UNIVERSITY OF PADOVA

Civil, Environmental and Architectural department DICEA

Master's degree in Mathematical Engineering

Topology Optimization in a solid mechanics problem

Andrea Gorgi, Badge 2010658
Gianmarco Boscolo, Badge 2029060

Academic year 2021/2022

Contents

1 Introduction

This project is an extension and improvement of the one developed by Artem Mavliutov in [6]. In such work is explained how to construct an algorithm for the topology optimization of a given specif mechanical problem with the SIMP method (Solid Isotropic Material with Penalization method).
The goal of this paper is to update the algorithm to all the possible structural problems, providing simple ways to impose any set of boundary conditions or loads to the system. Moreover two new optimization gradient-based method have been explored, the Method of the Moving Asymptotes (MMA), which has a stronger mathematical background and should more reliable than the previously used Optimality Criteria (OC) (especially for problems with high numbers of degrees of freedom) and the Generalyzed Optimality Criteria.

In the end some possibilities for further future developments have been explored, like meshes with triangular or mixed type of elements, eventually loaded from an external mesh constructor.

1.1 Topology optimization, an overview

Layout optimization has applied with growing interest in the most diverse production environments in the last decades, with the desire to find the best layout of a domain for a specific problem maximizing or minimizing a chosen functional. Usually, layout optimization can be categorized into three types, i.e. size optimization, shape optimization and topology optimization. Those three types can be easily distinguished as demonstrated in (1), where the compliance of the cantilever is minimized respectively using the size optimization, shape optimization and topology optimization approaches.
In size optimization, the structure is parameterized using sizes and positions of geometrical features, and the corresponding parameters are chosen to be the optimization variables; the shape optimization, instead, parameterizes the geometrical characteristics of structures using the spline interpolation etc.,where the sample points are the optimization variables. After parameterization, the optimization variables are iteratively evolved in the predefined feasible regions, using global optimization or gradient information-based optimization algorithm. Size optimization and shape optimization are subjected to the inflexibility on changing the topology of the initial guesses of the structure, and thus are characterized by a strong dependence on the initial guesses of the optimization variables.

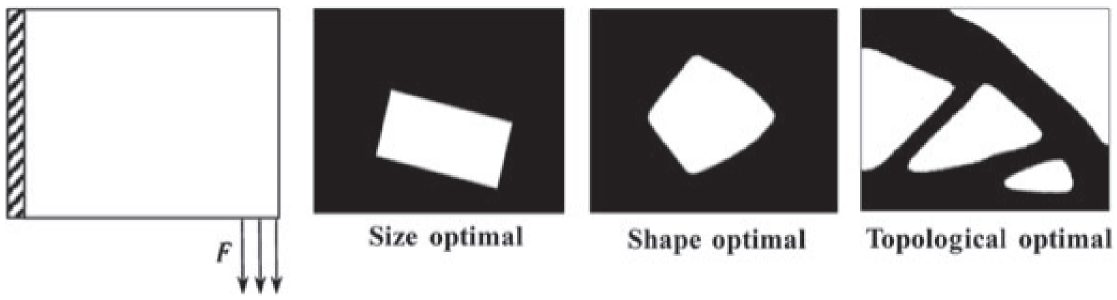


Fig.(1) Size optimization, shape optimization and topology optimization of a cantilever with minimizing compliance

The topology optimization on the other side is far less constrained by the guess on the initial configuration, and therefore should be a more powerful, flexible and robust tool for the structure optimization. It uses material distribution represented by the optimization variables. Since the material distribution is determined, topology optimization can fix the topology, shape and size of structures simultaneously.

Topology optimization arises from the structural optimization problem in elasticity and compliance mechanisms, but it has been extended to multiple physical problems, such as acoustics, electromagnetics, fluidics, optics and thermal dynamics.

Nowadays is widely used especially in engineering fields, but mostly at the concept level of a design process. In fact, due to the free forms that naturally occur, the result is often difficult to manufacture and needs to be fine-tuned for manufacturability. Adding constraints to the formulation in order to increase the manufacturability is an active field of research.

Usually topology optimization problems are solved with the finite element method (FEM). Then, the design is optimized using either gradient-based mathematical programming techniques such as the optimality criteria ([8]) algorithm and MMA or non gradient-based algorithms such as genetic algorithms.

2 Problem definition

The goal of this project is to adapt efficiently the work of Artem Mavliutov on an algorithm that allows to find the optimal structure or material distribution in a specific problem to a larger set of boundary and external loads conditions. As written in [6] this material distribution problem consist of discretizing the domain into finite elements. At each element coorresponds a parameter called material density (or design variable, later called γ) that determines if this element is filled with the material or not. In principle such variable should only take discrete values 0 or 1 (void or material), but solving the problem within the continuous range $[0, 1]$ allows the use of faster numerical techniques, like the OC or MMA cited above. Of course for the final solution to make sense at the end the algorithm should provide a γ with values at each element as close as possible to 0 or 1. We use 2- dimensional elements with four nodes, each one with 3 degrees of freedom.

2.1 Parameters and Variables Index

List and definition of principal quantities of the topology optimization problem:

- $J(\mathbf{u}, \gamma)$: problem functional;
- $\mathbf{u}(\gamma)$: displacement;
- γ : design variable;
- V_0 : domain volume;
- $V_{r_{min}}$: minimal admissible volume fraction;
- $V_{r_{max}}$: maximal admissible volume fraction;
- K : stiffness matrix;

- $\mathbf{f}$: forcing term or external load vector.

2.2 Optimization problem

We can now state explicitly our optimal design task as continuous constrained non-linear optimization problem

$$\begin{cases} \min\limits_{\gamma} J(\mathbf{u}(\gamma),\gamma) & & \\ \text{subject to} & : V(\gamma) \leq V_0 V_{rmax} & \text{Maximum Volume constraint,} \\ \text{subject to} & : V(\gamma) \leq V_0 V_{rmin} & \text{Minimum Volume constraint,} \\ & : K\mathbf{u} = \mathbf{f}, & \text{Governing equations,} \\ & : 0 \leq \gamma \leq 1, & \text{Design variable bounds} \end{cases} \tag{2.1}$$

Using the standard notation for an optimization problem, J will also be referred as the objective function, γ and $\mathbf{u}$ as the the decision variables and the equalities or inequalities that the system has to satisfy as the constraints. In this case J is the chosen functional to minimize (for maximization is sufficient to substitute the original J with $-J$). The two volume constraint are two additional constraints useful from both a physical and mathematical point of view. The maximal volume constraint fixes the maximum amount of material that can be used, and prevent the algorithm from finding trivial solutions, like completely filling the optimization region with solid medium. The minimal volume constraint instead, calibrated in a proper way, will help the method to exclude solutions with too few material and too high stresses in the elements.
Most of the tools for solving continuous optimization problem are based on the gradient of the functional to minimize with respect to the optimization variables, and so is the chosen Method of Moving Asymptotes. From eq.(2.1) is easy to see that once a value for γ has been fixed, it's possible to solve the governing equations for $\mathbf{u}$. Hence, $\mathbf{u}[\gamma]$ defines an implicit function for γ.
If we try to compute the gradient of J we would have in general

$$\frac{d}{d\gamma}[J(\mathbf{u}[\gamma],\gamma)] = \frac{\partial J}{\partial \gamma} + \frac{\partial J}{\partial \mathbf{u}}\frac{\partial \mathbf{u}}{\partial \gamma}. \tag{2.2}$$

Since $\mathbf{u}[\gamma]$is implicit, it's not so simple to evaluate the derivative $\frac{\partial \mathbf{u}}{\partial \gamma}$ directly. Two possible general ways to deal with calculation are explored in the following chapter. The developed optimization algorithm is then:

1. Choose an initial value for γ (initial material configuration),
2. Solve Governing equations ($K\mathbf{u} = \mathbf{f}$) for the nodal displacements $\mathbf{u}$),
3. Compute derivative of objective function and constraints with respect to gamma:
   (a) Evaluate the implicit derivative $\frac{\partial \mathbf{u}}{\partial \gamma}$ with the direct or adjoint method (next chapter)
   (b) Compute the gradient of the objective function
4. Filter the gradient to avoid the checkboard effect [2]
5. Use MMA or OC to update the value of γ to the value that minimizes J based on past iteration history and gradient informations,
6. Check convergence, if no, go back to step (2), if yes end the process.

We remark that the most expensive step of the process in terms of computational time is point 2 given that we would have to solve each time a system of non linear partial differential equations.

2.3 Implicit derivative: direct and adjoint method

The main idea to compute the implicit derivative $\frac{\partial \mathbf{u}}{\partial \gamma}$ is to exploit the constraint $K\mathbf{u} = \mathbf{f}$ of the governing equations.
Taking its variation with respect to γ we have in fact [3]

$$\frac{\partial K}{\partial \gamma}\mathbf{u} + K\frac{\partial \mathbf{u}}{\partial \gamma} = \frac{\partial \mathbf{f}}{\partial \gamma}, \tag{2.3}$$

from which

$$\frac{\partial \mathbf{u}}{\partial \gamma} = K^{-1}\left[\frac{\partial \mathbf{f}}{\partial \gamma} - \frac{\partial K}{\partial \gamma}\mathbf{u}\right]. \tag{2.4}$$

The gradient of the functional becomes then (direct method)

$$\frac{d}{d\gamma}[J(\mathbf{u}[\gamma], \gamma)] = \frac{\partial J}{\partial \gamma} + \frac{\partial J}{\partial \mathbf{u}}K^{-1}\left[\frac{\partial \mathbf{f}}{\partial \gamma} - \frac{\partial K}{\partial \gamma}\mathbf{u}\right]. \tag{2.5}$$

We note here that if the objective functions depends on more than one decision variable, this computation must be developed for each one of those. The adjoint method instead works through the Lagrangian built using both the objective functional and the governing equation constraint,

$$L(\mathbf{u}, \gamma, \lambda) = J(\mathbf{u}, \gamma) - \lambda^T [K\mathbf{u} - \mathbf{f}]. \tag{2.6}$$

Any possible minimal solution should then satisfy

$$\frac{\partial J}{\partial \mathbf{u}} = \lambda^T K. \tag{2.7}$$

We can now insert it in (2.5) to obtain the adjoint method

$$\frac{d}{d\gamma}[J(\mathbf{u}[\gamma], \gamma)] = \frac{\partial J}{\partial \gamma} + \lambda^T\left[\frac{\partial \mathbf{f}}{\partial \gamma} - \frac{\partial K}{\partial \gamma}\mathbf{u}\right]. \tag{2.8}$$

With this approach the computations does not depend anymore from the number of decision variables, but just on the objective functional (look at (2.7)), i.e. must be repeated only for different functionals. This means that the convenience of one or the other approach relies on the characteristic of the specific optimization problem, or rather the direct method is preferable when the number of decision variables is less than the number of funtionals, whereas the adjoint one is optimal when the number of functionals to minimize is less than the number of decision variables. In our case γ is to be thought as obtained by the linear interpolation of the elemental values

$$\gamma(x, y) = \sum_{iel=1}^{N} \phi_{iel}(x, y)\gamma_{iel}, \tag{2.9}$$

where $\phi_{iel}(x, y) = 1$ if point $(x, y) \in \mathit{Element}(iel)$ and zero otherwise. Thus we really have N decision variables in the system, one for each element, and the adjoint method would be the preferable one. Nonetheless, our choice for the objective functional as the elastic potential energy (see next section) allows the computation of the implicit derivative to be straightforward even with the direct method.

3 The objective functional

The most common choice for the functional in structure optimization problem is the compliance, which can be obtained multiplying the external forces by the displacement vector. Minimize it is the same as minimize the work generated by the forces in the structure or equivalently increase the stiffness of the structure. Hence we have

$$J(\mathbf{u},\gamma) = \mathbf{u}\cdot\mathbf{f} = \mathbf{u}\cdot K\mathbf{u} = \sum_{iel=1}^{N} \gamma_{iel}^{p}\mathbf{u}_{iel}\cdot\mathbf{k}_{iel}\mathbf{u}_{iel}, \tag{3.1}$$

where $\mathbf{u}_{iel}$ are the nodal displacements of the iel element vertices. Here $p$is just a penalty term that helps the algorithm to converge to discrete values for the γ_{iel} densities, since values $0 < \gamma_{iel} < 1$ are pushed closer to zero, and given that

$$\mathbf{k}_{iel} = \gamma_{iel}^{p}\mathbf{k}_{iel}^{ideal}, \tag{3.2}$$

where $\mathbf{k}_{iel}^{ideal}$ is just the standard element stiffness matrix, an element will contribute to the total stiffness of the structure only if its density is enough close to one. Clearly this effect becomes more relevant the more we increase the value of p. It has been proven that a value $p \geq 3$ is sufficient for convergence.
With simple calculations, according to the direct method shown above, or following the work of Mavliutov [6], we obtain the following gradient for the chosen objective function

$$\frac{dJ}{d\gamma_{iel}} = -p\gamma_{iel}^{p-1}\mathbf{u}_{iel}\cdot\mathbf{k}_{iel}\mathbf{u}_{iel}. \tag{3.3}$$

This choice for the objective functional is by far the most used for topology optimization problems. Another interesting possibility could be to choose the functional as the material volume, adding the constraint of maximum stress in each element, in order to find the structure that with the smallest amount of the chosen material is capable of sustain the load without yields or break. This formulation however needs to add to the formulation one constraint for each element which check that the stress is lower than the yield one, and this clearly results in a huge increase in the computational time.

4 FEM analysis

4.1 Construct Stiffness Matrix

This section discuss the formulation of the Stiffness matrices of two basic displacement-based elements as it's developed in [1]. As said in the introduction we only worked by using rectangular elements to create the mesh, but we studied anyway the formulation of triangular elements for future possible developments of the code, in order to load general meshes from any eventual external mesh creator software.

Stress-Strain Relations:

Let's consider σ as the stresses' array, ε as the array of strains and $\mathbf{E}$ the constitutive matrix.
The Stress-Strain relations are stated as:

$$\textbf{Stresses} = \textbf{Constitutive Matrix} \quad \times \quad \textbf{Strain} \quad + \quad \textbf{Initial Stresses}$$

$$\sigma = \mathbf{E}\varepsilon + \sigma_0 = \mathbf{E}(\varepsilon + \varepsilon_0). \tag{4.1}$$

$$\begin{pmatrix}\sigma_x\\ \sigma_y\\ \tau_{xy}\end{pmatrix} = \begin{bmatrix}E_{11} & E_{12} & E_{13}\\ E_{21} & E_{22} & E_{23}\\ E_{31} & E_{32} & E_{33}\end{bmatrix}\begin{pmatrix}\varepsilon_x\\ \varepsilon_y\\ \gamma_{xy}\end{pmatrix} + \begin{pmatrix}\sigma_{x0}\\ \sigma_{y0}\\ \tau_{xy0}\end{pmatrix}.$$

The constitutive matrix $\mathbf{E}$ is a symmetric invertible matrix that can represent isotropic or anisotropic properties.

$$\mathbf{E} = \frac{E}{1-\nu^2}\begin{bmatrix}1 & \nu & 0\\ \nu & 1 & 0\\ 0 & 0 & \frac{1-\nu}{2}\end{bmatrix} \tag{4.2}$$

where ν is the Poisson's ratio.

Formulas for Element Matrices:

To formulate the element stiffness matrices a fundamental role is played by the principle of virtual displacements:

$$\int \delta\varepsilon \cdot \sigma dV = \int \delta\mathbf{u} \cdot \mathbf{F} dV + \int \delta\mathbf{u} \cdot \Phi dS \tag{4.3}$$

where $\mathbf{u}$ be the displacements, $\mathbf{F}$ are the volume body-forces and Φ the surface tractions.
Let's now consider a discretized domain and therefore introduce $\mathbf{N}$, the shape functions and $\mathbf{d}$ the nodal displacements. Therefore:

$$\mathbf{u} = \mathbf{N}\mathbf{d}.$$

Now since the **Strain-Displacement** relation is:

$$\varepsilon = \partial\mathbf{u}$$

where ∂ is a derivative operator, it follows:

$$\varepsilon = \mathbf{B}\mathbf{d}, \qquad \text{with } \mathbf{B} = \partial\mathbf{N}.$$

The matrix $\mathbf{B}$ is called the *Strain-Displacement matrix.*
Therefore considering virtual displacements we get:

$$\delta\mathbf{u}^T = \delta\mathbf{d}^T\mathbf{N}^T, \quad \text{and} \quad \delta\varepsilon^T = \delta\mathbf{d}^T\mathbf{B}^T.$$

Finally, let's apply the principle of virtual displacements substituting the new quantities:

$$\delta\mathbf{d} \cdot \Big((\int \mathbf{B}^T\mathbf{E}\mathbf{B} dV)\mathbf{d} - \int \mathbf{B}^T\mathbf{E}\varepsilon_0 dV + \int \mathbf{B}^T\sigma_0 dV - \int \mathbf{N}^T\mathbf{F} dV - \int \mathbf{N}^T\Phi dS \Big) = 0, \quad \forall\delta\mathbf{d} \in \Delta. \tag{4.4}$$

where Δ is the admissible region of virtual displacements.

Observation. *The* $\delta\mathbf{d}$ *and* $\mathbf{d}$ *vectors can be taken out of the integral signs since the do not depend on the coordinates.*

In order to rewrite all in a more compact form let's define the **Element Stiffness Matrix** for a generic element of thickness t:

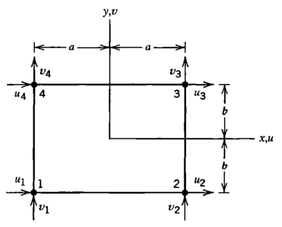
$$\mathbf{k} = \int \mathbf{B}^T\mathbf{E}\mathbf{B} dV = \int_0^t \int_A \mathbf{B}^T\mathbf{E}\mathbf{B} dA d\tau = \int \mathbf{B}^T\mathbf{E}\mathbf{B} dA \cdot t \tag{4.5}$$

and the **vector of loads applied** to structure nodes by elements:

$$\mathbf{r}_e = \int \mathbf{N}^T\mathbf{F} dV + \int \mathbf{N}^T\Phi dS + \int \mathbf{B}^T\mathbf{E}\varepsilon_0 dV - \int \mathbf{B}^T\sigma_0 dV.$$

Observation. *The initial code worked only with one rectangular mesh and the only possibility to work with more elements was to increase the size of the domain, therefore it wasn't possible to refine the mesh to get more more accurate solutions. At this scope we developed the possibility to work with any number of custom rectangular elements.*

Bilinear Rectangle (Q4):



Let's consider a linear rectangular element with nodal coordinates matrix:

$$\begin{bmatrix} N_1 \\ N_2 \\ N_3 \\ N_4 \end{bmatrix} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \end{bmatrix}.$$

The procedure developed in [1] yields to the following strain-displacement matrix for each rectangular element:

$$\mathbf{B} = \frac{1}{4ab}\begin{bmatrix} -(b-y) & 0 & (b-y) & 0 & (b+y) & 0 & -(b+y) & 0 \\ 0 & -(a-x) & 0 & -(a+x) & 0 & (a+x) & 0 & (a-x) \\ -(a-x) & -(b-y) & -(a+x) & (b-y) & (a+x) & (b+y) & (a-x) & -(b+y) \end{bmatrix}$$

where a is the horizontal semi-length and b is the vertical semi-length.
Since $\mathbf{B}$ is a function of the position to calculate the local stiffness matrix for each element must be performed a quadrature.
In order to do that it has been chosen to perform a Gaussian quadrature with two points $\xi_{1,2} = \pm\frac{1}{\sqrt{3}}$ and weights $w_{1,2} = 1$, since it's exact for polynomials of degree less or equal to $2n - 1 = 3$.
Let $f(x)$ be a polynomial of degree ≤ 3:

$$\int_{\alpha}^{\beta} f(x)dx = \frac{\beta-\alpha}{2}\sum_{i=1,2} w_i f(\frac{\beta-\alpha}{2}\xi_i + \frac{\beta+\alpha}{2})$$

Now, since $f = f(x,y)$ we can perform a double Gaussian quadrature imposing $\beta = a = -\alpha$ for the *x-integration* and $\beta = b = -\alpha$ for the *y-integration*, obtaining:

$$\int_{-b}^{b}\int_{-a}^{a} f(x,y)dxdy = ab\sum_{i,j=1,2} f(a\xi_i, b\xi_j)$$

In the following is attached part of the code used to compute the local stiffness matrix for each rectangular element.

```
%---------------------------------------------------------------------------
%% RECTANGULAR LOCAL STIFFNESS
%------------------------------
syms a; syms b;
syms x; syms y;
syms E; syms nu;
assume(a,'real'); assume(b,'real');
assume(x,'real'); assume(y,'real');
assume(E,'real'); assume(nu,'real');

B(1,:)   = [-b+y, 0      , b-y , 0      , b+y, 0     , -b-y, 0];
% Bt(:,1) = [-(b-y), 0      , (b-y) , 0      , (b+y), 0     , -(b+y), 0];
B(2,:)   = [0      , -a+x, 0      , -a-x, 0     , a+x, 0      , a-x];
% Bt(:,2) = [0      , -(a-x), 0      , -(a+x), 0     , (a+x), 0      , (a-x)];
B(3,:)   = [-a+x, -b+y, -a-x, b-y , a+x, b+y, a-x , -b-y;];
% Bt(:,3) = [-(a-x), -(b-y), -(a+x), (b-y) , (a+x), (b+y), (a-x) , -(b+y);];
B   = 1/(4*a*b)*B;
Bt  = B';

matE = E/(1-nu^2)*[1 , nu, 0;
                   nu, 1 , 0;
```

```
                    0 , 0 , (1-nu)/2;];
matrix = Bt*matE*B;

[diffValVect,diffValMat] = diffValues(matrix);

for i = 1:length(diffValVect)
    vectV(i,1) = diffValVect{i};
end
vect(x,y) = vectV;

%quadrature procedure
intVect = intMatrixComps(vect,a,b,2);
intVect(a,b,E,nu) = intVect;
intVect(a,b,E,nu) = expand(intVect);

% Since some of the components of intVect are equal but MATLAB doesn't
% recognize it due to their different expression it has been performed a
% posteriori handmade simplification in order to have each different value
% only once. These values has been stored in stiffVect.
% Since the diffValues has stored the indices of intVect it must be create
% a new relative idices vector: newRelVect.

c(a,b,E,nu) = E/(24*a*b*(1-nu^2));
stiffVect(a,b,E,nu) = [ 4*(2*b^2+a^2*(1-nu)); ...
                        3*a*b*(1+nu); ...
                        2*(a^2*(1-nu)-4*b^2); ...
                        3*a*b*(3*nu-1); ...
                        2*(a^2*(nu-1)-2*b^2); ...
                        3*a*b*(-nu-1); ...
                        4*(b^2-a^2*(1-nu)); ...
                        3*a*b*(1-3*nu); ...
                        4*(b^2+a^2*(2-nu)); ...
                        4*(a^2-b^2*(1-nu)); ...
                        2*(b^2*(nu-1)-2*a^2); ...
                        2*(b^2*(1-nu)-4*a^2); ...
                        4*(2*a^2+b^2*(1-nu));];
stiffVect(a,b,E,nu) = c*stiffVect;
newRelVect = [1,2,3,4,5,6,7,8,9,8,10,6,11,4,12,1,6,7,4,2,13,8,12,2,1,2,3,4,13,8,10,1,6,13];

% construction of the new indices matrix
indexMatrix = zeros(8,8);
for i = 1:8
    for j = 1:8
        indexMatrix(i,j) = newRelVect(diffValMat{i,j});
    end
end
```

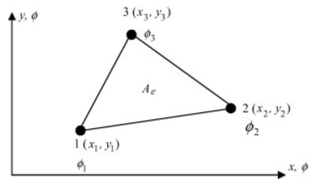
```
%-------------------------------------------------------------------------
%% DIFFERENT VALUES SELECTION
%---------------------------
function [vect, mat] = diffValues(M)
    sizeM = size(M);
        syms vect;
        syms mat;
    mat = {};
    vect = {};
    for i = 1:sizeM(1)
        for j = 1:sizeM(2)
            currVal = M(i,j);
            isMemb = 0;
            for membId = 1:length(vect)
                if currVal == vect{membId}
                    isMemb = 1;
                    mat{i,j} = membId;
                    break
                end
            end
            if isMemb == 0
                vect{end+1} = currVal;
                mat{i,j} = length(vect);
            end
        end
    end
end
```

```
%--------------------------------------------------------------------------
%% INTEGRATE MATRIX COMPONENTS
%-----------------------------
function [intA] = intMatrixComps(A,a,b,deg)
    if deg == 1
        ipX = [0];
        ipY = [0];
        w = 2;
    elseif deg == 2
        ipX = [a/sqrt(3), -a/sqrt(3)];
        ipY = [b/sqrt(3), -b/sqrt(3)];
        w = 1;
    end

    sizeA = size(A);
    intA = zeros(sizeA(1), sizeA(2));

    for i = 1:deg
        for j = 1:deg
            intA = intA + A(ipX(j),ipY(i));
        end
    end
    intA = w^2*a*b*intA;
end
```

```
%-------------------------------------------------------------------
%% BILINEAR RECTANGLE (Q4) LOCAL STIFFNESS MATRIX
%-------------------------------------------------------------------
function [Kloc] = localStiffness(this,E,nu)
    a = this.a;
    b = this.b;

    k = [   4*(2*b^2+a^2*(1-nu)); ...
            3*a*b*(1+nu); ...
            2*(a^2*(1-nu)-4*b^2); ...
            3*a*b*(3*nu-1); ...
            2*(a^2*(nu-1)-2*b^2); ...
            3*a*b*(-nu-1); ...
            4*(b^2-a^2*(1-nu)); ...
            3*a*b*(1-3*nu); ...
            4*(2*a^2+b^2*(1-nu)); ...
            4*(a^2-b^2*(1-nu)); ...
            2*(b^2*(nu-1)-2*a^2); ...
            2*(b^2*(1-nu)-4*a^2);];

    Kloc = E/(24*a*b*(1-nu^2))*[k(1),  k(2),  k(3),  k(4),  k(5),  k(6),  k(7),  k(8);
                                k(2),  k(9),  k(8), k(10),  k(6), k(11),  k(4), k(12);
                                k(3),  k(8),  k(1),  k(6),  k(7),  k(4),  k(5),  k(2);
                                k(4), k(10),  k(6),  k(9),  k(8), k(12),  k(2), k(11);
                                k(5),  k(6),  k(7),  k(8),  k(1),  k(2),  k(3),  k(4);
                                k(6), k(11),  k(4), k(12),  k(2),  k(9),  k(8), k(10);
                                k(7),  k(4),  k(5),  k(2),  k(3),  k(8),  k(1),  k(6);
                                k(8), k(12),  k(2), k(11),  k(4), k(10),  k(6),  k(9);];
end
```

Observation. *Despite the developed code works in its totality just with rectangular elements it has been performed the computation of the local stiffness matrix also for triangular elements in order to provide some ideas for possible future developments of the project.*

Constant-strain Triangle (CST):



Let's consider a linear triangular element with nodal coordinates matrix:

$$\begin{bmatrix} N_1 \\ N_2 \\ N_3 \end{bmatrix} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \end{bmatrix}.$$

Let's now define $x_{ij} := x_i - x_j$ and $y_{ij} := y_i - y_j$. Therefore the area of and element can be calculated as

$$A = \frac{x_{21}y_{31} - x_{31}y_{21}}{2}.$$

The procedure developed in [1] yields to the following strain-displacement matrix for each linear triangular element:

$$\mathbf{B} = \frac{1}{2A} \begin{bmatrix} y_{23} & 0 & y_{31} & 0 & y_{12} & 0 \\ 0 & x_{32} & 0 & x_{13} & 0 & x_{21} \\ x_{32} & y_{23} & x_{13} & y_{31} & x_{21} & y_{12} \end{bmatrix}$$

Now since both **B** and **E** are constant for each element the local stiffness matrix can be calculated as:

$$\mathbf{k} = \int \mathbf{B}^T \mathbf{E} \mathbf{B} dV = \mathbf{B}^T \mathbf{E} \mathbf{B} \cdot A \cdot t.$$

In the following is attached part of the code used to compute the local stiffness matrix for each triangular element.

```
%------------------------------------------------------------------------------
%% TRIANGULAR LOCAL STIFFNESS MATRIX
%--------------------------------------
syms x21; syms x31; syms x32;
syms y21; syms y31; syms y32;
syms E;
syms nu;

assume(x21,'real'); assume(x31,'real'); assume(x32,'real');
assume(y21,'real'); assume(y31,'real'); assume(y32,'real');
assume(E,'real');
assume(nu,'real');

A2 = (x21*y31-x31*y21);
AreaEl = A2/2;
constB = 1/(A2);
B = [-y32,    0,  y31,    0,  -y21,    0; ...
        0,  x32,    0, -x31,     0,  x21; ...
      x32, -y32, -x31,  y31,   x21, -y21;];


constE = E/(1-nu^2);
matE = [ 1, nu,        0; ...
        nu,  1,        0; ...
         0,  0, (1-nu)/2;];

sM = B'*matE*B;
[val,idMat] = diffValues(sM);

for i = 1:length(val)
    vectV(i,1) = val{i};
end
vect(x21,x31,x32,y21,y31,y32,E,nu) = vectV;

sM(x21,x31,x32,y21,y31,y32,E,nu) = sM;
stiffnessMatrix(x21,x31,x32,y21,y31,y32,E,nu) = AreaEl*constB*constE*constB*sM;
```

```
%------------------------------------------------------------------------------
%% DIFFERENT VALUES SELECTION
%----------------------------
function [vect, mat] = diffValues(M)
    sizeM = size(M);
        syms vect;
        syms mat;
    mat = {};
    vect = {};
    for i = 1:sizeM(1)
        for j = 1:sizeM(2)
            currVal = M(i,j);
            isMemb = 0;
            for membId = 1:length(vect)
                if currVal == vect{membId}
                    isMemb = 1;
                    mat{i,j} = membId;
                    break
                end
            end
            if isMemb == 0
                vect{end+1} = currVal;
                mat{i,j} = length(vect);
            end
        end
    end
end
```

Here is attached the **Triangle** class method to compute the local stiffness matrix for each element.

```
%-------------------------------------------------------------------
%% LINEAR TRIANGLE (CST) LOCAL STIFFNESS MATRIX
%-------------------------------------------------------------------
function [Kloc] = triangLocalStiffness(this,nodes,E,nu)

    %consider idn as the property of the element in which nodes id are
    %stored and ordered counterclockwise starting from the left-bottom one

    %x_ij = x_i - x_j;
    %y_ij = y_i - y_j;
    x21 = nodes(idn(2),1)-nodes(idn(1),1); x31 = nodes(idn(3),1)-nodes(idn(1),1); x32 = ...
        nodes(idn(3),1)-nodes(idn(2),1);
    y21 = nodes(idn(2),2)-nodes(idn(1),2); y31 = nodes(idn(3),2)-nodes(idn(1),2); y32 = ...
        nodes(idn(3),2)-nodes(idn(2),2);

    % Common factors of the matrix multiplication and integration
    AreaEl = abs((x21*y31-x31*y21))/2;
    constB(x21,x31,y21,y31) = 1/(2*AreaEl);
    constE = E/(1-nu^2);

    % Stiffness Matrix Values
    k =  [  (1/2 - nu/2)*x32^2 + y32^2; ...
            x32*y32*(nu/2 - 1/2) - nu*x32*y32; ...
            x31*x32*(nu/2 - 1/2) - y31*y32; ...
            nu*x31*y32 - x32*y31*(nu/2 - 1/2); ...
            y21*y32 - x21*x32*(nu/2 - 1/2); ...
            x32*y21*(nu/2 - 1/2) - nu*x21*y32; ...
            x32^2 + (1/2 - nu/2)*y32^2; ...
            nu*x32*y31 - x31*y32*(nu/2 - 1/2); ...
            y31*y32*(nu/2 - 1/2) - x31*x32; ...
            x21*y32*(nu/2 - 1/2) - nu*x32*y21; ...
            x21*x32 - y21*y32*(nu/2 - 1/2); ...
            (1/2 - nu/2)*x31^2 + y31^2; ...
            x31*y31*(nu/2 - 1/2) - nu*x31*y31; ...
            x21*x31*(nu/2 - 1/2) - y21*y31; ...
            nu*x21*y31 - x31*y21*(nu/2 - 1/2); ...
            x31^2 + (1/2 - nu/2)*y31^2; ...
            nu*x31*y21 - x21*y31*(nu/2 - 1/2); ...
            y21*y31*(nu/2 - 1/2) - x21*x31; ...
            (1/2 - nu/2)*x21^2 + y21^2; ...
            x21*y21*(nu/2 - 1/2) - nu*x21*y21; ...
            x21^2 + (1/2 - nu/2)*y21^2; ];

    % Stiffness Matrix
    Kloc = AreaEl*constB*constE*constB * [ k(1),  k(2),  k(3),  k(4),  k(5),  k(6);
                                           k(2),  k(7),  k(8),  k(9), k(10), k(11);
                                           k(3),  k(8), k(12), k(13), k(14), k(15);
                                           k(4),  k(9), k(13), k(16), k(17), k(18);
                                           k(5), k(10), k(14), k(17), k(19), k(20);
                                           k(6), k(11), k(15), k(18), k(20), k(21);];
end
```

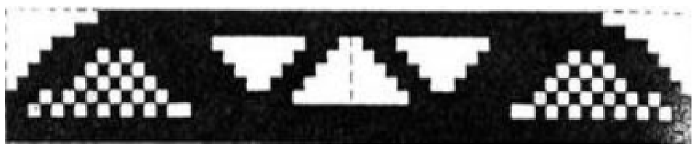
Observation. *In the previously attached code it is not considered the constant thickness of the elements t resulting from* (4.5).
For the sake of simplicity the t factor is multiplied in the stiffness matrix assembly procedure in the element class:

```
        %-----------------------------------------------------------------
        % STIFFNESS MATRIX ASSEMBLY
        %-------------------------
        function [K] = stiffness(this, prop)
            % ns : element nodes
            % prop: elements properties
            %...
            E = prop.young;
            nu = prop.nu;
            Kloc = prop.t*this.localStiffness(E, nu);
            K = Kloc;
        end
```

5 Filter and checkboard effect

5.1 Checkboard effect

Nowadays, many commercial softwares offer topology optimization algorithm to solve simple problems. When developing a new computer code, many computational and theoretical issues appear. The most common are: a) checkerboard patterns; b) mesh dependency; c) local minima; and d) singular topologies (for stress constrained problems) [2]. To overcome the second and third point is enough to solve the problem with mesh of different sizes and with different initial condition, or in the worst scenario using different optimization solvers (like MMA instead of OC). A checkerboard is defined as a periodic pattern of high and low values of Pseudo-densities, arranged in a fashion of checkerboards. This behaviour is undesirable as it is the result of a numerical instability and does not correspond to an optimal distribution of material. The checkerboards possess artificially high stiffness, and also such a configuration would be difficult to manufacture.



The checkerboard pattern in simply supported beam example [Sigmund and Petersson, 1998]

To avoid this checkboard effect in the solution many methods are possible:

- Use higher order polynomials in the FEM
- Perimeter control technique
- Patch technique
- The Poulsen scheme
- Filtering of sensitivities technique.

Of these we've chosen the lat. Al the others techniques are explored in detail in [2].

5.2 Filter

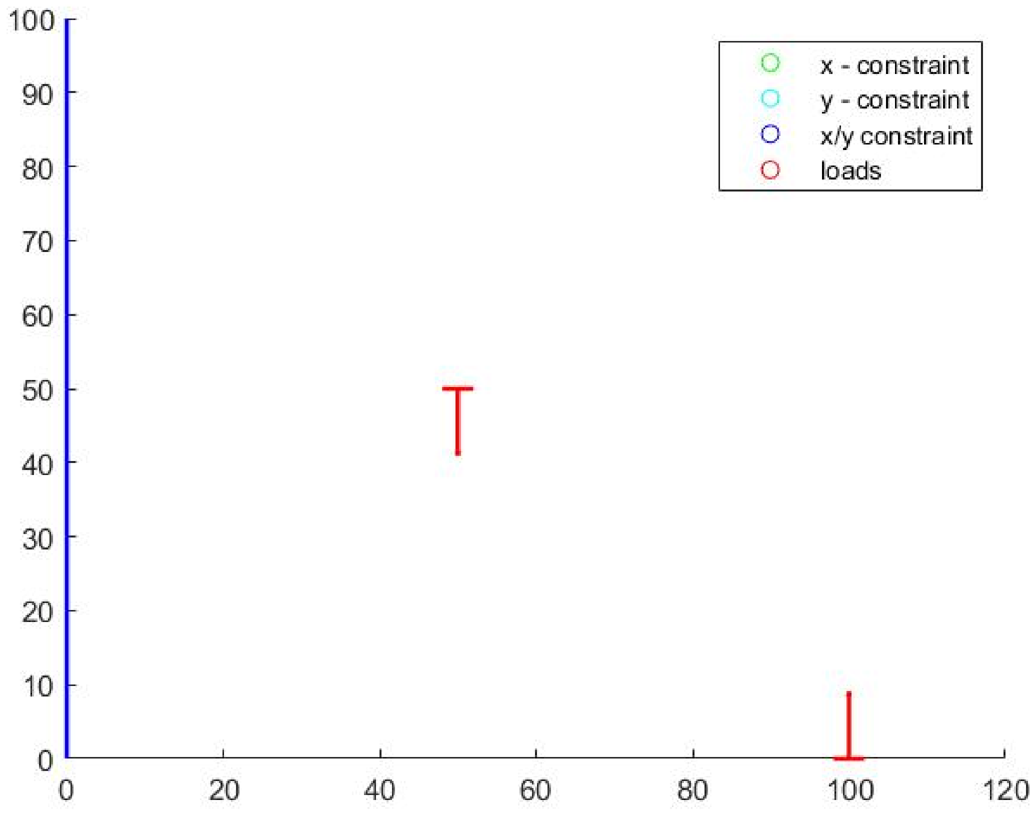
Filters are used to prevent checkerboarding by smoothening the stiffness in a fashion similar to the filtering of an image. With that the stiffness of an element depends on the densities of the neighbor elements. The method gives existence of solutions and convergence with refinement of the mesh. Filtering the sensitivity information of the optimization problem is moreover an efficient way to ensure mesh-independency. Filtering works by modifying the density sensitivity of a specific element based on weighted average of the element sensitivities in a fixed neighborhood. The scheme works by modifying the element sensitivity of the compliance as:

$$\frac{dJ}{d\gamma_{el}} = \frac{1}{\sum_{i=1}^{N} \gamma_i W_i} \sum_{i=1}^{N} W_i \gamma_i \frac{dJ}{d\gamma_i}, \tag{5.1}$$

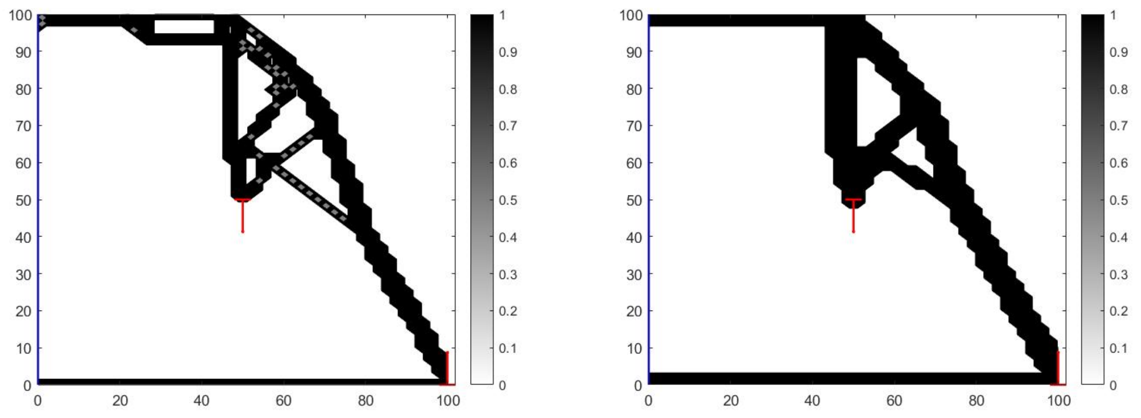
where W_i are just weighted factors of the values of γ in the mesh elements. We choose to have $W_i \neq 0$ only for the elements connected with the current one by at least one node, and fix $W_i = 1/4 * N_{common\ vertices}$. The implemented code for the filter (improved from the work in [6]) is reported here below (as method of the class TopOpt.m).

```
function [dcn]=check(this,gamma,dc)
         %-----------------------------
         % gamma = current vector of elemental densities
         % dc    = current compliance gradient
         %-----------------------------
         Nelem = length(this.Elements);
         dcn = zeros(Nelem,1); % filtered compliance gradient
         for iel = 1 : Nelem
             el = this.Elements(iel);
             sum = 0.0;
             for iloc = 1 : length(el.idn)
                 inode = el.idn(iloc);
                 node  = this.Nodes(inode);
                 for elloc = 1 : length(node.ConnectedElId)
                     weigth = 1/4;
                     jel    = node.ConnectedElId(elloc);
                     dcn(iel) = dcn(iel) + weigth*gamma(jel)*dc(jel);
                     sum = sum + weigth;
                 end
             end
             dcn(iel) = dcn(iel)/(gamma(iel) * sum);
         end
         %--------------------------------
       end
```

We now show a comparison of the OC and MMA methods for a simple mechanical problem with and without the filter.



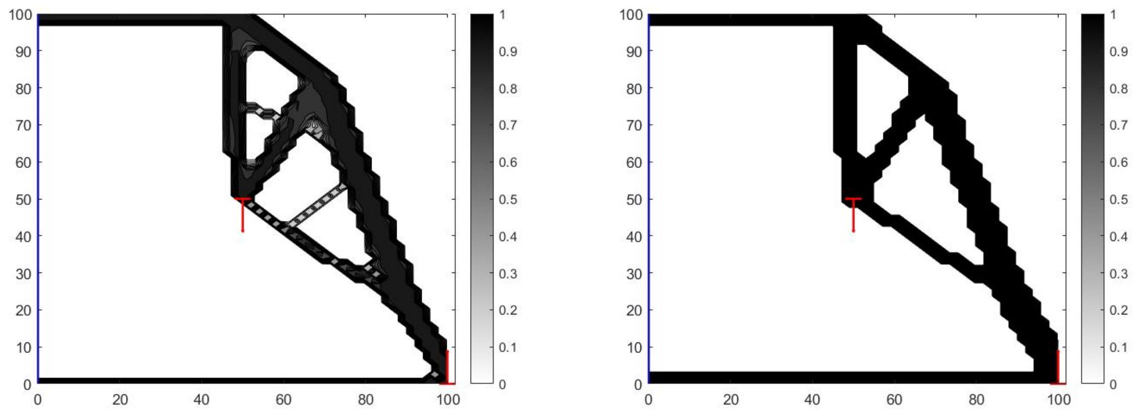
External conditions for the chosen optimization problem. The domain is 100×100 *millimeters with a mesh of* 50×60 *rectangles.*



OC final solution without (left panel) and with the filter (right panel). It's easy to see the influence of the checkboard effect when the filter is not active.

6 Optimization methods

In topology optimization problems three are the most used optimization algorithms: Optimality Criteria Method (OCM), Method of Moving Asymptotes (MMA), and Sequential Linear Programming (SLP).



MMA final solution without (left panel) and with the filter (right panel). Even in this case it's easy to see the influence of the checkboard effect when the filter is not active. Both loads have intensity of $10^3 N$.

These of all do not guarantee the best performance or the best final result, but are surely the most convenient. They are preferable to the "gradient-free" algorithms, like evolutionary or simulated annealing methods, since they would require a huge amount of functional evaluations, which is extremely costly in finite element analysis with high degrees of freedom, and to the algorithms working with Hessian information. Info on the Hessian of the functional can be useful to speed up the convergence of the schemes, but its computation is usually heavy and it would require a huge portion of memory to store it.
For this reason gradient algorithms like the three mentioned above are the most common choice for this problems.
The **Optimality Criteria** has a simple implementation that guarantees that the optimality criteria are met at each iteration of the scheme, and has the only limit is that it only works with minimizing compliance with the volume fraction constraint (with equality, not the two inequalities for minimal and maximal volume presented above). This is because the gradients for the compliance and volume fraction are “almost” free of computation.
The **Method of Moving Asymptotes** instead is a general-purpose algorithm that can support various types of optimization problems. It is based on convex approximation suitable for topology optimization, but its efficiency strongly depends on asymptote and move limits.
Finally, the **Sequentially Linear Programming** treats the nonlinear optimization by linearizing objective functions and constraints using the gradient informations. This simple algorithm tends to converge to the corner of move limits because those corners have the largest design change. The effort to remove those corners of move limits turns out to be the quadratic programming subproblem.
In this project we only made use of the OC and MMA methods, exploring a generalized and more efficient version of the OC.

6.1 Optimality Criteria (OC)

As said above, the Optimality Criteria method only works for the specif problem of optimizing the compliance with the volume constraint. For this reason we are going to derive it starting from the problem statement of the SIMP:

$$\begin{cases} \min\limits_{\gamma} J(\gamma) = \sum\limits_{iel=1}^{N} (x_e)^p \mathbf{u}_{iel} \cdot \mathbf{k}_{iel} \mathbf{u}_{iel} & \\ \text{subject to} & : V(\gamma) = V_0 V_r \quad \text{Volume constraint,} \\ & : K\mathbf{u} = \mathbf{f}, \quad \text{Governing equations,} \\ & : \gamma_{min} \leq \gamma \leq \gamma_{max}, \quad \text{Design variable bounds.} \end{cases} \tag{6.1}$$

Note how the two volume inequality constraints here are reduced to only one equality constraint. We can now convert this constrained optimization problem into an unconstrained one by defining the following Lagrange function (recalling that $\mathbf{u}$ it's fixed by the governing equations from the value of γ, so the functional doesn't really depend on it):

$$L(\gamma,\lambda) = J(\gamma) + \lambda(V(\gamma) - V_r V_0). \tag{6.2}$$

The Karush-Kuhn-Tucker first-order optimality conditions state that

$$\begin{cases} \dfrac{\partial L}{\partial \gamma} = \dfrac{\partial J}{\partial \gamma} + \lambda \dfrac{\partial V(\gamma)}{\partial \gamma} = 0 \\ \dfrac{\partial L}{\partial \lambda} = V(\gamma) - V_r V_0 = 0. \end{cases} \tag{6.3}$$

Since the Lagrange multiplier λ and the design variable γ_{iel} are coupled, the two equations must be solved simultaneously, but work on nonlinear equations is computationally expensive and difficult due to numerical instabilities. Thus the common solving procedure is composed of two-level loops. In the inner loop we update the design variable γ_{iel} to satisfy the first condition of (6.3) for a given value of the Lagrange multiplier λ. On the other hand the outer loop is used to update the value of λ in order to satisfy the volume constraint.
Once defined the element scale factor

$$D_{iel} = -\frac{\frac{\partial J}{\partial \gamma_{iel}}}{\lambda \frac{\partial V}{\partial \gamma_{iel}}}, \tag{6.4}$$

the inner-loop update rule for the γ_{iel} is

$$\gamma_{iel}^{new} = \gamma_{iel}^{old}\sqrt{D_{iel}},\ \gamma_{iel}^{min} \le \gamma_{iel}^{new} \le \gamma_{iel}^{max}. \tag{6.5}$$

One can easily verify that with such choice the first equation in (6.3) is verified. Just looking at such equation, we note how the optimality condition is satisfied when the objective sensitivity has same value and opposite sign of the constraint sensitivity times λ. Thus in an optimal configuration the ratio of the two quantities (D_{iel}) should be one. For this reason the design does not change when $D_{iel} = 1$ and the optimality condition is already satisfied. When $D_{iel} < 1$ increasing the material density γ_{iel} has a greater effect in increasing the volume than in decreasing the compliance, so its value is decreased. On the opposite when $D_{iel} > 1$ increasing γ_{iel} becomes convenient since it produces an higher improvement in the functional than in the volume. To prevent the scheme from instabilities due to hasty shifts we impose a maximum value for the change of each variable ($movel = 0.2$ in the code below).
Then, in the outer loop the Lagrange multiplier is updated to satisfy the volume constraint using a simple bisection method. Starting from some given lower and upper bound for γ_{iel}, like $[0, 10^6]$ in the code below, the range is halved at each iteration of the outer loop, and we choose the incumbent λ as the middle value of the range. The procedure is to evaluate the volume constraint, $V(\gamma) - V_r V_0$, then if the result is positive we keep the upper half of the range (greater values of λ results in smaller D_{iel} and probably smaller densities, hence smaller volume), otherwise the lower one, and repeat the routine until the range is smaller than a given tolerance.

```
%% OPTIMALITY CRITERIA UPDATE
function [gammanew]=OC(gamma,df0,df,f_func,isUniform,Vr)
%--------------------------------------------------------------
% df0: functional sensitivity
% df : current constraint sensitivity
% f_func: function to update constraint value
% isUniform: flag, "true" if mesh uniform
%--------------------------------------------------------------
  lower = 0; upper = 100000; maxmove = 0.2; tol = 1e-4;
  Nelem = length(df);
  while (upper-lower > tol)
    lmid = 0.5*(upper+lower);
    if isUniform
        gammanew = max( 0.001, ...
            max(gamma-maxmove,min(1.,min(gamma+maxmove,gamma.*sqrt(-df0 ./ ...
            (lmid*1/Nelem))))));
```

```
          f = sum(gammanew) - Vr*Nelem;
     else
         gammanew = max( 0.001, ...
             max(gamma-maxmove,min(1.,min(gamma+maxmove,gamma.*sqrt(-df0 ./ (lmid*df'))))));
         [f,¬] = f_func(gammanew);
          f = f(1);
     end
     if f > 0
       lower = lmid;
     else
       upper = lmid;
     end
   end
   %-----------------------------------------------------------------------
end
```

As said above, this method its very fast and of simple implementation but doesn't allow any modification to the compliance optimization with volume constraint problem. This is due to the fact that sensitivity of the volume constraint is constant and independent of design, whereas the compliance sensitivity can be easily computed once solved the governing equations of the structural problem. Moreover in (6.4) we assumed that the compliance sensitivity is always negative, and the volume constraint one is always positive. This is true for this specific case (just checking their formulations), but clearly depends in general from the problem formulation.
These concerns will be overcomed in the generalized version of the OC method, with also a different approach for the $\gamma - \lambda$ coupled problem than the bisection method, which can become expensive for large problems.

6.2 Method of moving asymptotes (MMA)

Th method of moving asymptotes, MMA, is an extremely efficient algorithm for constrained non linear optimization, developed by Krister Svamberg ([10]-[11]) based on a special type of convex approximation. Here we won't develop all the computations needed to completely define the method, but just some hints of its derivation. The code used in our schemes has been adapted from the MATLAB version reported in [11]. The schemes presented below are written for a general optimization problem, since we recall that, differently from the OC method, this approach is general and valid for various type of optimization problems, not just the one presented above.

6.3 General description of the method

Consider an optimization problem (in a standard form) :
P: minimize

$$f_0(\mathbf{x}) \quad (\mathbf{x} \in \mathscr{R}^N),$$

subject to

$$f_i(\mathbf{x}) \leq \hat{f}_i, \quad for\ i = 1, ..., M$$
$$\underline{x}_j \leq x_j \leq \bar{x}_j, \quad j = 1, ..., N,$$

where $\mathbf{x} = (x_1, ..., x_N)^T$ is the vector of design variable ($x_j = \gamma_j$ in our case), $f_0(\mathbf{x})$ is the objective function (J in our case), and $f_(\mathbf{x}) \leq \hat{f}_i$ are the constraints, with the additional lower and upper bounds for each variables $\underline{x}_j$ and $\bar{x}_j$. In short, the MMA scheme follow this iterative scheme:

1. Choose a starting point $\mathbf{x}^0$, and let the iteration index k = 0.

2. Given an iteration point $\mathbf{x}^{(k)}$, compute $f_i^{(x^{(k)})}$ and the gradients $\nabla f_i(\mathbf{x}^{(k)})$ for $i = 1, ..., M$.
3. Generate a subproblem $P^{(k)}$ by replacing, in P, the (usually implicit) functions f_i by approximating explicit functions $f_i^{(k)}$, based on the calculations from step 2.
4. Solve $P^{(k)}$ and let the optimal solution of this subproblem be the next iteration point $x^{(k+1)}$. Let $k = k + 1$ and go to step 2.

The algorithm stops when the chosen convergence criteria is fulfilled (in our case is a test on the solutions difference between consecutive iterations). Substantially, each $f_i^{(k)}$ is obtained by a linearization of f_i in variables of the type $1/(x_j - L_j)$ or $(U_j - x_j)$ dependent on the sings of the derivatives of f_i at $\mathbf{x}^{(k)}$. L_j and U_j are usually called "moving asymptotes" and are free to change between iterations.

Some insights of the method

Looking at above schematization, is clear that to define the method we must describe how the functions $f_i^{(k)}$ are defined and how each subproblem $p^{(k)}$ should be solved. We will only analyze the first question, just giving the brief idea for the second.
Assuming to have chosen $L_k^{(k)}$ and $U_j^{(k)}$ fot the current iteration k, such that $L_j^{(k)} < x_j^{(k)} < U_j^{(k)}$, for each $i = 0, ..., M$, we define $f_i^{(k)}$ as

$$f_i^{(k)}(\mathbf{x}) = r_i^{(k)} + \sum_{j=1}^{N} \left(\frac{p_{ij}^{(k)}}{U_j^{(k)} - x_j} + \frac{q_{ij}^{(k)}}{x_j - L_j^{(k)}} \right), \tag{6.6}$$

where

$$p_{ij}^{(k)} = \begin{cases} (U_j^{(k)} - x_j^{(k)})^2 \partial f_i/\partial x_j, & if\ \partial f_i/\partial x_j > 0 \\ 0, & if\ \partial f_i/\partial x_j \le 0 \end{cases} \tag{6.7}$$

$$q_{ij}^{(k)} = \begin{cases} 0, & if\ \partial f_i/\partial x_j \ge 0 \\ -(x_j^{(k)} - L_j^{(k)})^2 \partial f_i/\partial x_j, & \partial f_i/\partial x_j < 0 \end{cases} \tag{6.8}$$

$$\mathbf{r}_i^{(k)} = f_i(\mathbf{x}^{(k)}) - \sum_{j=1}^{N} \left(\frac{p_{ij}^{(k)}}{U_j^{(k)} - x_j(k)} + \frac{q_{ij}^{(k)}}{x_j(k) - L_j^{(k)}} \right) \tag{6.9}$$

and all derivatives $\partial f_i/\partial x_j$ are evaluated at $\mathbf{x} = \mathbf{x}^{(k)}$. With this formulation, $f_i^{(k)}$ is a first order approximation of f_i at $\mathbf{x}^{(k)}$,

$$f_i^{(k)}(\mathbf{x}^{(k)}) = f_i(\mathbf{x}^{(k)}) \quad \text{and} \quad \partial f_i^{(k)}/\partial x_j = \partial f_i/\partial x_j\ at\ \mathbf{x} = \mathbf{x}^{(k)}, \quad for\ i = 0, ..., M,\ j = 1, ..., N.$$

It can be easily checked that since $p_{ij}^{(k)}$, $q_{ij}^{(k)} \ge 0$, $f_i^{(k)}$ is a convex function, with second derivatives at $\mathbf{x} = \mathbf{x}^{(k)}$

$$\frac{\partial^2 f_i^{(k)}}{\partial x_j^2} = \begin{cases} \dfrac{2\partial f_i/\partial x_j}{U_j^{(k)} - x_j^{(k)}}, & if\ \partial f_i/\partial x_j > 0 \\ -\dfrac{2\partial f_i/\partial x_j}{x_j^{(k)} - L_j^{(k)}}, & if\ \partial f_i/\partial x : j < 0. \end{cases} \tag{6.10}$$

In particular, this means that the closer $L_j^{(k)}$ and $U_j^{(k)}$ are chosen to $x_j^{(k)}$, the larger will become the second derivatives, and therefore the more "curved" will be those functions and the more conservative will become the approximation of the original problem (since the approximation will be valid only close to the points $\mathbf{x}^{(k)}$). On the opposite, when $L_j^{(k)}$ and $U_j^{(k)}$ are chosen far from $\mathbf{x}^k$ then the approximating functions become more linear.

Assuming the lower and upper bounds $\underline{x}_j$ and $\bar{x}_j$ are phisically reasonable, as simple choice for $L_j^{(k)}$ and $U_j^{(k)}$ is

$$L_j^{(k)} = x_j - s_0(\bar{x}_j - \underline{x}_j), \quad and \quad U_j^{(k)} = \bar{x}_j + s_0(\bar{x}_j - \underline{x}_j), \tag{6.11}$$

7with a fixed real number $s_0 = 0 - 1$. The problem of this formulation is that the aymptotes would be fixed for each iteration, when it would be more clever to allow them to vary according to some reasonable rules.
An heuristical idea would be to, accordingly to the feature explained above,

- move the asymptotes closer to the current iteration point when the process tends to oscillate and needs to be stabilized.
- move the asymptotes away from the current iteration point if the process is monotone and slow, and needs to be relaxed.

With such approximating functions, comes the $P^{(k)}$ subproblem,
$P^{(k)}$: minimize

$$\sum_{j=1}^{N} \left(\frac{p_{0j}^{(k)}}{U_j^{(k)} - x_j} + \frac{q_{0j}^{(k)}}{x_j - L_j^{(k)}} \right) + r_0^{(k)}$$

subject to

$$\sum_{j=1}^{N} \left(\frac{p_{ij}^{(k)}}{U_j^{(k)} - x_j} + \frac{q_{ij}^{(k)}}{x_j - L_j^{(k)}} \right) + r_i^{(k)} \leq \hat{f}_i, \quad for\ i = 1, ..., M$$
$$\underline{x}_j < x_j < \bar{x}_j, \quad for\ j = 1, ..., N.$$

From this formulation the solution is usually obtained by a primal-dual algorithm, applying the Karush Kuhn Tacker conditions to the dual and solving them for example with a Fletcher Reeves gradient method. More details are given in [10] and [11], where is explained the final algorithm used in our simulations, which is based on the same approach, but with some additional artificial variables used to prevent the code from stopping at the first infeasibility due to a bad chosen starting point $\mathbf{x}^{(0)}$.

6.4 Generalized Optimilaty Criteria (GOC)

The Generalized Optimality Criteria Method (GOCM) [7] for topology optimization extends the capability of the OCM to multiple inequality constraints, leading also to an improved computational efficiency. The general optimization problem solvable with this technique can be defined as

$$\begin{cases} \min\limits_{\gamma} J(\gamma) & \\ \text{subject to} & : g_i(\gamma) \leq 0,\ i = 1, ..., NC, \\ & : K\mathbf{u} = \mathbf{f}, \quad \text{Governing equations,} \\ & : \gamma_{min} \leq \gamma \leq \gamma_{max}, \quad \text{Design variable bounds,} \end{cases} \tag{6.12}$$

NC number of constraints. This formulation can clearly represent also greater-than-or-equal-to inequalities constraint (by multiplication of a factor -1) and multiple functionals by weighted sums.
The associated Lagrangian unconstrained optimization problem has now the form

$$minimize\ L(\gamma, \lambda, s) = J(\gamma) + \sum_{i=1}^{NC} \lambda_i (g_u(\gamma) + s_i^2), \tag{6.13}$$

with λ_i Lagrange multiplier associated to constraint g_i. The variables s_i, called "slack variables", are used to convert the inequalities constraint into equalities, and are not zero only when the constrain is satisfied (less than zero). The necessary conditions for optimality are then

$$\begin{aligned}&\frac{\partial J}{\partial \gamma} + \sum_{i=1}^{NC} \lambda_i \frac{\partial g_i}{\partial \gamma} = 0,\\ &g_i(\gamma) + s_i^2 = 0,\ i = 1, ..., NC\\ &\lambda_i s_i = 0,\ i = 1, ..., NC.\end{aligned} \tag{6.14}$$

The third set of equations represent the "complementary slackness conditions", which states that only active constraints need to be considered in the necessary conditions (if $s_i \neq 0$ the constraint is satisfied, and $\lambda_i = 0$ so that it doesn't influence the optimality).
Hence, the second and third sets of equations in (6.14) are satisfied by identifying active constraints. For this reason the GOC goal is only to solve the first part of (6.14). Here again we find a coupling between the Lagrange multipliers λ_i and the decision variables γ_{iel}. Unfortunately with multiple constraints the OCM approach of a double-loop for the λ associated to the unique constraint and the γ_{iel} is not really exploitable here, since we would require an additional loop for each additional constraint.
The GOCM overcomes this problem by relaxing the controls over the Lagrange multipliers, stating that the don't have to satisfy the equations in (6.14) at each iteration, but imposing an update routine which makes them satisfy them when the optimization is converged.
More details on the method and on the possible updating routines for the Lagrange multipliers and the design parameters are given in [7].
In this project we used

$$\lambda_i^{k+1} = \lambda_i^k \left[1 + p_0(g_i^k + \Delta g_i^k)\right] \tag{6.15}$$

as update rule for the Lagrange multipliers, with k iteration number of the optimization problem, Δg_i variation on the value of i_{th} constraint with respect to last iteration, and p_0 chosen update parameter (usually $p_0 = 0.5$ or 1). For the decision variables the idea is still to use a scale factor D_{iel} for the updating algorithm, but in this case redefined as

$$D_{iel} = -\frac{\left\langle \frac{\partial J}{\partial \gamma_{iel}} \right\rangle_- + \sum_{i=1}^{NC} \lambda_i \left\langle \frac{\partial g_i}{\partial \gamma_{iel}} \right\rangle_-}{\left\langle \frac{\partial J}{\partial \gamma_{iel}} \right\rangle_+ + \sum_{i=1}^{NC} \lambda_i \left\langle \frac{\partial g_i}{\partial \gamma_{iel}} \right\rangle_+} \tag{6.16}$$

where $\langle x \rangle_- = min(0, a)$ and $\langle x \rangle_+ = max(0, a)$. When $D_{iel} = 1$, this formula will also satisfy the stationary condition of the Lagrange function.
In the theory of Lagrange multiplier, it is well known that it is zero when the constraint is not active, while it is positive when the constraint is active. That means, it is unnecessary to consider the Lagrange multiplier for inactive constraints and only active constraints are considered in the optimization. However, from a practical point of view, it is difficult to turn on and turn off constraints during optimization. Therefore, in the implementation, all constraints and Lagrange multipliers are retained. Then a constraint is inactive, the corresponding Lagrange multiplier will converge to its lower bound, which reduces its effect on the optimality criteria.

```matlab
function [lambda,gold,xnew] = GOC(Nelem,lambda,g,dg,gold,x,dc)
%-----------------------------------
eps = 0.05; maxmove = 0.2;
Δg = g - gold; gold = g;
xnew = zeros(length(x),1);
%--------------------
for i = 1 : length(g)
    if (g(i) > 0 && Δg(i) > 0) || (g(i) < 0 && Δg(i) < 0), p0 = 1.0;
    elseif (g(i) > 0 && Δg(i) > -eps) || (g(i) < 0 && Δg(i) < eps), p0 = 0.5;
    else, p0 = 0;
    end
    lambda(i) = lambda(i)*(1 + p0*(g(i) + Δg(i)));
end
%--------------------
for iel = 1: Nelem
    numerator   = 0;
    denominator = 0;
    if dc(iel) < 0
        numerator = dc(iel);
    else
        denominator = dc(iel);
    end
    for i = 1 : length(g)
        if dg(i,iel) < 0
            numerator = numerator + lambda(i)*dg(i,iel);
        else
            denominator = denominator + lambda(i)*dg(i,iel);
        end
    end
    De = - numerator/denominator;
    xnew(iel) = max(0.001,max(x(iel) - maxmove,min(1.,min(x(iel) + ...
        maxmove,x(iel).*sqrt(De)))));
end
%------------------------------------
end
```

7 Mesh creation

7.1 From Matlab

All the explored test problems have been studied with a MATLAB [5] built-in mesh. Using the function "meshgrid" from the MATLAB library is possible to create a uniform mesh of points in the chosen region. We hence simply used such command and from that defined each vertex as "Node" and from them rectangular elements and all the needed parameters (see "Implementation notes" section below).

7.2 Imported: Comsol

In order to provide more ideas for further improvements of the project here are presented some lines of code developed to introduce the possibility of solving the solid mechanics problem also for imported 2D-mesh, in our case from the commercial software COMSOL [4].
Once the geometry has been created, and then meshed, it's possible to export it in .txt file format from the *export* section of COMSOL.
In our case the only entities required are the nodes coordinates and the elements matrix, in which the i-th row contains the indices of the nodes that compose the i-th element, but more information can be exported by simply selecting them and appropriately modifying the following code.

```
%--------------------------------------------------------------------------
%% INITIALIZE PROBLEM
% NODES AND ELEMENTS CONSTRUCTORS
%--------------------------------------------------------------------------
meshdir = strcat('mesh_COMSOL\', problem, '\', problem, '.txt');
[nodesCoord, elements] = TopOpt.readFromComsolMesh(meshdir);


% meshType = "matlab";
meshType = "comsol";

Problem = TopOpt(nelx, nely, Lx, Ly, V_0, Vr, problem, meshType);

if meshType == "matlab"
    Problem = Problem.buildMesh();
```

```
elseif meshType == "comsol"
    Problem = Problem.comsolMesh(nodesCoord, elements);
end
```

Here is attached the **TopOpt** class methods used to import the mesh from COMSOL:

```
%-------------------------------------------------------------------
%% USE COMSOL MESH
%-----------------
function this = comsolMesh(this, nodesCoord, elements)
    % NODES CONSTRUCTOR
    this.Nodes = Node.empty(size(nodesCoord,1),0);
    for inode = 1 : size(nodesCoord,1)
   this.Nodes = this.Nodes';
    %-------------------
         this.Nodes(inode) = Node(inode, nodesCoord(inode,:));
    end
    % ELEMENTS CONSTRUCTOR
    nEl = size(elements,1);
    if size(elements,2) == 3
        this.Elements = Triangle.empty(nEl,0);
        for i = 1:nEl
            vertices = elements(i,:);
            coords = this.constructElCoords(vertices, nodesCoord);
            this.Elements(i) = Triangle(vertices, coords, 1);
            for iloc = 1:3
                inode = vertices(iloc);
                this.Nodes(inode).addElements(i);
            end
        end
    elseif size(elements,2) == 4
        this.Elements = ThinBeam.empty(nEl,0);
        for i = 1:nEl
            tempVertices = elements(i,:);
            coords = this.constructElCoords(tempVertices, nodesCoord);
            [vertices, a, b] = this.reorderRectEl(tempVertices, coords);
            this.Elements(i) = ThinBeam(vertices, 1, a, b);
            for iloc = 1:4
                inode = vertices(iloc);
                this.Nodes(inode).addElements(i);
            end
        end
    end
    this.Elements = this.Elements';
    %-------------------
    % PROBLEM DIMENSION
    this.dimProblem      = 0;
    for inode = 1:length(this.Nodes)
        this.dimProblem = this.dimProblem + this.NDOF;
    end
end

%-------------------------------------------------------------------
%% READ DATA FROM COMSOL MESH
%---------------------------
function [coord, elements, bound_edges, bound_edges_id] = readFromComsolMesh(meshdir)
    cellData = readcell(meshdir);
    nNodes = cellData{15,1};
    nEl = cellData{15+2+nNodes+5,1};
    count = 17;
    coord= zeros(nNodes,2);
    for i=1:nNodes
        count =count+1;
        for j=1:2
            coord(i,j) = cellData{count,j};
        end
    end
    count = count + 6;
    elements = zeros(nEl,3);
    for i=1:nEl
        count = count+1;
        for j=1:4
            elements(i,j) = cellData{count,j}+1;
        end
    end
end

%------------------------------------------------------------------
%% REORDER NODES ID COUNTERCLOCKWISE
%-----------------------------------
function [vertices, a, b] = reorderRectEl(idn, coords)
    x(1) = min(coords(:,1));
    y(1) = min(coords(:,2));
    x(2) = max(coords(:,1));
    y(2) = max(coords(:,2));
    a = (x(2)-x(1))/2;
    b = (y(2)-y(1))/2;
    vertices = zeros(4,1);
    for i = 1:4
        if abs(coords(i,1)-x(1)) < 1e-6
            if abs(coords(i,2)-y(1)) < 1e-6
                vertices(1) = idn(i);
            else
                vertices(4) = idn(i);
            end
        else
            if abs(coords(i,2)-y(1)) < 1e-6
                vertices(2) = idn(i);
            else
                vertices(3) = idn(i);
            end
        end
    end
end
```

Observation. *The "reorderRectEl" method is needed because the rectangular elements ordering of COMSOL is not the one required to perform the analysis.*

8 Implementation Notes

The codes have been implemented with an object oriented programming approach in MATLAB [5], and the simulation runs through a "main.m" script, which defines the specific problem settings and calls the useful classes methods. The classes implemented are

- **Node.m:** Coordinates and methods associated to each point of the discretized domain;
- **Property.m:** Defines Young modulus, Poisson coefficient, cross sectional area for each element in the grid;
- **RectBeam.m:** Defines nodes indices, property index and some geometrical properties for each rectangular element of the grid, plus some useful methods;
- **Constraint.m**: Defines coordinates and fixed dof of the specific constraint;
- **Load.m**: Defines coordinates and load intensity dof of the specific load;
- **TopOpt.m:** Defines all the useful parameters and methods to build or load the mesh for the problem, create the necessary nodes, properties and elements, and solve the mechanical problem;

- **MMA.m:** Defines all methods and properties to launch the MMA optimizer.

The OCM and GOCM have simpler algorithms than the MMA, so we simply used a MATLAB function for their implementation.

8.1 Main file

The first lines of the main file are used to fix the values for all the geometrical parameters of the problem, like x or y range of the domain, the values for the algorithm parameters, like the number of elements in the x or y direction in the case of the Matlab built-in mesh or the maximum number of iterations, and to initialize the "TopOpt" object with its mesh.

```
%%% MAIN SCRIPT FOR THE SOLUTION OF A TOPOLOGY OPTIMIZATION PROBLEM
% Problem statement
%----------------------------------------------
%%% minimize work done on the system
%%% P.      min C(x)  = \sum (x_e)^p * u(x_e)^T K_e u(x_e)
%%% s.t.    V(x)/V_0 ≤ Vr_max
%%%         V(x)/V_0 ≥ Vr_min;
%%%         0 < x_min < x_e ≤ 1
%----------------------------------------------
% Main parameters and variables
%------------------------------
% C(x)       : objective functional, work done on the system
% x          : design variable, density of material, x = [x_e], e = 1,...,Nelem
% u(x)       : displacement vector [mm]
% K          : global stiffness matrix
% K_e        : local stiffness matrix
% f          : vector of forces acting on the system
% V(x)       : material volume after optimization [mm^2]
% V_0        : domain total volume [mm^2]
% Vr_max     : maximum fraction of total volume to be occupied by material
% Vr_min     : minimal fraction of total volume to be occupied by material
% penal      : SIMP penalty coefficient (usually p = 3)
% Lx         : domain horizontal length [mm]
% Ly         : domain vertical length [mm]
%-----------------------------------------------------------
%% INITIALIZE PARAMETERS
close all; clear all;
% geometrical
%-------------
Lx  = 100;
Ly  = 100;
V_0 = Lx*Ly;
E   = 2.1100e+11;
nu  = 0.29;
Yield_S = 2.2e+8; %Pa

% algorithm parameters
%---------------------
nelx   = 50; nely = 60; Nelem = nelx*nely;
penal  = 3;
Vr_min = 0.04;
Vr_max = 0.1;
rmin   = 1.5;
Vmin   = 0.001;
maxIt  = 50;
%------------------------------------------------------------------------
%% INITIALIZE PROBLEM
%------------------------------------------------------------------------
Problem = TopOpt(nelx, nely, Lx, Ly, V_0, Vr_max, Vr_min);
Problem = Problem.buildMesh();
isUniform = true;
```

```
%-----------------------------
%% SET PROPERTY
%-----------------------------
Problem = Problem.createProp(1, E, nu);
%---------------------------------------------------------------------------
```

Then we add some constraints and loads to our problem (left side fixed, a concentrated load in in example below).

```
%% IMPOSE BC
% constraint = [fixX,fixY , xMin,xMax, yMin,yMax]
%-----------------
constraints(1,:) = [1,1, 0,0, 0,Ly];
sizeConstraints = size(constraints);
for i = 1:sizeConstraints
    fix = constraints(i,1:2);
    xrange = constraints(i,3:4);
    yrange = constraints(i,5:6);
    Problem = Problem.constraint(fix, xrange, yrange);
end
%---------------------------------------------------------------------------
%% IMPOSE EXTERNAL LOADS
% concentrated load = [coordX,coordY, xLoad, yLoad]
% distributed load  = [ [xmin, xmax], [ymin, ymax], xLoadDensity,
%                                                    yLoadDensity ]
%-------------------------

Problem = Problem.addConcLoad(Lx,0, 0,1e4);
%------
Problem = Problem.addDistrLoad([50,100],[100,100],[0,-500000/50]);
```

Finally we initialize some loop parameters (like the loop counter of iteration) and start the optimization cycle.

```
while (loop < maxIt) && ( change > 0.0001)
    %----------------------------------------
    %%% SOLVE THE CURRENT MECHANICAL PROBLEM
    %----------------------------------------
    loop = loop + 1;
    xold = gamma;
    f0_old = f0;
    if loop ≠ 1
        [¬, Problem]    = Problem.Solve(gamma);             % global displacement ...
            vector U
    end
    %----------------------------------------
    %%% EVALUATE FUNCTIONAL AND CONSTRAINTS
    %----------------------------------------
    [f0, df0, f, df, Vol] = Problem.evaluate(gamma);
    [df0]    = Problem.check(gamma,df0);
    %----------------------------------------
    %%% UPDATE DESIGN VARIABLE
    %----------------------------------------
    %1 % OPTIMALITY CRITERIA METHOD

%     [gamma]    = OC(gamma,df0, df(1,:), @(x) Problem.evalConstraints(x), isUniform, ...
    Vr_max);

    %2  MMA method
    [gamma]    = Optimizer.solve(f0, df0, f, df, gamma);

    %3 GOC METHOD
%     [lambda,gold,x] = GOC(Nelem,lambda,f,df,gold,x,df0);
    %----------------------------------------
    % PRINT RESULTS
    change = max(max(abs(gamma-xold)));
    disp([' It.: '  sprintf('%4i',loop) ' Obj.: ' sprintf('%10.4f',f0) ...
          ' Vol.: ' sprintf('%6.3f', Vol) ...
          'stress:' sprintf('%10.4f',sqrt(sum(sum(S.*gamma)))/10^6) ...
          ' ch.: '  sprintf('%6.3f',change ) ...
          'compl:'  sprintf('%10.4f',f0-f0_old) ])

      %  CHECK ON YIELD STRESS
      countVol = countVol +1;
      Smax = Problem.maxStress;
      SF = Smax / Yield_S^2;
      if loop > 5 && countVol > 2
      if SF < 0.7 && (Vr_max > Vmin)
        Vr_max = Vr_max - 0.02;
        if Vr_max < Vr_min
            Vr_min = Vr_max - 0.05;
        end
        countVol = 0;
      elseif SF > 0.8
        Vr_min = Vr_min + 0.02;
        if Vr_max-Vr_min < 5e-2
            Vr_max = Vr_max + 0.05;
        end
        countVol = 0;
      end
      end
    %
    figure(1)
    Problem.Plot(gamma);
    %-----------------------------------------------------------------------
end
```

The last part of the cycle allows the program to self-adapt to too high or low material constraints, depending on the maximum stress found in the structure.

9 Some results

We will here compare the three developed optimization processes (with OCM, MMA and GOCM) in three different structural problem.
For all of them we will considered an S275 steel ($E = 210000MPa$, $\nu = 0.3$) with thickness $t = 60mm$, and $Vr_{min} = 0.1$, $Vr_{max} = 0.2$ at the beginning.

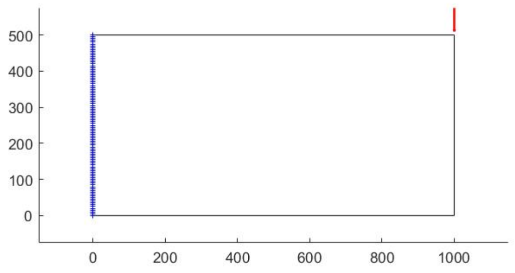
9.1 Example 1: Simple supported beam with concentrated load

We impose the following external conditions:

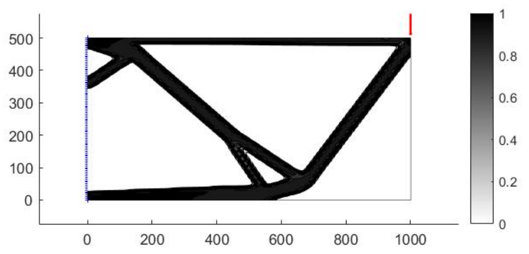
```
% geometrical
%-------------
Lx   = 500; %% mm
Ly   = 1000; %% mm
V_0 = Lx*Ly;
E    = 210000;
nu   = 0.3;
t    = 60; %% mm (elements width)
Yield_S = 275; %MPa

% algorithm parameters
```

```
%---------------------
nelx   = 100; nely = 100; Nelem = nelx*nely;
Vr_min = 0.1;
Vr_max = 0.2;
rmin   = 1.5;
Vmin   = 0.001;
maxIt  = 40;
```

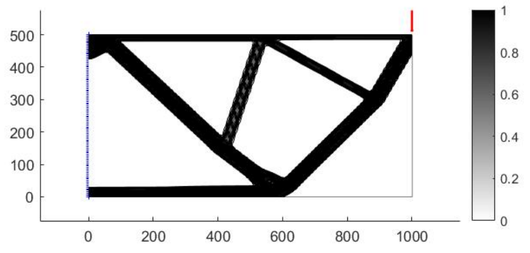
```
constraints(1,:) = [1,1, 0,0, 0,Ly];
Problem = Problem.addConcLoad([Lx,Ly], [0,-1e4]);
```



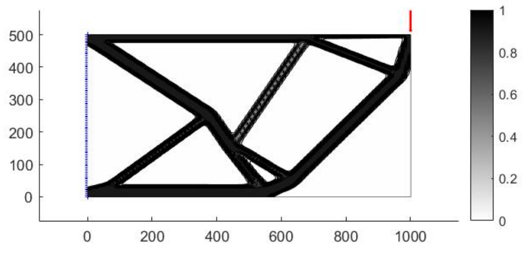
(a) *Geometry setting of example 1*



(b) *Final result with OC approach.*



(a) *Final result with GOC approach.*



(b) *Final result with MMA approach.*

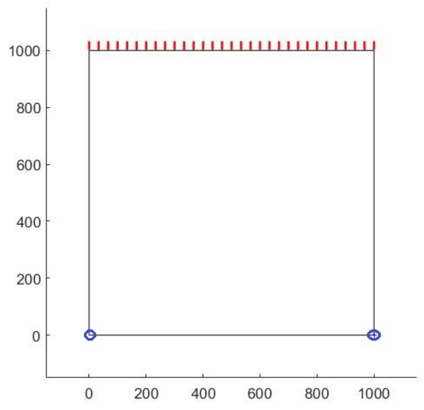
	OC	GOC	MMA
$obj[N \cdot mm]$:	$6.9e-3$	$7.5e-3$	$9.8e-3$
it :	40	40	40
$wall\ time[sec]$:	76.3	66.96	80.4
SF :	0.69	0.5	0.51
$Vr.$	0.2	0.195	0.2

In this SF is the squared ratio between the maximum stress found in the system and the material yield stress.

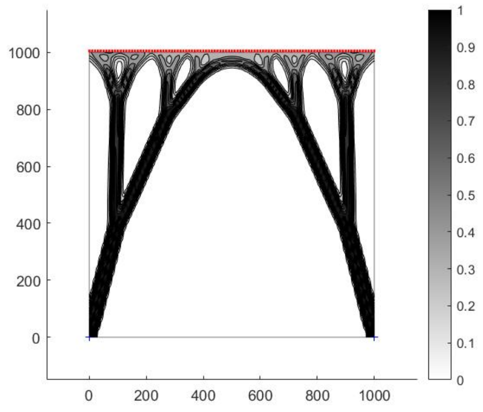
9.2 Example 2: Distributed load

```
% geometrical
%-------------
Lx   = 1000; %% mm
Ly   = 1000; %% mm
V_0  = Lx*Ly;
E    = 210000;
nu   = 0.3;
t    = 60; %% mm (elements width)
Yield_S = 275; %MPa

% algorithm parameters
%---------------------
nelx   = 100; nely = 100; Nelem = nelx*nely;
Vr_min = 0.1;
Vr_max = 0.2;
rmin   = 1.5;
Vmin   = 0.001;
maxIt  = 60;
```

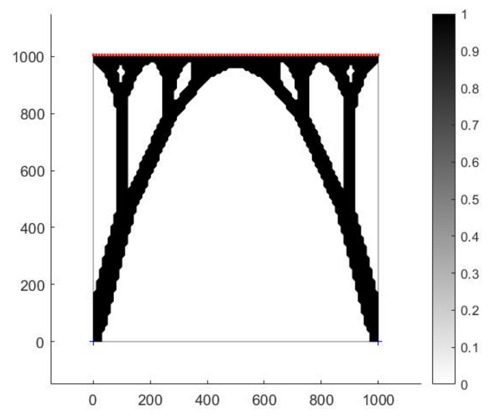
```
constraints(1,:) = [1,1, 0,0, 0,0];
constraints(2,:) = [1,1, Lx,Lx, 0,0];
Problem = Problem.addDistrLoad([0,Lx],[Ly,Ly],[0,-1e4/Lx]);
```

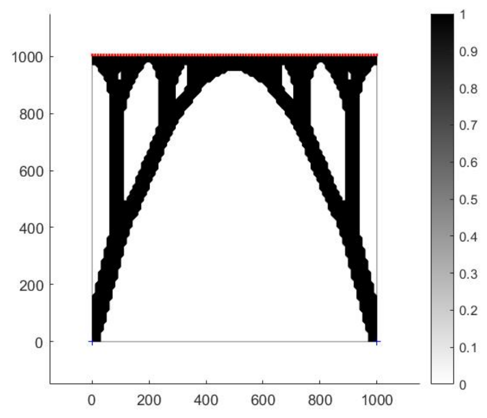


(a) *Geometry setting of example 2*



(b) *Final result with OC approach.*


(a) *Final result with GOC approach.*

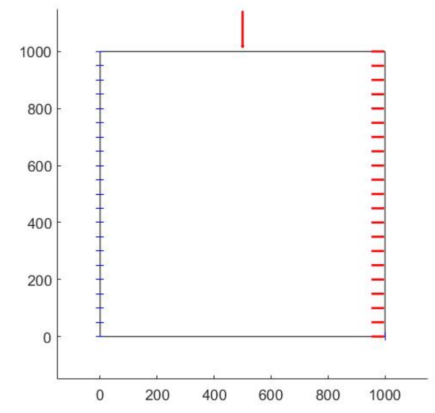

(b) *Final result with MMA approach.*

	OC	GOC	MMA
$obj[N \cdot mm]$:	$4.7e-2$	$3.6e-2$	$4.3e-2$
it :	60	60	60
$wall\ time[sec]$:	42.1	38.6	50.7
SF :	0.34	0.55	0.65
$Vr.$	0.13	0.18	0.2

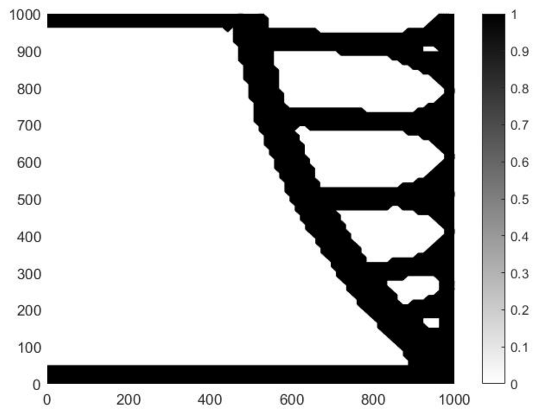
9.3 Example 3

```
% geometrical
%-------------
Lx   = 1000; %% mm
Ly   = 1000; %% mm
V_0  = Lx*Ly;
E    = 210000;
nu   = 0.3;
t    = 60; %% mm (elements width)
Yield_S = 275; %MPa

% algorithm parameters
%---------------------
nelx   = 80; nely = 80; Nelem = nelx*nely;
Vr_min = 0.1;
Vr_max = 0.2;
rmin   = 1.5;
Vmin   = 0.001;
maxIt  = 60;
```

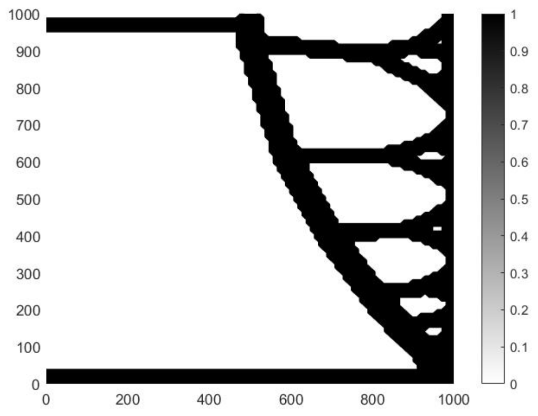
```
constraints(1,:) = [1,0, 0,0, 0,Ly];
constraints(2,:) = [0,1, Lx,Lx, 0,0];
Problem = Problem.addConcLoad([Lx/2,Ly], [0,-1e4]);
Problem = Problem.addDistrLoad([Lx,Lx],[0,Ly],[1e4/Ly, 0]);
```



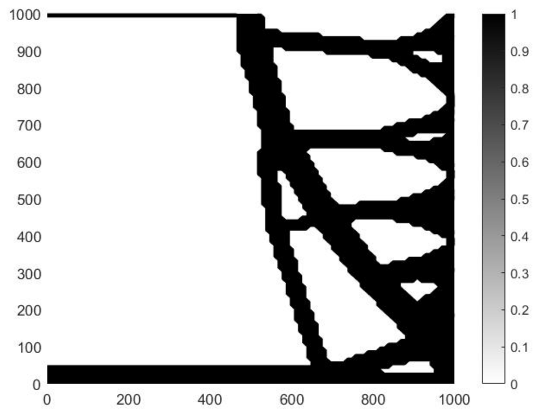
(a) *Geometry setting of example 3*



(b) *Final result with OC approach.*



(a) *Final result with GOC approach.*



(b) *Final result with MMA approach.*

	OC	GOC	MMA
$obj[N \cdot mm]$:	$3.7e-1$	$2.1e-1$	$8.0e-2$
it :	60	60	60
$wall\ time[sec]$:	57.8	42.8	72.7
SF :	0.39	0.74	0.5
$Vr.$	0.25	0.234	0.257

10 Conclusion and final considerations

The three examples explored in the previous section are clearly not enough to compare properly the efficiency of the three methods (we should take statistics over several trial with the same number of dofs and compare them), but in general give some ideas on the efficacy of these algorithms.
First we note how in each case all the schemes tend to converge to similar final configurations, and none of them seems to be the global best choice in terms of optimizing the objective function. In all the test

cases the GOCM has proven to be the fastest of the three methods, whereas the MMA is the slowest.
In some cases the optimization algorithms try to converge to non completely discrete solutions (figure(b) of second example). This practically means that in such areas where the density is lower than 1, the algorithms suggest that materials with a lower Young modulus would be just as effective as the provided one (recall that $E_{iel} = \gamma^{penal} E$).
Future interesting developments would be to generalize the implementation of the "reading" of external meshes, with triangles or rectangles, from an external software into the optimization code, or to slightly modify the optimization problem to admit two or more possible materials, and let the algorithm find which one is better use in each portion of the domain.

Bibliography

[1] R. D. Cook et al. *Concepts and Applications of Finite Element Analysis.* John Wiley Sons, Inc., 2002.

[2] Sunil Kumar Avinash Shukla Anadi Misra. “CHECKERBOARD PROBLEM IN FINITE ELEMENT BASED TOPOLOGY OPTIMIZATION”. In: (2013).

[3] M.E. Biancolini C. Groth A. Chiappa. “Shape optimization using structural adjoint and RBF mesh morphin”. In: (2017).

[4] *COMSOL Multiphysics®, v. 6.0. www.comsol.com. COMSOL AB, Stockholm, Sweden.*

[5] MATLAB. *version R2021b.* Natick, Massachusetts: The MathWorks Inc., 2021.

[6] Artem Mavliutov. “Advanced Solid Mechanics Topology optimization”. In: (2021).

[7] David Weinberg Nam H. Kim Ting Dong and Jonas Dalidd. “Generalized Optimality Criteria Method for Topology Optimization”. In: (2011).

[8] O.Sigmund. “A 99 line topology optimization code written in Matlab”. In: (2001).

[9] Rolf Rannacher. “Finite Element Methods for the Incompressible Navier-Stokes Equations”. In: (1999).

[10] Krister Svanberg. “The Method of Moving Asymptotes - A new Method for Structural Optimization”. In: (1987).

[11] Krister Svanberg. “The Method of Moving Asymptotes - Modelling aspects and solution schemes”. In: (1998).