

Last revision: January 2012

An explanation of the basic characteristics, of the Scilab code we have used here, has already been given in [6]. However, this description mostly aimed to familiarize one with this optimization method and to help somebody obtain some optimal shapes in an easy way. We will try here to give a more thorough explanation of the details of the code, which will allow one to interfere to the original files and adjust the code to his needs.

We will analyze the original files that were used in [6] and we will give any further detail separately. As already mentioned in [6], three basic files are needed for the code to run. The first one, called functions.sci contains all the necessary functions for the algorithm. The second one takes its name from the specific shape and structural problem to which it refers to. For example, the bridge.sce file contains the geometry information, the boundary conditions and an "appropriate" loading for the study of a typical bridge. The last file, optalg.sce, contains the optimization algorithm and makes use of the two previous files.

Since the functions.sci, as well as the optalg.sce files use much information described in the specific problem's file, we prefer to start the description with the last one. We will present just the bridge.sce file, since elementary changes are needed to consider another problem. For the sake of comprehension, we have split each code in several parts.

The version bridgetop.sce should be used instead of bridge.sce if the user wants to use the topological derivative in addition to the shape derivative. In this version, we have added some more lines which we explain separately.

i) bridge.sce

Lines 1-25:

```
//////////////////////////////////////////////////////////////////
// Copyright G. Allaire, A. Karrman, October 2009
//
// A Scilab toolbox for 2-d structural optimization
// by the level set method.
//
// Based on the work of G. Allaire, F. Jouve, A.-M. Toader,
// J. Comp. Phys. Vol 194/1, pp.363-393 (2004).
//////////////////////////////////////////////////////////////////
//
// This file contains the parameters of the following test case:
// bridge
//////////////////////////////////////////////////////////////////
// PARAMETERS
nelx = 40 ;          // number of elements count in x direction
nely = 40 ;          // number of elements count in y direction
xlength = 2. ;       // working domain length
dx = xlength/nelx ;  // x space step size
dy = dx ;            // y space step size
yheight = dy*nely ;  // working domain height
```

```
hx = 5 ;             // number of holes in x direction
hy = 6 ;             // number of holes in y direction
r = .7 ;            // hole size (between 0 and 1)
```

In these first lines, some general characteristics of the structure are given. We define the dimensions of the shape, as well the number of elements to be used, as it is obvious by the comments next to the code. Of course, the consideration **dy=dx** is not obligatory, and one can create a finer mesh in one of the two directions.
At the last three lines, an easy choice for creating structures with different initial topologies is given, which is based on an initial consideration of the level set function describing the structure given in the file functions.sci. After choosing the number of holes in each direction, we can modify their size by varying **r**. Increasing its value, we get larger holes in the initial topology of the shape.
The reader should not forget that our method is strongly dependent on the initial topology, i.e. it is sensitive to local minima. Thus, changing the last three parameters, one can observe differences in the topologies of the resulting optimal shapes.

<u>Lines 27-29:</u>

```
eps = .001 ;         // "hole" material density
lagV = 15. ;         // the volume Lagrange multiplier
lagP = 0.0 ;         // the perimeter Lagrange multiplier
e2 = 4*dx^2.;     // coefficient for the regularization in front of the Laplacian
```

The value **eps** is used in the “ersatz material” approach that we follow, in order to represent the weak material mimicking void. It must have a very low value (eps<<1) to limit its impact, but such that we avoid the singularity of the rigidity matrix.
As we have explained earlier, the **lagV** and **lagP** represent the weight multipliers of the volume and perimeter correspondingly in the multi-objective optimization. The reader should expect that decreasing too much the **lagV** multiplier in **compliance minimization** should result in a great increase in the volume of the structure. It is possible that the whole working domain gets covered by the full-material as a result of reducing the significance of the volume’s contribution to the objective function.
The **lagP** multiplier is usually used to penalize topologies with many holes and thus create simpler topologies in the optimal structures. This is clearly a result of penalizing the total perimeter, which is very high when the number of connected components of the shape is big.
Both values of the last two multipliers are heuristic and should by chosen by the user after performing some numerical tests on his specific problem.
The coefficient **e2** is used to multiply the Laplacian in the regularization algorithm, so as to control the regularization effect.

<u>Lines 31-34:</u>

```
nodex = linspace(0,xlength,nelx+1) ;  // x space for nodes
nodey = linspace(0, yheight,nely+1) ; // y space for nodes
FEx = linspace(0,xlength,nelx) ;      // x space for finite elements
FEy = linspace(0, yheight,nely) ;     // y space for finite elements
```

Here, we just create some vectors containing the coordinates of the points of the grid, which can be used in describing a desirable initial shape for the structure to be optimized.

Lines 36-42:

```
RIiterinit = 50 ; // number of time steps in the re-initialization of the initial design
RIiter = 5 ; // number of time steps in the re-initialization of further designs
RIfreq = 5 ;  // frequency of re-initialization, i.e. number of time steps in the
// transport level set equation between two re-initializations
HJiter0 = 30 ; // original number of transport time steps
entire = 20 ; // total number of optimization iterations
allow = .02 ; // fraction that the objective function can increase
```

The first three variables of these lines play a significant role in the method we are using, i.e. the level-set method. The reason for it is that the level-set function describing the shape of the structure can become very steep during the optimization process, especially near the border. Since the accuracy of the approximation of several geometrical features of the shape, such as the normal vector of the surface or the mean curvature, is crucial for our method, the steepness of the level-set function can result in serious errors.

In our code, we start with an initial function that takes the value -1 inside the structure (full-material) and 1 outside (weak-material). Then we need to construct a signed-distance function out of it. For this reason, the number of steps of the initial re-initialization should be big enough. This generally depends also on the size of our mesh. We would say that the number of the initial iterations **RIiterinit** should be such that the wave can transfer the necessary information to all of the structure. For the reader who wants to understand better this procedure, we address to [8].

As we foresaid, even if the initial approximation of the signed-distance function to the boundary of the shape is satisfying, numerical experience shows that the level-set function goes far from keep being a signed-distance function to the new boundary after some optimization iterations. Therefore, we use **RIfreq** to define after how many iterations of the Hamilton-Jacobi advection equation the level-set function should be re-initialized and **RIiter** to define the number of steps for the re-initialization. This last number should be such that the re-initialization is satisfying at least close to the border.

We propose that the reader performs some numerical tests with various values for these variables and checks their effectiveness on his problem, by plotting the level-set function. Also, the reader should take great care when his problem demands that he works very close to the signed-distance function. Then, we propose that **RIfreq=1** and that he uses a big enough number of **RIiter** to have a good approximation, since the re-initialization procedure is not in general numerically expensive.

With **HJiter0** we choose the step we take in the gradient method. In general, the time step **dt** coming from the CFL condition is very small in comparison to the optimization step that reduces the objective function. Since the advection equation is solved explicitly and so it is numerically cheap, we prefer to take much more than one step at each optimization iteration, since the latter involves the solution of a linear system with the FEM.

The variable **entire** just defines the number of iterations for the optimization algorithm, since we prefer not to use a stopping criterion, which is not clear in shape optimization.

The variable **allow** is considered for numerical reasons. In fact, numerically, a topological change cannot be as small as we want, but instead depends on our grid. Therefore, it is possible for example that a topological change indicated by the gradient method is larger that it should be to reduce the objective. Thus, we allow such small increases of the objective, expecting that after the topological change takes place, the algorithm will search for a better minimum point.

<u>Lines 44-48:</u>

```
// FINITE ELEMENT MATRICES
KE = lk(); // the stiffness matrix
K = sparse([],[],[2*(nelx+1)*(nely+1), 2*(nelx+1)*(nely+1)]); // the global stiffness matrix
F = sparse([],[],[2*(nely+1)*(nelx+1),2]); // the matrix of applied forces
U = sparse([],[],[2*(nely+1)*(nelx+1),2]); // the vector displacement matrix

// FINITE DIFFERENCES MATRIX
K1 = sparse([],[],[(nelx+1)*(nely+1),(nelx+1)*(nely+1)]) ;  //We define the matrix of the
velocity regularization
```

Here we just define the matrices for the FEM, as well as for the finite differences used for regularizing the advection velocity. The **sparse** definition in Scilab results in a great gain in computational time and memory and the user should use it whenever possible.

<u>Lines 50-64:</u>

```
// SETTING OF THE ELASTICITY PROBLEM

  // THE FORCE
  // Each row corresponds to a different force.
  // The first two values are the fraction along the x and y axes of the working
  // domain (the origin is the top left corner).  The third value is binary indicating
  // whether the force is horizontal (0) or vertical (1).  The fourth value gives
  // the strength of the force and its direction (negative for leftward or downward
  // forces; positive for rightward or upward forces).
  forceMatrix = [.5 1 1 -1] ;

  forceCount = size(forceMatrix,1) ; // We count the number of applied forces
  for force = 1 : forceCount
    F(c(forceMatrix(force,1:3)),force) = forceMatrix(force,4) ;
  end
```

The way forces are described is very clearly described in the comments above. We suggest that the user takes care so that the region where the loads are applied, as well as some part of the boundary where we impose Dirichlet conditions is covered with the full-material.

<u>Lines 66-70:</u>

```
  // FIXED BOUNDARIES WITH DIRICHLET CONDITIONS
   fixeddofs = [c([0 1 1]) c([0 1 0]) c([1 1 1])] ; // We fix x and y degrees of freedoms for
nodes

  alldofs     = [1:2*(nely+1)*(nelx+1)];
  freedofs    = setdiff(alldofs,fixeddofs);
```

In this part we define the parts of the boundary where the structure is clamped (Dirichlet conditions). We do this with the use of the **c** function. The first and the second arguments in the bracket give the fractions along the x and y axes that correspond to the location of the point

to clamp. The third argument takes the value 0 or 1, depending on whether we fix the horizontal or the vertical degree of freedom. So, in the lines above the first call of the **c** function fixes the vertical displacement of the down-left node, the second call fixes the horizontal displacement of the same node and the third call fixes the vertical displacement of the down-right node of the working domain.
At this point, we prefer to analyze a little bit more the definition of the axes in our code. The user shall be very careful with it when trying to interfere to the code.
The location of the forces, the location of the points where we impose boundary conditions and other features that are directly related to the coordinates of the nodes, follow the way these coordinates are defined. So, since the coordinates of the grid's nodes were defined in an increasing order through the matrices **nodex**, **nodey** (lines 31-32), the axes for all these features should be the ones shown in Figure 1. The reader should not be confused with features defined in another sense, such as for example the direction of the forces, which is different than the positive direction defined in Figure 1.

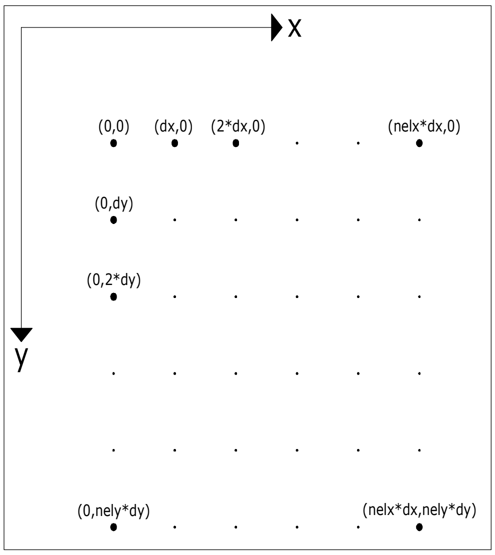

Figure 1: Definition of axes for the node coordinates.

The numbering of the degrees of freedom is presented in Figure 2.

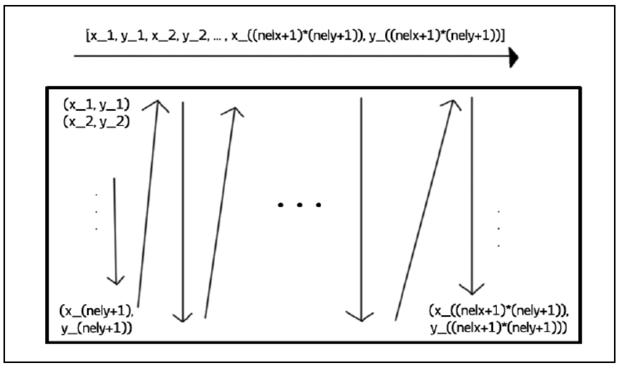
[x_1, y_1, x_2, y_2, ... , x_((nelx+1)*(nely+1)), y_((nelx+1)*(nely+1))]
(x_1, y_1)
(x_2, y_2)
(x_(nely+1), y_(nely+1))
(x_((nelx+1)*(nely+1)), y_((nelx+1)*(nely+1)))

Figure 2: Numbering of degrees of freedom ([6]).

<u>Lines 72-77:</u>

```
// PASSIVE ELEMENTS
// For this part we can make sure that certain areas in the working domain
// are either always part of the structure or always not part of the structure.
function FEthetaOut = passive(FEthetaIn)
  FEthetaOut = FEthetaIn ;
endfunction
```

In these lines we use a function that keeps some elements of the working domain with the full-material’s density. The reason is that in some problems this is physically imposed, as for example the pavement of the bridge should always exist, in any configuration of a bridge’s shape!

<u>Lines 80-82:</u>

```
// INITIALIZATION
phi0 = mesh0(hx, hy,r) ;
phi0(($-floor(.08*nely)):$,:) = -.1 ;
```

Finally, in these lines we call the **mesh0** function that gives the initial function describing the shape, under the characteristics we have attributed to it.

ii) bridgetop.sce

<u>Lines added:</u>

```
allow_top = .1 ; // fraction that the objective function can increase in topological derivation
ntop = 6 ; //number of advection steps between two topological gradient steps
percentage_in = 0.05; // percentage of the current volume to be removed in the first topological
                            // gradient step
percentage = 0.02; // percentage of the current volume to be removed in each topological
                        // gradient step
```

A thorough explanation for the choice of these parameters can be found in [13]. We will also give a short description here for the sake of completeness.
We have defined a different allowance for the topological sensitivity step, named **allow_top**, as suggested in [13], which is significantly higher than the one used in the shape sensitivity step. This is mainly due to the inability to take numerically a “small step” in the topological derivative algorithm, that is to create very small holes. To create a hole we have to change the sign of at least one node of the mesh and therefore the step cannot be arbitrarily small, unless the mesh is very dense.
Then, since we have decided to couple the shape and topological derivatives, we have to choose a frequency for the execution of the topological step, called **ntop**.
The **percentage** parameter defines the percentage of the current volume to be removed in every topological gradient step. In the same sense, we have used the parameter **percentage_in** just for the first step. It is sometimes useful to set this value bigger that the one of **percentage** when we start with a full-domain initialization (no initial guess of the shape). This happens because numerically it may be difficult to remove a small volume when the topological gradient at the nodes takes values that are very close between them.

iii) functions.sci

We continue with the file containing the functions that we use in the optimization algorithm. We have chosen not to give a detailed description of all of them, since they are described in a very satisfying way in the comments, but also because the user has to use them as “black box” and it is not recommended to interfere into them. For this reason, we avoid describing the **FE** and **FEtop** function for the finite element analysis and the **lk** function for the forming of the stiffness matrix, which in fact are a translation in Scilab of Ole Sigmund’s code in [11]. Moreover, we avoid detailing the **c** function, which as we have already seen fixes the degrees of freedom of grid points and we give a short description of the **FEdensity** function, without presenting the code, which will help the reader understand why some regions of the domain appear in different grayscale than others.

FEdensity

The concept under which we attribute a density value to each element is the following: First, we check the values of the level-set function at the four grid points forming an element. If all of the values are negative, that means that all the nodes are contained in the structure and so the full-material density is given to the element. Accordingly, if all the values are positive, that means that no point is included, so we have to give the weak-material’s density to the element. If nothing of these happens, then we have to give some intermediate value of density. So, we split the rectangular element into four triangles and we give to the common point of the triangles, as value of the level-set function, the average of the values of the four points. Then, we examine each triangle separately under the some logic. The final element’s density is the average of the contribution of each triangle.

mesh0

```
// INITIALIZE THE STRUCTURE
// This function will just distribute holes
// uniformly throughout our mesh.  hx is the number
// of horizontal holes while hy is number of vertical
// holes.  r is a variable that lets you change the
// size of the holes;  a default size would be .5; .1
// would give you very small holes and .9 would give
// you very large holes.  (0 < r < 1)
function [phi0] = mesh0(hx,hy,r)
  if isdef('ntop') then
   phi0 = -ones(nely+1,nelx+1) ; // Full-domain initialisation
  else
   phi0 = zeros(nely+1,nelx+1) ; // First create an empty matrix
   //In order to make our holes, we use a 3d function
   //that maps x and y coordinates to z = cos(x)*cos(y)
   //and then we flatten holes to .1 and the structure
   //part to -.1
   phi0 = -cos((hy+1)*(nodey*%pi)/yheight)'*cos((hx+1)*(nodex*%pi)/xlength)+r-1 ;
   phi0 = .2*ceil(max(phi0,0))-.1 ;
  end
endfunction
```

The way the initialization takes place is described in the comments in complete detail. We would like to mention that the level-set function is in fact a matrix, and it is formed in the same sense that the grid is formed, i.e. it follows the numbering of the **nodex** and **nodey** vectors. So, the coordinate system for the level-set function is shown in Figure 3.

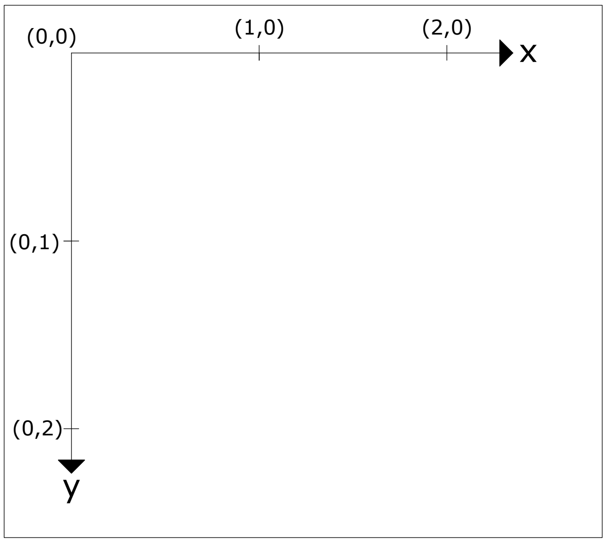

Figure 3: Coordinate system for the level-set function.

Thus, if the user wants to create his own initialization, then he should replace the line:
"phi0 = -cos((hy+1)*(nodey*%pi)/yheight)'*cos((hx+1)*(nodex*%pi)/xlength)+r-1 ;" by his own representation. Below, we give some examples for a better understanding of what we have mentioned.

Example 1: Circular hole with center (x,y) = (1,1) and radius r = 0.4.

```
for i = 1:nely+1
    for j = 1:nelx+1
                    phi0(i,j) = -(nodex(j)-1)^2-(nodey(i)-
1)^2+0.4^2 ;
    end
end
```

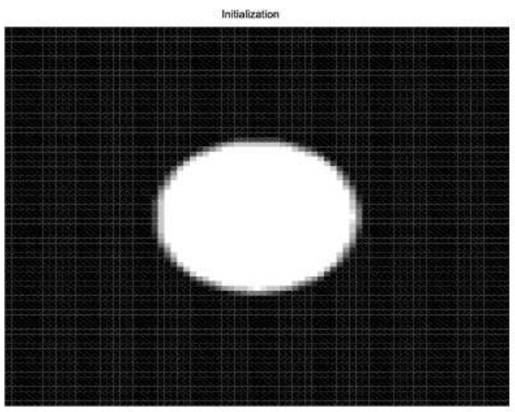

Figure 4: Circular hole with equation: $(x-1)^2+(y-1)^2=0.4^2$.

Example 2: Circular hole with center (x,y) = (1.5,0.5) and radius r = 0.2.

```
for i = 1:nely+1
  for j = 1:nelx+1
            phi0(i,j)  =  -(nodex(j)-1.5)^2-(nodey(i)-
0.5)^2+0.2^2 ;
  end
end
```

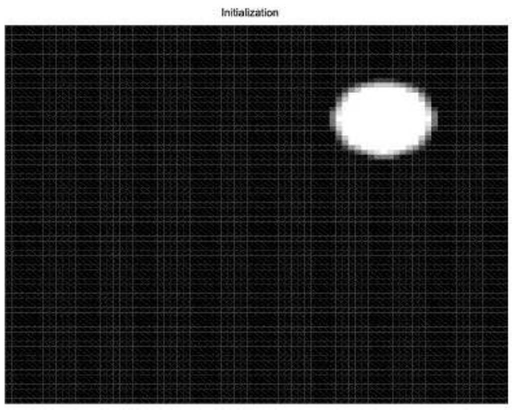

Figure 5: Circular hole with equation:

$$\left(x-1.5\right)^2+\left(y-0.5\right)^2=0.2^2\,.$$

shift2n

```
// SPACE SHIFT FUNCTION
// Using Neumann, or Neumann for the directional gradient boundary conditions,
// we just shift
// our matrix over one index in a certain direction
// in order to take derivatives.
function phishift = shift2n(direction,phi,conditions)
  // SHIFTS LEVEL SET FUNCTION WITH NEUMANN OR NEUMANN FOR THE
GRADIENT CONDITIONS
 select direction
   case 'w' then        // SHIFT WEST
        [m,n] = size(phi) ;
        phishift(1:m,1:n-1) = phi(1:m,2:n) ;
        select conditions
          case 'n' then
             phishift(1:m,n) = phi(1:m,n) ;    // NEUMANN CONDITIONS
          case 'ng' then
                    phishift(1:m,n) = 2*phi(1:m,n)-phi(1:m,n-1) ; // NEUMANN FOR THE
DIRECTIONAL GRADIENT CONDITIONS
        end
   case 'e' then        // SHIFT EAST
        [m,n] = size(phi) ;
        phishift(1:m,2:n) = phi(1:m,1:n-1) ;
```

```
          select conditions
            case 'n' then
                phishift(1:m,1) = phi(1:m,1) ;     // NEUMANN CONDITIONS
            case 'ng' then
                        phishift(1:m,1) = 2*phi(1:m,1)-phi(1:m,2) ;    // NEUMANN FOR THE
DIRECTIONAL GRADIENT CONDITIONS
          end
    case 'n' then         // SHIFT NORTH
          [m,n] = size(phi) ;
          phishift(1:m-1,1:n) = phi(2:m,1:n) ;
          select conditions
            case 'n' then
                phishift(m,1:n) = phi(m,1:n) ;      // NEUMANN CONDITIONS
            case 'ng' then
                       phishift(m,1:n) = 2*phi(m,1:n)-phi(m-1,1:n) ;   // NEUMANN FOR THE
DIRECTIONAL GRADIENT CONDITIONS
          end
    case 's' then         // SHIFT SOUTH
          [m,n] = size(phi) ;
          phishift(2:m,1:n) = phi(1:m-1,1:n) ;
          select conditions
            case 'n' then
              phishift(1,1:n) = phi(1,1:n) ;      // NEUMANN CONDITIONS
            case 'ng' then
                     phishift(1,1:n) = 2*phi(1,1:n)-phi(2,1:n) ;          // NEUMANN FOR THE
DIRECTIONAL GRADIENT CONDITIONS
          end
    else
          error('SHIFT N,S,E, OR W?')
  end
endfunction
```

We suggest that the user takes great attention to this function, especially the one that wants to interfere to the codes or create his own ones. This function is indeed used to form finite differences to any direction, so that after we can easily create forward or backward schemes for approximating derivatives. **We need to mention that the numbering of the arrays follows the coordinate system described earlier, considering the upper-left corner as the origin.**
Considering Neumann conditions for the level-set function, all we have to do is to define the direction towards which we want to move the discrete values of the level-set function and keep the same values for the “void” part.
In the following Figures, we try to explain in detail the numbering of arrays and the procedure of moving the values of phi (level-set function) to the west and we give the form of the translated function. It is easy then to understand that the directions west, east, north and south correspond to the directions of the real world and have no relation to any other coordinate system defined in the code.
Another possible choice for the boundary conditions is to take Neumann conditions for the directional derivative. This seems to be a more natural choice, since it alleviates the artificial effect on boundaries that are perpendicular to the boundary of the working domain.
In this work we have chosen to use Neumann conditions for the reinitialization algorithm and Neumann for the directional gradient for all the rest.

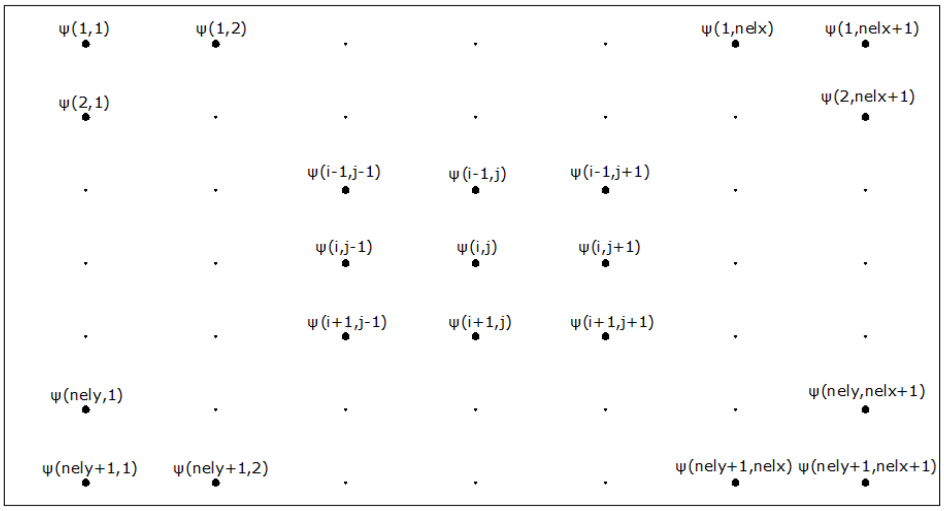

Figure 6: Numbering of arrays.

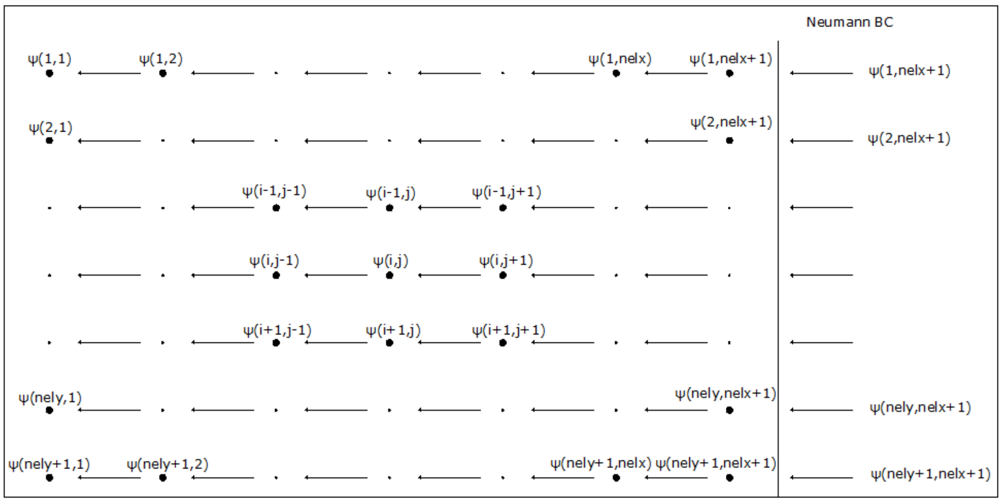

Figure 7: Moving the values of the level-set function to the west.

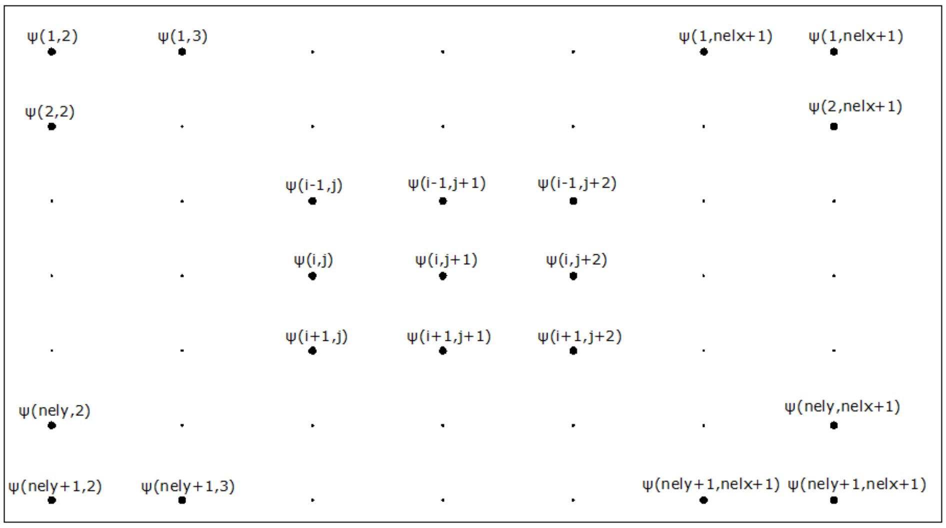

Figure 8: Moved to the west level-set function.

perimeter

```
// THE PERIMETER
// In order to roughly calculate the perimeter
// we find the norm of the gradient of the sign
// of phi and integrate it, then divide by 2:
function totperim = perimeter(phi)
  // To smooth sign(phi):
  epsperim = min(dx,dy)/20 ;
  sx = phi./sqrt(phi.^2+epsperim^2) ;

  sxn = shift2n('n',sx,'ng') ;
  sxs = shift2n('s',sx,'ng') ;
  sxe = shift2n('e',sx,'ng') ;
  sxw = shift2n('w',sx,'ng') ;

  // We now calculate d(phi)/dx and d(phi)/dy:
  dsxx = (sxw-sxe)/(2*dx) ;
  dsxy = (sxn-sxs)/(2*dy) ;

  dV = dx*dy ;

  // And then integrate:
  totperim = .5*sum(sqrt(dsxx.^2+dsxy.^2))*dV ;
endfunction
```

For the calculation of the perimeter, we use the following approximation: $perimeter = \int_{\partial\Omega} ds = \int_D \delta_{\partial\Omega} dx$, where $\delta_{\partial\Omega} = \frac{1}{2}\left|\nabla \operatorname{sgn}\left(\psi(x)\right)\right|$ is the dirac mass function of the boundary $\partial\Omega$.

We use an approximation of the sign function: $\operatorname{sgn}\left(\psi(x)\right) = \frac{\psi(x)}{\sqrt{\psi(x)^2 + (epsperim)^2}}$,

where the value of epsperim=min(dx,dy)/20 is chosen so as to spread the jump in the sign function over 2 cells in average. In the code, we have called $\operatorname{sgn}\left(\psi(x)\right) = sx$. Then, we shift this function to all directions, and we form the central differences approximations of the derivatives of sx in the direction x (**dsxx**) and y (**dsxy**). So, we have formed the components of $\nabla \operatorname{sgn}\left(\psi(x)\right)$ and the perimeter is easily calculated by the approximation we gave above.

curv

```
// THE CURVATURE
// The shape gradient of the perimeter is the
// mean curvature which we compute by
// div(grad(phi)/|grad(phi)|)
// where phi is the level set function.
function H = curv(phi)
  // When finding the normal vector, we need
  // to divide by the norm of the gradient of phi;
  // we use this small value to make sure that
  // the gradient of phi never goes to 0:
  epscurv = min(dx,dy)/20 ;

  // Here are the first derivatives for finding
  // the gradient of phi:
  phin = shift2n('n',phi,'ng') ;
  phis = shift2n('s',phi,'ng') ;
  phie = shift2n('e',phi,'ng') ;
  phiw = shift2n('w',phi,'ng') ;

  dphix = (phiw-phie)/(2*dx) ;
  dphiy = (phin-phis)/(2*dy) ;

  //  Here is |grad(phi)| and then the x and y
  // components of the normal vector field:
  mag = sqrt(dphix.^2+dphiy.^2+epscurv^2) ;
  nx = dphix./mag ; ny = dphiy./mag ;

  // Now to find the divergence, just take the
  // partials with repsect to x and y of the
  // x and y components of our normal vector field
  // (respectively) and then add these together
  // to get our end mean curvature, which is a
  // function across our working domain.
  nxe = shift2n('e',nx,'ng') ;
  nxw = shift2n('w',nx,'ng') ;

  nyn = shift2n('n',ny,'ng') ;
  nys = shift2n('s',ny,'ng') ;

  divnx = (nxw-nxe)/(2*dx) ;
  divny = (nyn-nys)/(2*dy) ;

  H = divnx+divny ;
endfunction
```

Here we compute the mean curvature H, which in 2-d is the same with the min or max curvature. It is compute as: $H = div\left(\vec{n}\right)$, where $\vec{n}$ is the unit vector normal to the boundary $\partial\Omega$. Since $\vec{n} = \frac{\nabla\psi}{|\nabla\psi|}$ in terms of the level-set function, we first try to form the $\nabla\psi$. We shift the level-set function to every direction and we compute the central difference approximations of the derivatives in each direction. The user has to be careful with the fact that forming the normal vector requires that the differences are taken in the direction of the axes in Figure 3, else it is possible to form the tangent vector or the $-\nabla\psi$ vector. The small parameter added to the magnitude certifies that we do not divide with zero.
Finally, we follow exactly the same approach with the components of the normal vector to form its divergence.

volume

```
// THE VOLUME OF THE STRUCTURE
// To find the total volume of our structure, we just
// integrate the density*dV:
function totvol = volume(FEtheta)
  dV = dx*dy ;
  totvol = sum(FEtheta)*dV ;
endfunction
```

Since we have assigned a value of density to each element, the total volume, which is given by $V = \int_D \rho(x)\,dx$, is nothing else but the sum of the density values multiplied by the element's volume.

compliance

```
// COMPUTE THE COMPLIANCE
// The compliance is the main part of our objective function.
// It is the integral of the elastic energy density, equal to
// the total work done by the applied forces.
function totcomp = compliance(FEAeueu)
  totcomp = sum(FEAeueu) ;
endfunction
```

The compliance is the work done by the loads. We know that this equals the work of the internal forces: $\text{compliance} = \int_{\Gamma_N} g \cdot u\,ds = \int_D Ae(u):e(u)\,dx$. So, we just have to sum the values of the elastic energy density.

regularize

This function is used to regularize the velocity field for the advection of the level-set function, by substituting the L^2 with the H^1 inner product for the derivative. The user is not supposed to interfere with this function, but he can control the extend of regularization via the **e2** parameter given in the specific problem's file.

<u>solvelvlset</u>

This function is used for the advection of the level-set function under a velocity normal to the boundary computed via the shape sensitivity method [7,12]. The user is not supposed to interfere into this function and the same yields for the functions **minmod** and **g** that are used in it.

We do not give here a description of the code used, since the whole theory behind it can be found in detail in [10] and in the references contained there. Also, the function **mesh00** which is used for the re-initialization, is exactly of the same type with the advection equation and follows the same sense.

In general, we would say that the algorithm uses upwind schemes for approximating the advection part of the equation and central differences for the diffusive part. The logic is the same as with hyperbolic conservation laws and is summarized in the fact that "the numerical domain of dependence should contain the mathematical domain of dependence" ([10]). In other words, we have to check in which direction the wave moves to and advect the numerical information in the same direction. As for the diffusive part, it is natural that the information should be transmitted to both directions.

<u>solvelvlset_top</u>

This function is used for the update of the level-set function during a topological gradient step. The choice we have made is in accordance with [13], that is we remove a certain percentage of the volume of the current shape. The areas removed are those where the topological gradient takes its lowest values.

So, what we try in fact to do in this function is to determine which points should change sign so that the target volume is removed. This is done through a dichotomy algorithm. The user should note that the finer the mesh, the better this algorithm is expected to perform, since a finer mesh enables for the creation of small holes and a lower value of the parameter **percentage** can be chosen.

iii) optalg.sce

Lines 21-44:

```
// Check if the variable 'ntop' for topological gradient is defined.
if isdef('ntop') then
  ntop = ntop;
else
  ntop = 0;
end
// REINITIALIZATION
phi00 = mesh00(phi0,RIiterinit) ;

// Set phi to the reinitialized state:
phi = phi00 ;

// Plot the initialization:
scf(0);
clf()
xset('colormap',graycolormap(10))
title('Initialization')
grayplot(FEx,-FEy,-passive(FEdensity(phi))',axesflag = 2) ;

filename='Initialization';
xs2jpg(0,filename);

printf('\nOptimization started\n') ;
stacksize('max') ; // We want to make sure that we have enough memory available
```

We start our optimization algorithm by re-initializing the initial level-set function phi0, which was defined at bridge.sce, so that it becomes the signed-distance function to the zero level-set of phi0. Then, we set the level-set function phi equal to the re-initialized one.
After, we just plot the density of the initial shape in the graph with the title "Initialization" and we export the plot to a jpg file.
We would like here to mention something that can set a lot of questions to the user. We need to understand that we do not plot the structure itself, but instead the density of the material at the elements, which comes from interpolation procedure described in functions.sci. This means, that the boundary of the shape can be very smooth, but in the plot always the grayscaled image will appear.

Lines 46-77:

```
// FE ANALYSIS

// Define the elements' densities based on phi.
FEtheta = FEdensity(phi,eps) ;
// Set unchanging densities depending on the 'passive' function.
// This 'passive' zone is defined in the parameter file of the
// considered test case.
FEtheta = passive(FEtheta) ;
```

```
if ntop==0 then
  // shape derivative
  // The output of the finite element analysis is the elastic energy density:
  // lvlAeueu is that field defined on nodes (it is used as the velocity in
  // transport level set equation), FEAeueu is the same field defined on
  // elements (it is used to compute the compliance).
  [lvlAeueu,FEAeueu] = FE(FEtheta,KE,K,F,U) ;
else
  [FEAeueu,FEAeueucomp] = FEtop(FEtheta,KE,K,F,U);
  // topological derivative
  // In this case, FEAeueu denotes the topological gradient,
  // while FEAeueucomp contains the energy density defined on elements.
end
// Define the velocity field:
if ntop==0 then
  // shape gradient
  V = lvlAeueu/(dx*dy) - lagV;
  // regularization of the velocity field
  V = regularize(phi,V,e2,K1,lagP);
else
  // topological gradient
  V = FEAeueu - lagV;
end
```

This part is well described in the comments above. We use the **FEdensity** function to assign a density value to each element, according to the value of the level-set function at its nodes. Then we update these values by keeping steady some part of it via the **passive** function and finally we perform the FE analysis. In the case that ntop=0, that is if we use only the shape sensitivity method, having obtained the values of the energy density at each node, we define the values of the vector field, normal to the boundary, with which we will advect the level-set function and finally we regularize the advection velocity. In case we use the topological sensitivity, we have chosen to perform a topological gradient step at the first iteration, usually because in such a case we start with a full-domain initialization.

Lines 80-90:

```
// CALCULATE THE OBJECTIVE FUNCTION
if ntop==0 then
  totcomp = compliance(FEAeueu);
else
  totcomp = compliance(FEAeueucomp) ;
end
totvol = volume(FEtheta) ;
objective = lagV*totvol+lagP*perimeter(phi)+totcomp ;

// We track the objective function after each optimization iteration using 'objectivePlot'
objectivePlot = objective ;
```

In these lines we compute the objective function of our multi-objective optimization problem and save its value in the vector **objectivePlot**, which will be used for plotting the convergence history.

Lines 93-115:

```
// OPTIMIZATION LOOP

i = 1 ; // The current optimization iteration
HJa = 1 ; // The level set solution "attempt"
HJiter = HJiter0 ; // The initial number of time steps for solving the transport level set
equation
e3 = 1;  // Coefficient used to reduce, if neccessary, the advection time step below the
// limit imposed by the cfl condition
allow_adv = allow;

while i<=entire

  if modulo(i,ntop)==0 | (i==1 & ntop~=0) then
    // Test shape using the topological derivative:
    if (i==1) then
      phiTest = solvelvlset_top(phi,V,percentage_in,FEtheta);
    else
      phiTest = solvelvlset_top(phi,V,percentage,FEtheta);
    end
  else
    // Solve the transport level set equation using the velocity V:
    dt = 0.5*e3*min(dx,dy)/max(abs(V)) ;
    phiTest = solvelvlset(phi,V,dt,HJiter,lagP) ;
  end
```

It is time to start the optimization algorithm. Our stopping criterion has to do with the total number of optimization iterations, which is just the simplest consideration. More sophisticated criteria can of course be imposed.
In case the gradient comes from a shape sensitivity analysis, the time step **dt** is defined under a CFL condition described in [22], while other considerations can also be made.
We update the level-set function **phi** that describes our current shape to get a new level-set function **phiTest**. The reason for this notation is that we still don’t know if this new shape described by **phiTest** has led to a reduction of the objective and therefore we have to check this before adopting this new shape.

Lines 117-145:

```
  // FE ANALYSIS
  // Define material density based on phi
  FEthetaTest = FEdensity(phiTest,eps) ;
  // Set passive element densities to 'eps'
  FEthetaTest = passive(FEthetaTest) ;
  // Perform the finite element analysis and compute
  // the elastic energy density:
```

```
  if modulo(i,ntop)==0 | (i==1 & ntop~=0) then
    [FEAeueuTest,FEAeueucompTest] = FEtop(FEthetaTest,KE,K,F,U);
  else
    [lvlAeueuTest,FEAeueuTest] = FE(FEthetaTest,KE,K,F,U);
  end

  // CALCULATE THE OBJECTIVE FUNCTION
  if modulo(i,ntop)==0 | (i==1 & ntop~=0) then
    totcompTest = compliance(FEAeueucompTest) ;
  else
    totcompTest = compliance(FEAeueuTest) ;
  end
  totvolTest = volume(FEthetaTest) ;
  objectiveTest = lagV*totvolTest+lagP*perimeter(phiTest)+totcompTest ;
  // Plot the test shape:
  scf(0)
  clf()
  xset('colormap',graycolormap(10)) ;
  title('Test shape')
  grayplot(FEx,-FEy,-FEthetaTest',axesflag = 2) ;
   printf('\nIteration %d of %d,  HJ attempt %d, HJiter %d, objective = %f, objectiveTest =
%f, volume = %f\n',...
    i,entire,HJa,HJiter,objective,objectiveTest,totvolTest) ;
```

We perform a finite element analysis and compute energy densities, depending on whether we have done a shape or a topological sensitivity analysis. We use these results to calculate the objective function of the test shape.

Lines 147-156:

```
  // OBJECTIVE FUNCTION MUST DECREASE (up to some tolerance 'allow')
  if modulo(i,ntop)==0 | (i==1 & ntop~=0) then
    allow = allow_top;
  else
    allow = allow_adv;
  end
  if i>=(entire*3/4) then
    allow = 0;  // After some iterations, we switch-off the 'allow' parameter,
                // in order to converge.
    ntop = 0;
  end
```

As we have said before, we allow the objective function to increase significantly more in the case of topological sensitivity, due to numerical difficulties to create “small” holes. Moreover, after some iterations, expecting that the major topological changes have already happened, we switch off this variable and the topological gradient steps, else it is natural to observe oscillations of the shape, or even an increase of the objective function over many iterations.

Lines 147-156:

```
if objectiveTest <= objective*(1+allow) then
  // The current design is OK: move on to the next iteration,
  // using the test versions as our new variables to work with
  i=i+1; // Move on to the next optimization iteration
  phi = phiTest ;
  FEtheta = FEthetaTest ;
  objective = objectiveTest ;
  if modulo(i,ntop)==0 | modulo(i-1,ntop)==0 | i==2 then
    if modulo(i,ntop)==0 then
      [FEAeueu,FEAeueucomp] = FEtop(FEtheta,KE,K,F,U);
      V = FEAeueu - lagV;
    else
      [lvlAeueu,FEAeueu] = FE(FEtheta,KE,K,F,U);
      V = lvlAeueu/(dx*dy) - lagV;
      V = regularize(phi,V,e2,K1,lagP);
    end
  else
     lvlAeueu = lvlAeueuTest ;
     FEAeueu =FEAeueuTest ;
     V = lvlAeueu/(dx*dy) - lagV;
     V = regularize(phi,V,e2,K1,lagP);
   end
  HJiter = min(10,max(floor(HJiter*1.1),HJiter+1)) ;
  objectivePlot($+1) = objective ;
  // Plot the new shape:
  clf()
  xset('colormap',graycolormap(10)) ;
  title('Evolving shape')
  grayplot(FEx,-FEy,-FEtheta',axesflag = 2) ;

  HJa = 1 ; // Reset the lvl-set "attempt" variable
  e3 = 1;
```

In case the objective function has decreased, we accept the test shape and we need to calculate the new velocity.
There are three cases: Either in the next step we will perform a topological sensitivity step (modulo(i,ntop)==0), or the last step was the first step or a topological sensitivity step (modulo(i-1,ntop)==0) and therefore we need to compute the velocity for the shape sensitivity step, or last we both had and will continue with a shape sensitivity step, therefore we have already all necessary information.

Lines 194-216:

```
 else
   i=i+1;
   if modulo(i,ntop)==0 | modulo(i-1,ntop)==0 then
     if modulo(i,ntop)==0 then
       [FEAeueu,FEAeueucomp] = FEtop(FEtheta,KE,K,F,U);
       V = FEAeueu - lagV;
     else
       [lvlAeueu,FEAeueu] = FE(FEtheta,KE,K,F,U);
       V = lvlAeueu/(dx*dy) - lagV;
       V = regularize(phi,V,e2,K1,lagP);
     end
   else
     // The current design is bad: try again, this time with fewer
     // time steps for the transport level set equation
     HJiter = floor(HJiter/2) ;
     if HJiter == 0 then
        HJiter = 1 ;
        e3 = e3/2;
     end
     HJa = HJa + 1 ; // Increment the lvl-set "attempt" variable
   end
 end
end
```

In case the objective function has not decreased, we again examine several cases. If a topological gradient step is to follow, we compute the corresponding gradient. We do same thing if we pass from a topological gradient to a shape gradient step. Finally, in case the last and the current iteration are both using the shape sensitivity analysis, the rejection of the objective function means either that we have moved too much or that we have been lying on a local minimum. Therefore, we just reduce the advection step.

Lines 218-234:

```
// Plot the final shape:
clf()
xset('colormap',graycolormap(10)) ;
title('Final shape')
grayplot(FEx,-FEy,-FEtheta',axesflag = 2) ;
scf(0);
filename='Finaldesign';
xs2jpg(0,filename);
printf('\n Export of Final Design\n') ;
// Plot the objective function:
clf()
xtitle('Convergence history','Iteration','Objective function')
plot((1:length(objectivePlot)) - 1,objectivePlot) ;
printf('\nOptimization finished\n') ;
```

Finally, we plot the final design and the convergence history.

Exact Volume Constraint

In case one wants to keep the total volume constant during the optimization process, then one needs to update the Lagrange multiplier for the volume at each step. The user should remember that lagV does not appear in the objective anymore, but it does in the velocity for the advection of the level-set.
In the case when the volume has reduced, one should decrease lagV, while if the volume has increased, one should increase lagV. Unfortunately, it is possible that in the beginning of the algorithm this procedure is very costly. The reason is that for each increase of lagV the algorithm solves the advection equation and since the initial choice of lagV is arbitrary in general, this can be repeated too many times until finding the proper value of lagV. Fortunately, this is not the case close to convergence, where almost all topological changes have taken place. Moreover, we suggest that the user chooses a fine enough mesh for such an algorithm, since a coarse mesh makes small topological changes become significant. Even with a fine mesh, we needed many iterations for the disconnected parts to completely disappear.
In the following, we give an explanation of the function we have used for the volume constraint.

```
// UPDATE OF THE LAGRANGE MULTIPLIER FOR THE VOLUME
// we update lagV so that the volume remains stable in each iteration

function [phiTest,lagVTest,VTest,dtTest] = lagVupdate(phi,V,dt,HJiter,totvolinit,lagV,eps)

 phiTest = solvelvlset(phi,V,dt,HJiter) ;
 FEthetaTest = FEdensity(phiTest,eps) ;
 FEthetaTest = passive(FEthetaTest) ;
 totvolTest = volume(FEthetaTest) ;
 lagVTest = lagV;
 energy = V+lagV;
 totvolTest1=totvolTest;
```

We start by advecting the current level-set function phi, under the current lagV and so under the current V and dt. We name the advected level-set function “phiTest”. We compute the volume of the new shape, “totvolTest”, and we name it “totvolTest1”. The update of the velocity V will depend just on the lagV. So, for simplifying the computations, since the energy term of V remains constant, we prefer to separate it and name it “energy”.

```
if totvolTest1<totvolinit then
   while totvolTest<totvolinit
     lagVmax = lagVTest;
     lagVTest = lagVTest-0.1;
     VTest = energy-lagVTest;
     dtTest = 0.5*min(dx,dy)/max(abs(VTest)) ;
     phiTest = solvelvlset(phi,VTest,dtTest,HJiter) ;
     FEthetaTest = FEdensity(phiTest,eps) ;
     FEthetaTest = passive(FEthetaTest) ;
     totvolTest = volume(FEthetaTest) ;
     lagVmin = lagVTest;
   end
 end
```

If the volume has decreased, then we have to decrease lagV. So, every value that we will try will be lower than the one we have used and therefore we name the current value “lagVmax”. Then we decrease it by subtracting an arbitrary value and compute the new velocity “VTest”, which of course results in a new time step “dtTest”. We solve the advection equation under the new quantities and compute the volume. We name this new value of lagV as “lagVmin”.
We continue this procedure until we have found a value of lagVmin such that the volume has exceeded the initial one. This means that now, we have two values “lagVmax” and “lagVmin” and the desired value of lagV, i.e. the one that keeps the volume constant, lies between them.

```
if totvolTest1>totvolinit then
   while totvolTest>totvolinit
     lagVmin = lagVTest;
     lagVTest = lagVTest+0.1;
     VTest = energy-lagVTest;
     dtTest = 0.5*min(dx,dy)/max(abs(VTest)) ;
     phiTest = solvelvlset(phi,VTest,dtTest,HJiter) ;
     FEthetaTest = FEdensity(phiTest,eps) ;
     FEthetaTest = passive(FEthetaTest) ;
     totvolTest = volume(FEthetaTest) ;
     lagVmax = lagVTest;
   end
 end
```

We follow exactly the same logic in case the volume has increased to obtain the two values “lagVmax” and “lagVmin”.

```
if totvolTest1 == totvolinit then
   lagVmin = lagVTest;
   lagVmax = lagVTest;
end
```

Obviously, if the volume has not changed, the two values “lagVmin” and “lagVmax” coincide.

```
//Dichotomy on the value of the Lagrange multiplier
    while ((abs(1.-totvolTest/totvolinit))>.01)
      lagVTest = (lagVmin+lagVmax)/2. ;
      VTest = energy-lagVTest;
      dtTest = 0.5*min(dx,dy)/max(abs(VTest)) ;
      phiTest = solvelvlset(phi,VTest,dtTest,HJiter) ;
      FEthetaTest = FEdensity(phiTest,eps) ;
      FEthetaTest = passive(FEthetaTest) ;
      totvolTest = volume(FEthetaTest) ;
      if totvolTest < totvolinit then
        lagVmax = lagVTest;
      end
      if totvolTest > totvolinit then
        lagVmin = lagVTest;
      end
    end
```

Finally, we try to choose the value for lagV does not change the volume more than 1%. Of course, this value is arbitrary, but we suggest that one does not pick a too small value, since numerical problems will probably make the algorithm too slow. So, we take the average value of “lagVmax” and “lagVmin” and we check its impact. If the volume is lower than we desire, we set “lagVmax” equal to this average value and if the opposite happens, we do the same with “lagVmin”. We continue until our criterion is satisfied.

```
phiTest = phiTest ;
lagVTest = lagVTest;
VTest = energy-lagVTest ;
dtTest = dtTest ;

endfunction
```

Having determined the appropriate value of lagV and the corresponding level-set function, velocity and time step, we set them as the output of the function and continue the optimization algorithm.

References

[1] Allaire G, Conception optimale de structures, Mathematiques and applications, vol 58, Heidelberg:Springer;2006.

[2] Allaire G, Jouve F, Toader A-M, Structural optimization using sensitivity analysis and a level set method. J Comp. Phys. 2004;194/1:363-93.

[3] Allaire G, Jouve F, Toader A-M, A level set method for shape optimization. C R Acad Sci Paris Ser I 2002;334:1125-30.

[4] Allaire G, Pantz O, Structural optimization with FreeFem++.

[5] Dousset C., Allaire G. Calcul de formes optimales, Rapport de Stage, (2005).

[6] Karrman A., Allaire G., Structural optimization using sensitivity analysis and a level-set method, in Scilab and Matlab, (2009).

[7] Murat F., Simon S., Etudes de problemes d’optimal design. Lecture Notes in Computer Science 41, Springer Verlag, Berlin, 1976, p.54-62.

[8] Osher S., Fedkiw R., Level set methods and dynamic implicit surfaces, Applied Mathematical Sciences, 153, Springer-Verlag, New York (2003).

[9] Scilab, a scientific software developed by INRIA and ENPC, freely downloadable at
http://www.scilab.org.

[10] Sethian J.A., Level-Set Methods and fast marching methods: evolving interfaces in computational geometry, fluid mechanics, computer vision and material science, Cambridge University Press (1999).

[11] Sigmund O., A 99 line topology optimization code in Matlab, Struct. Multidisc. Optim., Vol.21, pp.120-127 (2001).

[12] Sokolowski J, Zolesio JP, Introduction to shape optimization: shape sensitivity analysis. Springer series in computational mathematics, vol.10, Berlin: Springer; 1992.

[13] Allaire G., De Gournay F., Jouve F., Toader A-M, Structural Optimization using topological and shape sensitivity via a level-set method, Control and Cyb., 34:59-80, 2005.