Automatic Differentiation for both Explicit and Implicit Topology Optimization using Julia Language

ROBIN GRAPIN[1,2] **AND JOSEPH MORLIER**[1,3]

[1] *Institut Supérieur de l'Aéronautique et de l'Espace (ISAE-SUPAERO), Université de Toulouse, 31055 Toulouse, FRANCE*
[2] *Email : robingrapin@orange.fr*
[3] *Email : Joseph.MORLIER@isae-supaero.fr*

Compiled November 17, 2020

The topology optimization consists on determining the best shape of a part for a given problem. As the algorithms do require a lot of calculation, mathematical processes can allow to be more efficient, and then to be able to find the optimal shape of a whole structure. The recent method of the automatic differentiation made possible to calculate quickly and exactly the derivatives of functions. This paper presents the improvements caused by this new technique to the classic topology optimization methods.

1. INTRODUCTION

The origin of the topology optimisation is around 1960, year of the publication of Lucien Schmit's [1] work. He was the first to see the potential of the combination of finite-element analysis with optimization methods. The seminal idea, which consists of computing the optimal distribution in space of an anisotropic material that is constructed by introducing as many small holes as possible in a given material, appeared in 1988 [2]. This paper marked the beginning of encoded optimization methods. This fundamental article was based on the study of existing solutions, not depending from the original disposition. The first developed methods were based on an homogenization approach [3], which has been aborted because it often gave non pathwise connected results. Because they allowed to obtain buildable structures, the density [4] or level-set [5] methods became more popular.

As Matlab was one of the most popular language around 2000, the work of Bendsøe and Sigmund [6] allowed the whole scientific community and the students to use the optimization codes. They worked on the so-called "power-law". First, they proved that under perimeter constraints, gradient constraints or with filtering techniques, solutions do exist, as long as simple conditions on the power are satisfied (relations between the penalty power and the Poisson's ratio). Other filtering techniques were developed then to avoid weaknesses in the structure.

In 2001, the 99 line topology optimization code [7] written in Matlab has been published in open source, to help for the education of engineers. The related article details the function of every line. This code was then improved to a vectorized version [8], shorter and quicker, also using new methods of filtering : the density filter despite of the sensitivity filter. More recently, new upgrades have been made to apply it to three-dimension examples [9] and to make it even more versatile [10] and efficient [11].

In parallel, new optimizers emerged. The most used are the lagrangian multipliers and moving asymptotes [12] based methods. It participated to the amelioration of the performances of topology optimization algorithms. But to be always more accurate and quick in the resolution of problems, most of the algorithms of optimization are being adapted to use the automatic differentiation [13]. The programming language *Julia* is performing as it is close to machine and the modules of automatic differentiation take benefit from its compilation system, called just-in-time compilation, to make it more efficient.

2. PROBLEM STATEMENT

When structures are created, it is always necessary to study the shape of each part to use as few material as possible and in the same time to be able to resist to equivalent stresses. It leads to the sparing of material and in aeronautics, removing mass made possible better performance, but to ensure the reliability of the machine stays the priority.

The mathematical writing of a topology optimization problem is the following :

$$find\ min_x J(x)$$
$$subject\ to : V(x)/V_0 = f$$
$$KU = F$$
$$and\ 0 \leq x \leq 1$$

Where x is a matrix in which every cell contains the proportion of material in the corresponding element of the design space, 0 means that it is empty, 1 that it is full. $V(x)$ is the volume filled by the material and V_0 the volume of the whole design space. K is the matrix of stiffness, U the vector of displacements and F the vector of forces. About the objective function, it can vary in function on what we aim to do. The most usual example is :

$$the\ compliance : J(x) = U^T K U \sum_{element\ e} E_e(x_e) u_e^T k_0 u_e$$

But other objective functions are common, and their derivatives are not necessarily explicit. that is why automatic differentiation will be a useful tool. Other constraints (non linear) can be also added, like forcing a part to have a hole somewhere.

The control experiment will be the MBB beam problem with the compliance as objective function in two dimensions. Minimizing the compliance is equivalent to maximize the stiffness of the structure. It is usual to symmetrize the problem to only calculate a half of the solution. Some other examples will be treated like the cantilever or the L-shape beam. With the given formulation of the problem, since the the material's density is homogeneous, the mass of the piece will not be to optimize because of the equality constraint on the volume fraction.

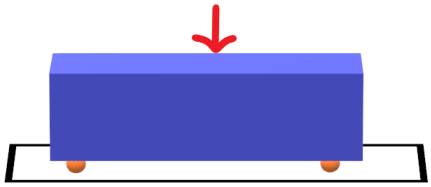

Fig. 1. MBB problem

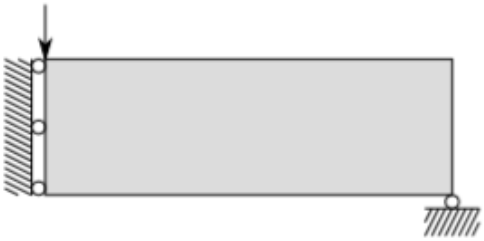

Fig. 2. Symmetrized MBB problem

To solve such a problem, the diagram is the following :

Algorithm 1. optimization algorithm

```
procedure MassRepartition(designspace, constraints, J)
    initialize x
    meshing
    calculate elementary matrix K_e
    create filter H
    while | J(x_i) − J(x_{i−1}) | > ϵ do        ▷ optimization loop
        x_{i+1} = optimize(J, x_i, constraints)
        filtering x_{i+1}
    return x
```

The optimizer called in the algorithm's loop needs the derivatives of J at the current value of x. When the formulae of $J'(x)$ is not explicit, usual methods approximate this value using the fact that

$$\lim_{||\epsilon||\to 0} \frac{J(x+\epsilon)-J(x)}{\epsilon} = J'(x)$$

The computing of these techniques has been well improved in speed and precision, but it still gives an approximation, which can be far from the reality in some examples. As the function to minimize is the sensitivity, the priority is the safety and approximation are not sufficient. Furthermore, the objective function is sometimes also approached to have a form easier to derive, as it is the case in the SIMP method. To introduce automatic differentiation, let us start with dual numbers. They are of the form $y = a + b \times \epsilon$, where $\epsilon^2 = 0$. If we use such a ϵ different from 0, the development of any derivable function f becomes :

$$f(y+\epsilon) = f(y) + \epsilon \times f'(y)$$

The important fact is, that this is an equality and not an approximation. That is the main idea of the process used to compute the exact derivative of J. The second key point is to look differently at the chain rule for composed derivatives. Every computed function is a succession of compositions of elementary operations. For example,

$$f : x, y \to 6 \times x + y$$

can be re-written as

$$f(x,y) = g(h(x,y))$$

where $h(x,y) = (6x, y)$ and $g(u,v) = u + v$. Then,

$$f(w_1) = g(h(w_1)) = g(w_2) = w_3$$

and what we are searching is $\frac{\partial f}{\partial x} = \frac{\partial w_3}{\partial w_1}$. But

$$\frac{\partial w_3}{\partial w_1} = \frac{\partial w_3}{\partial w_2}\frac{\partial w_2}{\partial w_1}\frac{\partial w_1}{\partial w_1} = \frac{\partial w_3}{\partial w_2}\frac{\partial w_2}{\partial w_1}$$

and the partial derivatives at each step are obtained thanks to the fact that they are done through elementary operations. It includes the binary arithmetic operations, the sign switch and usual applications such as the exponential, the logarithm, and the trigonometric functions. Different modes exist for automatic differentiation, and the easiest to understand is the *forward mode*, described on the example on the following array, but the *reverse accumulation mode* is better for functions having values in a much smaller space than the space of its variables.

Primal trace	Derivative trace
$w_{-1} = x = 2$	$\frac{\partial w_{-1}}{\partial x} = \frac{\partial x}{\partial x} = 1$
$w_0 = y = 5$	$\frac{\partial w_0}{\partial x} = \frac{\partial y}{\partial x} = 0$
$w_1 = 6 \times w_{-1} = 6 \times 1$	$\frac{\partial w_1}{\partial x} = 6 \times \frac{\partial w_{-1}}{\partial x} + \frac{\partial 6}{\partial x} \times x_{-1} = 6 \times 1 + 0 \times 2$
$w_2 = w_1 + w_0 = 12 + 5$	$\frac{\partial w_2}{\partial x} = \frac{\partial w_1}{\partial x} + \frac{\partial w_0}{\partial x} = 6 + 0$
$f(2,5) = w_2 = 17$	$\frac{\partial f}{\partial x}(2,5) = \frac{\partial w_2}{\partial x} = 6$

They are different points of view from which we can start to adapt the solving method. Two of them will be touched on in this paper. The first one is Solid Isotropic Material with Penalization (SIMP). It consists on adding mass where it is needed to make the part stronger and to remove mass where it is not needed, and then adding a penalty to the cells of the space having a density different from 1 or 0 to make it physic. The Young modulus is approximated by a power-law : $E(x) = E_{min} + x^p(E_0 - E_{min})$. The E_{min} value can be set to 0 but using a small minimal Young modulus avoid divisions by zero. Thanks to this form, the derivative of the stiffness is explicitly given by $\frac{\partial c_e}{\partial x_e} = -px^{p-1}(E_0 - E_{min})u_e^T k_0 u_e$. This method is well adapted to the fabrication of parts thanks to 3D-printers, because it leads to mass distributions with complicated structures, optimizing as much as possible the structure. This is the approach

chosen in the $top99, top88$ and $top88 - AD$ algorithms. $top88$ is a vectorized version of $top99$, using more efficiently the memory, but both codes have the same principle.

First, during the meshing, the design space is divided into square elements, which aim to contain material or not. The degrees of freedom are set and the density is initialized at f, the fraction of volume, for every cell. At each iteration, the finite element analysis is performed, the obtained density and sensitivity are filtered, and then the design variables are updated through an optimizer. The method of the optimal criteria is used to solve the mathematical problem, but if complicated constraints are effective, it is preferable to use more evolved method, such as MMA, as it has been implemented in the next method.

But to build such parts might be impossible because of their complicated structures, so the Generalized Geometry Projection (GGP) will be taken in consideration here. It consists on choosing a defined number of elementary parts, with easy to build shapes, and making them move, change of size and of inclination to form the most solid composition. The result is then an explicit design of the final part that is to built.

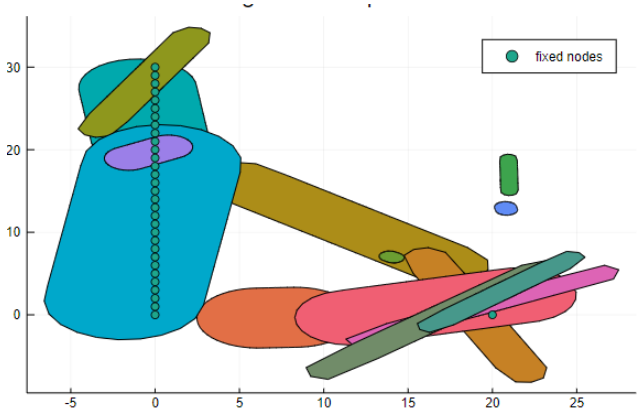

Fig. 3. GGP design

3. RESULTS AND DISCUSSION

Julia language offers an access to modules of automatic differentiation for free, like $ForwardDiff$. For this reason, the first step of this work was to encode existing optimization codes in Julia, like Ole Sigmund's codes top99 and top88. The algorithms we encoded in this new language worked as quick as the originals in Matlab. The outputs of both versions are a little bit different because of the approximations made by the native functions to invert large matrix of each language. They will be used as control experiment to compare the results obtained with the new method using automatic differentiation.

To use the $ForwardDiff$ package, the function to derive has to be unary. Knowing that the model taken in account to calculate the compliance is $J(x) = U^T KU$, the objective function is in fact a function U and x, which evolve at each iteration. The main idea is then to define the objective function as a function of U and x and to redefine inside of the loop the partial unary function of x with the current value of U fixed, and to apply the automatic differentiation to this function.

The evolutions brought by the algorithm using automatic differentiation are that it is now possible to use the algorithm on any objective function of the mass distribution x, the displacements U and the stiffness matrix K, without having to know explicitly its derivative. The results we obtained after testing the code on known problems thanks to the top88 code, as the control problem chosen, shows that the results are precise with 11 exact decimals on large examples. The used memory on the computer to run this algorithm is about 10 times bigger than if the derivative of the function is explicitly given and 5 times slower in average. It is also interesting to compare these results with those obtained with a method of approximation of the derivatives. The finite difference algorithm is about 6 times slower than the reference, and has "only" 7 exact decimals for the same problem.

About the speed, the algorithm top88 took, for the "relatively big" example 100x100 :
12.845634 seconds (156.44 M allocations: 6.358 GiB, 10.76 percent gc time)
against
58.238602 seconds (182.39 M allocations: 70.522 GiB, 13.31 percent gc time)
for top88 using AD and
66.370186 seconds (106.86 M allocations: 192.019 GiB, 9.99 percent gc time)
for top88 using finite differences (stopping both of them after 25 iteration as the stopping condition on the change isn't reached.). This demonstrates that finite difference is not necessarily quicker.

Fig. 4. top88 design

The second pass is the central point of this research : the use of the automatic differentiation for a GGP method. Once again, the tests and comparisons were done with the MBB problem with the sensitivity as objective function. The code written in *Julia* is based on Simone Coniglio's Generalized Geometry Projection in Matlab. The finite element analysis is using the Gaussian quadrature method. To store the elementary parts of the model, the technique used is the one of the centroid coordinates. The model updates at each iterations are then applied to these coordinates, moving their center and modiying their size and orientations. As the constraints are more complex to materialize moving parts in the design space, it was necessary to use a solver based on the method of moving asymptotes. Because the existing *Julia*'s MMA modules were not adapted to solve a problem such as GGP, we had to encode an new one. Whereas the SIMP density initialization is a design space with an equal density everywhere, the starting point of the GGP process is important to discuss. The choice made here is the *cross* design, which is an uniform distribution of the density if we consider large areas.

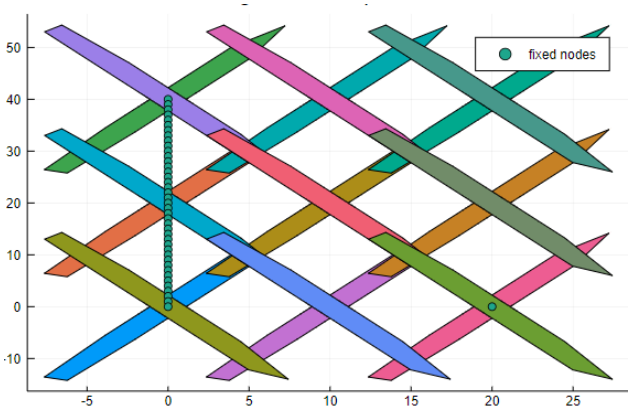

Fig. 5. Initial state

4. CONCLUSION AND FUTURE WORKS

The automatic differentiation's precision and efficiency has been demonstrated by the obtained results, and its benefit will be more evident on the GGP code, because the derivatives of the compliance over the 6 degrees of freedom are calculated in two steps :

$$\frac{\partial c}{\partial x_i} = \frac{\partial c}{\partial E}\frac{\partial E}{\partial x_i} \; 0 \leq i \leq 6$$

. That means that they are two potential errors because of finite differences approximations for each of them. Secondly, the constraints due to the use of the aggregation functions are complicated and leads to important errors. Using the automatic differentiation, larger problems such as modes shapes or vibration will be calculable, avoiding an augmentation of the errors due to finite differences methods or to hand derivation errors. The problems of modes shape and vibrations and problems about buckling need complicated models and are related to any part of a plane which could vibrate, as the example of the winglet. The shape of such parts could then be optimized thanks to this work.

Another comment about the algorithm's results is, that the GGP method is very depending on the initial condition and do not always converge to the global optimum, contrarily to the implicit methods.

An other development possibility is to use automatic differentiation also for the constraints functions, as they appear in the Karush-Kuhn-Tucker condition, used in most of the optimizers :

$$\frac{\partial f}{\partial x_k}(x*) + \sum_{1 \leq i \leq m} \lambda_i \frac{\partial g_i}{\partial x_k}(x*) = 0 \;\; 0 \leq k \leq n$$

5. ANNEX

All the encoded algorithms encoded to achieve the results and related on this paper are available at : https://github.com/mid2SUPAERO/TopoRobin

Each of the codes on the folder *TopoRobin* are touching on the MBB problem. Some other basic examples of use on different load cases are in the *examples* subfolder. To use and extend these algorithms for other problems, the following steps and modifications should be done.

A. top99

- Import the packages : $Plots, SparseArrays$;
- Modify the material properties E and nu ;
- Modify the applied stresses line 6 of the *top* function ;
- Modify the constraints on the fixed nodes line 9 of the *top* function ;
- Modify the ending criterion lines 18 and 37 ;
- An other possibility (also for top88 and its extensions) is to add the non-linear condition to have no material somewhere. To do so, passive elements must be defined before the optimization while creating an array of booleans of the size of the design space. If an element is defined as passive, at the end of each step of the optimization loop, this element's density must be set to zero. See the *examples* folder for such an example ;
- Run the function with the parameters $nelx, nely$ for the number of element along the x and y axis, $volfrac$ is the wished proportion of material in the design space of the part, the penalty $penal$ must be chosen in function of the Poisson coefficient nu, usually $penal = 3$ for nu $= 0.3$ and $rmin$ is the filter size divided by the element size ;
- Plot the resulting density distribution (output x) and the evolution of the objective function (output c) in function of the number of iterations (output l).

B. top88

- Import the packages : $Plots, SparseArrays, LinearAlgebra, Statistics, SuiteSparse$;
- Modify the material properties E0, Emin and nu ;
- Modify the applied stresses line 22 of the *top88* function ;
- Modify the constraints on the fixed nodes line 24 of the *top88* function ;
- Modify the ending criterion lines 54 and 104 ;
- In addition to *top99* parameters, ft allows to chose between the sensitivity filter ($ft = 1$) or the density filter ($ft = 2$).

C. top88-AD

In addition to top88 steps, it is necessary to :

- Import $ForwardDiff$ package ;
- Define the objective function as the $objectif$ parameter. It must be a function of the density x and the elementary sensitivity ce.

D. top88-FD

To use finite differences instead of automatic differentiation on a same problem, the steps to follow are the same, but instead of using $ForwardDiff$ package, it will be $FiniteDiff$ package.

The difference of efficiency between the version using the automatic differentiation and the version with an explicit expression for the derivative, as the SIMP codes are used, can be compared. Once the modifications in both algorithms have been made, the $Julia$ command @*time* will be useful, returning the execution time, the memory allocated during the execution and the proportion of time used for the garbage collection. An example of use is :

```
penal = 3
sensi( x, ce)= sum(sum((Emin*ones(size(x)).+(x.^penal)*(E0-Emin)).*ce));# example of the sensitivity
@time top88AD(100,100,sensi,0.5,3,2)
```

Fig. 6

E. topGGP

The GGP code is much more complicated, as the amount of possibilities is more important as SIMP methods for a similar design space. It also allows to chose different methods to solve the problem. To adapt the code to another problem than the one by default (MBB, solver GP), the obligations and possibilities are :

- Import *SparseArrays*, *LinearAlgebra*, *Plots*, *Statistics*, *SuiteSparse* and *VectorizedRoutines* packages ;
- Chose the design space through the inputs *nelx* and *nely* ;
- Select the problem through the input *problem*, or define a new one writing from the line 114 of the main function ;
- Change the end conditions through the input *maxoutit* or the parameters *kkttol* and *changetol* lines 223 and 224.

6. REFERENCES

[1] Schmit LA "Structural design by systematic synthesis" (ASCE 1960) In: 2nd ASCE conference of electrical compounds. Pittsburgh, pp 139–149

[2] Martin Philip Bendsøe and Noboru Kikuchi, "Generating optimal topologies in structural design using a homogenization method" (Elsevier B.V. 1988) https://doi.org/10.1016/0045-7825(88)90086-2

[3] Katsuyuki Suzuki and Noboru Kikuchi "A homogenization method for shape and topology optimization " (Elsevier 1991) https://doi.org/10.1016/0045-7825(91)90245-2

[4] Ole Sigmund and Kurt Maute "Topology optimization approaches" Struct Multidisc Optim 48, 1031–1055 (2013). https://doi.org/10.1007/s00158-013-0978-6

[5] Michael Yu Wang, Xiaoming Wang and Dongming Guo "A level set method for structural topology optimization" (Elsevier 2002) PII: S 0 0 4 5 - 7 8 2 5 (0 2) 0 0 5 5 9- 5

[6] Martin P.Bendsøe and Ole Sigmund "Topology optimization - Theory, Methods and Applications" (2003) 10.1007/978-3-662-05086-6

[7] Sigmund, O. (2001). A 99 line topology optimization code written in Matlab. Structural and multidisciplinary optimization, 21(2), 120-127.

[8] Andreassen, E., Clausen, A., Schevenels, M., Lazarov, B. S., Sigmund, O. (2011). Efficient topology optimization in MATLAB using 88 lines of code. Structural and Multidisciplinary Optimization, 43(1), 1-16.

[9] Liu, K., Tovar, A. (2014). An efficient 3D topology optimization code written in Matlab. Structural and Multidisciplinary Optimization, 50(6), 1175-1196.

[10] Coniglio, S., Morlier, J., Gogu, C. et al. Generalized Geometry Projection: A Unified Approach for Geometric Feature Based Topology Optimization. Arch Computat Methods Eng (2019). https://doi.org/10.1007/s11831-019-09362-8

[11] Ferrari, F., Sigmund, O. (2020). A new generation 99 line Matlab code for compliance Topology Optimization and its extension to 3D. arXiv preprint arXiv:2005.05436.

[12] Krister Svanberg « The method of moving asymptotes – a new method for structural optimization » (1986) https://doi.org/10.1002/nme.1620240207

[13] Jarrett Revels, Miles Lubin and Theodore Papamarkou "Forward-Mode Automatic Differentiation in Julia" (2016) arXiv:1607.07892 [cs.MS]

7. APPENDICES

The following code is the transcription of Ole Sigmund top88 code, from Matlab to Julia, also available on the repository given in annex.

```
using SparseArrays
using LinearAlgebra
using Plots
using Statistics
using SuiteSparse
E0 = 1;
Emin = 1e-9;
nu = 0.3;
```

```
function top88(nelx,nely,volfrac,penal,rmin,ft)

    A11 = [12  3 -6 -3;  3 12  3  0; -6  3 12 -3; -3  0 -3 12]
    A12 = [-6 -3  0  3; -3 -6 -3 -6;  0 -3 -6  3;  3 -6  3 -6]
    B11 = [-4  3 -2  9;  3 -4 -9  4; -2 -9 -4 -3;  9  4 -3 -4]
    B12 = [ 2 -3  4 -9; -3  2  9 -2;  4  9  2  3; -9 -2  3  2]
    KE = 1/(1-nu^2)/24*([A11 A12;A12' A11]+nu*[B11 B12;B12' B11])

    nodenrs = reshape(1:(1+nelx)*(1+nely),1+nely,1+nelx); #number of the nodes in columns
    edofVec = reshape(2*nodenrs[1:end-1,1:end-1].+1,nelx*nely,1) ;#1st dof of each element (x top le
    edofMat = zeros(nelx*nely, 8); #every line i contains the 8 dof of the ith element
    noeudsvoisins = [0 1 2*nely.+[2 3 0 1] -2 -1];
    for i = 1:8
        for j = 1:nelx*nely
            edofMat[j,i]= edofVec[j]+ noeudsvoisins[i] ;
        end
    end
        #meshing
    iK = reshape(kron(edofMat,ones(8,1))',64*nelx*nely,1);# line to build  K
    jK = reshape(kron(edofMat,ones(1,8))',64*nelx*nely,1);# columns
    #F = spzeros(2*(nely+1)*(nelx+1),1); F[2,1] = -1; impossible to solve in Julia with F sparse
    F = zeros(2*(nely+1)*(nelx+1),1); F[2,1] = -1;
    U = zeros(2*(nely+1)*(nelx+1),1);
    fixeddofs = union(1:2:2*(nely+1),[2*(nelx+1)*(nely+1)]);
    alldofs = 1:2*(nely+1)*(nelx+1);
    freedofs = setdiff(alldofs,fixeddofs);
        #filtering is linear + chain's rule -> matricial opration with operator H
    iH = ones(nelx*nely*(2*(convert(Int64,ceil(rmin)-1))+1)^2);#lines H
    jH = ones(size(iH));# columns
    sH = zeros(size(iH));# values
    k = 0;
    for i1 = 1:nelx
        for j1 = 1:nely
            e1 = (i1-1)*nely+j1;
            for i2 = max(i1-(ceil(rmin)-1),1):min(i1+(ceil(rmin)-1),nelx)
                for j2 = max(j1-(ceil(rmin)-1),1):min(j1+(ceil(rmin)-1),nely)
                    e2 = (i2-1)*nely+j2;
                    k = k+1;
                    iH[k] = e1;
                    jH[k] = e2;
                    sH[k] = max(0,rmin-sqrt((i1-i2)^2+(j1-j2)^2));
                end
            end
        end
    end
    H = sparse(iH,jH,sH);# matrix of the Hei
    Hs = [sum(H[i,:]) for i = 1:(size(H)[1])];# sum of the Hei for e fixed -> coeffs used to divide
        #variables
    x = volfrac*ones(nely,nelx);
    xPhys = x; #new variable for the density filter
    loop = 0;
    change = 1;
    cValues = []
    while change > 0.01
        loop = loop + 1;
        sK = [j*((i+Emin)^penal) for i in ((E0-Emin)*xPhys[:]') for j in KE[:] ];#new values EF
        K = sparse(iK[:],jK[:],sK); K = (K+K')/2;# force to be symmetric
        KK = cholesky(K[freedofs,freedofs]);
        U[freedofs] = KK\F[freedofs];
```

```
            # analyse, objective function
        edM = [convert(Int64,i) for i in edofMat] ;#to index by int
        mat = (U[edM]*KE).*U[edM];
        ce = reshape([sum(mat[i,:]) for i = 1:(size(mat)[1])],nely,nelx);
        c = sum(sum((Emin*ones(size(xPhys)).+(xPhys.^penal)*(E0-Emin)).*ce));
        push!(cValues,c)
        dc = -penal*(E0-Emin)*xPhys.^(penal-1).*ce;
        dv = ones(nely,nelx);

            #filtering
        if ft == 1
            dc[:] = H*(x[:].*dc[:])./Hs./max(1e-3,maximum(x[:]));
        elseif ft == 2
            dc[:] = H*(dc[:]./Hs);
            dv[:] = H*(dv[:]./Hs);
        end
            #Lagrangian multiplicators research, upload of the density x
        l1 = 0; l2 = 1e9; move = 0.2;
        xnew = zeros(nely, nelx);
        while (l2-l1)/(l1+l2) > 1e-3
            lmid = 0.5*(l2+l1);
            RacBe = sqrt.(-dc./dv/lmid) ;
            XB = x.*RacBe ;
            for i = 1:nelx
                for j = 1:nely
                    xji = x[j,i];
                    xnew[j,i]= max(0.001,max(xji-move,min(1,min(xji+move,XB[j,i]))));
                end
            end
            if ft == 1
                xPhys = xnew;
            elseif ft == 2 # if we chose the density filter
                xPhys[:] = (H*xnew[:])./Hs;
            end
            if sum(xPhys[:]) > volfrac*nelx*nely
                l1 = lmid;
            else
                l2 = lmid;
            end
        end
        change = maximum(abs.(x-xnew));
        x = xnew;
        println("loop = ", loop, "Change = ", change ," c = ", c , " densité struct = ", mean(x) )
        if loop == 25
            return heatmap(x), loop, cValues
        end
    end
    return heatmap(x) , loop , cValues
end
```