TopOpt in Python Documentation

Release 0.0.9

A.J.J. Lagerweij

Oct 19, 2021

About

1 Introduction **1**

2 About **3**
- 2.1 About this Project . . . 3
- 2.2 Background in Topology Optimization . . . 3
- 2.3 Different Objectives . . . 11
- 2.4 Setup of the Code . . . 11
- 2.5 MIT License . . . 11

3 Theory and Examples **13**
- 3.1 Global Compliance Minimization . . . 13
- 3.2 Maximum Local Compliance . . . 17
- 3.3 Stress Intensity Factor Minimization . . . 20
- 3.4 Fatigue Crack Growth Life Maximization . . . 26

4 Docstrings **37**
- 4.1 Global Compliance Minimization . . . 37
- 4.2 Maximum Local Compliance . . . 52
- 4.3 Stress Intensity Factor Minimization . . . 66
- 4.4 Fatigue Crack Growth Life Maximization . . . 86

5 Indices and Tables **105**

Python Module Index **107**

Index **109**

CHAPTER 1

Introduction

CHAPTER 2

About

2.1 About this Project

2.2 Background in Topology Optimization

Fig. 2.2.1: Topology optimization example, a cantilever beam with maximum stiffness.

This chapter will provide the reader with a basic insight into topology optimization (TO). can alter the layout of the structure. Within a design space it tries to distribute a limited amount of material such that a certain objective is maximized or minimized. This design space is limited by; the size of the design region, a material constrain, boundary conditions and others.

Here the formulation of a basic algorithm and the problems that can be encountered are disucessed. It will provide the reader the basic grasp that is required before a change in optimization objective can be discussed. For that purpose it will introduce a basic example of the TO algorithm that minimizes the global compliance, and thus maximizes stiffness.

This type of TO tries to minimize the global compliance. It will be the main example algorithm as it has been researched and documented extensively among others by the TopOpt group at the Technical University of Denmark

(DTU)[1],[2],[3],[4],[5]. The goal of the method is to minimize the compliance by distributing the assigned mass. It has to satisfy certain constraints, the volume constrain V limits the amount of mass available and the structure should be in equilibrium. If required, more constraints can be formulated. One can limit the size of the finest features and take manufacturing limitations in account or introduce a local density constraint to create porous structures which ensures structural stability[6].

Different implementations of global compliance TO exist, the one discussed here is based on a gradient method. Hence, it requires a continuous expression for the compliance as a function of the mass/density distribution. Therefore, it must allow elements with density values that are between 0 and 1 and it uses a proportional stiffness with penalization method (SIMP) to approximate a discrete 0-1 problem.

- *Continuum Formulation*
- *Discretization*
- *Sensitivity analysis and MMA*
- *Filtering Techniques*
- *Computational Implementation*
- *Changing the Objective*
- *References*

2.2.1 Continuum Formulation

The linear elastic optimization for small deformation as presented by N. Olhoff and J.E. Taylor[7] is used. It considers a design region Ω that is in $\boldsymbol{R}^2$ or $\boldsymbol{R}^3$ of which a subregion Ω^m is filled with material[1]. The optimal topology is reached when the optimal stiffness tensor $\boldsymbol{E}_{ijkl}(\boldsymbol{x})$ is found.

[1] M. P. Bendsøe, "Optimal shape design as a material distribution problem," Struct. Optim., vol. 1, no. 4, pp. 193–202, Dec. 1989.

[2] O. Sigmund, "A 99 line topology optimization code written in matlab," Struct. Multidiscip. Optim., vol. 21, no. 2, pp. 120–127, 2001.

[3] M. P. Bendsøe and O. Sigmund, Topology Optimization. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[4] B. S. Lazarov and O. Sigmund, "Filters in topology optimization based on Helmholtz-type differential equations," Int. J. Numer. Methods Eng., vol. 86, no. 6, pp. 765–781, May 2011.

[5] E. Andreassen, A. Clausen, M. Schevenels, B. S. Lazarov, and O. Sigmund, "Efficient topology optimization in MATLAB using 88 lines of code," Struct. Multidiscip. Optim., vol. 43, no. 1, pp. 1–16, Jan. 2011.

[6] J. Wu, N. Aage, R. Westermann, and O. Sigmund, "Infill Optimization for Additive Manufacturing—Approaching Bone-Like Porous Structures," IEEE Trans. Vis. Comput. Graph., vol. 24, no. 2, pp. 1127–1140, Feb. 2018.

[7] N. Olhoff and J. E. Taylor, "On Structural Optimization," J. Appl. Mech., vol. 50, no. 4b, p. 1139, 1983.

As all space within Ω^m is filled an equation of the mass distribution X can be formulated as a discrete function,

$$X(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{x} \in \Omega^m \\ 0 & \text{if } \boldsymbol{x} \in \Omega \backslash \Omega^m \end{cases}$$

This can be used to define the stiffness tensor,

$$\boldsymbol{E}_{ijkl}(\boldsymbol{x}) = X(\boldsymbol{x}) \overline{\boldsymbol{E}}_{ijkl}$$

in terms of this mass distribution function and the constant rigidity tensor $\overline{\boldsymbol{E}}_{ijkl}$. The constant rigidity tensor is function of the material properties only. As X is a discrete function all admissible tensors are discrete and thus the optimization problem has a discrete valued parameter function.

The amount of work due of the deformation $\boldsymbol{u}$ can be calculated by with a virtual work method. With the standard linearized strain formulation this results in,

$$l(\boldsymbol{u}) = \int_\Omega \boldsymbol{f}\boldsymbol{u} \, \mathrm{d}\Omega + \int_{\Gamma_T} \boldsymbol{t}\boldsymbol{u} \, \mathrm{d}\Gamma_T$$

A bi-linear energy equation with virtual work $a(\boldsymbol{u}, \hat{\boldsymbol{u}})$ is formulated,

$$a(\boldsymbol{u}, \hat{\boldsymbol{u}}) = \int_\Omega \boldsymbol{E}_{ijkl} \boldsymbol{\varepsilon}_{kl}(\boldsymbol{u}) \boldsymbol{\varepsilon}_{ij}(\hat{\boldsymbol{u}}) \, \mathrm{d}\Omega$$

$\hat{\boldsymbol{u}}$ is an arbitrary kinematically admissible deformation. Equilibrium is ensured when $l(\hat{\boldsymbol{u}}) = a(\boldsymbol{u}, \hat{\boldsymbol{u}})$ is satisfied for all admissible deformations $\hat{\boldsymbol{u}}$.

As minimizing the work, due to the traction forces for a given load, minimizes the deformation of a structure the problem can be formulated as:

$$\begin{aligned} \min_{\Omega^m} \ & l(\boldsymbol{u}) \\ \text{s.t. :} \ & a(\boldsymbol{u}, \hat{\boldsymbol{u}}) = l(\hat{\boldsymbol{u}}) \\ & \int_\Omega X(\boldsymbol{x}) \mathrm{d}\Omega \ = \ \mathrm{Vol}(\Omega^m) \ \leq \ V \end{aligned}$$

2.2.2 Discretization

To solve the continuum problem of the previous section it is discretized into a finite element analysis with N elements:

$$\begin{aligned} \min_{X_1, X_2, \ldots, X_N} \ & c = \boldsymbol{f}^T \boldsymbol{u} \\ \text{s.t. :} \ & \boldsymbol{K}\boldsymbol{u} = \boldsymbol{f} \\ & \sum_{e=1}^{N} v_e X_e \ \leq \ V \\ & X_e \in \{0, 1\} \quad \forall \quad e \in \{1, 2, \ldots, N\} \\ \text{where :} \ & \boldsymbol{K} = \sum_{e=1}^{N} \boldsymbol{K}_e(X_e, \overline{E}) \end{aligned}$$

it shows that the element stiffness matrix $\boldsymbol{K_e}$ depends on the element material value X_e and the material stiffness $\overline{E}$. The problem becomes unstable towards the element type and mesh when the discrete formulations of density are used. Such a distribution problem generally has no solution[8],[9]. Iterative search methods would not work because

[8] G. Strang and R. V. Kohn, "Optimal design in elasticity and plasticity," Int. J. Numer. Methods Eng., vol. 22, no. 1, pp. 183–188, Jan. 1986.

[9] R. V. Kohn and G. Strang, "Optimal design and relaxation of variational problems, I," Commun. Pure Appl. Math., vol. 39, no. 1, pp. 113–137, 1986.

they require the calculation of gradients. Therefore, the problem is changed so that the density becomes a continuous equation ranging from 0 to 1.

$$0 \leq X_e \leq 1$$

This method would result in a design with intermediate values. Although this makes sense for variable thickness plate design, see the work of M.P. Rossow and J.E. Taylor[10], for discrete topology design loses its direct physical representation. There is either material or there is not, intermediate values are meaningless. Adding a penalization that reduces the effectiveness of intermediate values results in a formulation that suppresses these intermediate values. The method used here, developed by E. Andreassen[5], is derived from the classical penalized proportional stiffness method (SIMP)[1],[3]. Here $E_{\min}$ is a small artificial stiffness used to avoid elements with zero stiffness as that could make the FEA unstable.

$$\boldsymbol{E}_{ijkl}(\boldsymbol{x}) = \boldsymbol{E}_{ijkl,\min} + X(\boldsymbol{x})^p \left(\overline{\boldsymbol{E}}_{ijkl} - \boldsymbol{E}_{ijkl,\min}\right)$$

When $p > 1$ the intermediate density values are less effective as there stiffness is low in comparison to the volume occupied. When p is sufficiently large, generally $p \geq 3$, the design converges to a solution that is close to a discrete (0-1) design.

2.2.3 Sensitivity analysis and MMA

The main focus on developing a robust and stable algorithm is the update scheme. The MMA scheme was chosen as it proofed to be very effective for this type of optimization[3]. MMA is an efficient method meant for non-linear non-convex problems that approaches those problems by generating purely convex sub-problems, based on the gradient information. It can be used to iterative solve the optimization problem.

The gradient of one element in the discretized form is $\partial c / \partial X_e$. This derivative does not have to be explicitly calculated as the problem is self adjoint. This is used by the following proof. It starts with a new formulation of the work, the difference is the zero term at the end. Again $\hat{\boldsymbol{u}}$ is any arbitrary admissible deformation[3].

$$c = \boldsymbol{f}^T \boldsymbol{u} - \hat{\boldsymbol{u}}^T \left(\boldsymbol{K}\boldsymbol{u} - \boldsymbol{f}\right)$$

taking the derivative to the density leads to:

$$\frac{\partial c}{\partial X_e} = \left(\boldsymbol{f}^T - \hat{\boldsymbol{u}}^T \boldsymbol{K}\right) \frac{\partial \boldsymbol{u}}{\partial X_e} - \hat{\boldsymbol{u}}^T \frac{\partial \boldsymbol{K}}{\partial X_e} \boldsymbol{u}$$

when $\hat{\boldsymbol{u}}$ satisfies the adjoint equation it becomes:

$$\frac{\partial c}{\partial X_e} = - \hat{\boldsymbol{u}}^T \frac{\partial \boldsymbol{K}}{\partial X_e} \boldsymbol{u}$$
$$\text{when} \quad \boldsymbol{f}^T - \hat{\boldsymbol{u}}^T \boldsymbol{K} = 0$$

Satisfying this adjoint equation is simple, just choose $\hat{\boldsymbol{u}} = \boldsymbol{u}$. The derivative of the stiffness matrix to the density of an element can be derived leading to the final expression of the gradient:

$$\frac{\partial c}{\partial X_e} = -p X_e^{p-1} \boldsymbol{u}^T \boldsymbol{K}_e \boldsymbol{u}$$

MMA approaches the problem with multiple convex approximations around the expansion point (current iteration). The goal here is to find the optimal density distribution of the current iteration where the influence of the densities is approximated with a convex function. This approximation is based on the sensitivity and some information of previous

[10] M. P. Rossow and J. E. Taylor, "A Finite Element Method for the Optimal Design of Variable Thickness Sheets," AIAA J., vol. 11, no. 11, pp. 1566–1569, Nov. 1973.

iterations. Solving these convex equation can be done by various basic algorithms. The obtained optimum is not the real optimum of the optimization problem as the convex function used is only an approximation of the real problem. However, it is a step into the direction of the real optimum. The obtained density distribution is then used as an input of the next iteration[3] (pp. 19-21). The optimization of this local problem must meet all the constraints. This means that the updated design has to meet the global volume constraint.

The MMA will approximate the compliance at iteration k. Here X^k is a vector with the densities of all elements at the current iteration. A description on the calculations of U_e and L_e follows later. The method was developed by K. Svansberg[11].

$$
\begin{aligned}
c &\approx c^k + \sum_{e=1}^{N} \left(\frac{r_e}{U_e - X_e} + \frac{s_e}{X_e - L_e} \right) \\
\text{where:} \quad r_e &= \begin{cases} 0 & \text{if } \frac{\partial c}{\partial X_e} \leq 0 \\ \left(U_e - X_e^k\right)^2 \frac{\partial c}{\partial X_e} & \text{if } \frac{\partial c}{\partial X_e} > 0 \end{cases} \\
s_e &= \begin{cases} 0 & \text{if } \frac{\partial c}{\partial X_e} \geq 0 \\ -\left(X_e^k - L_e\right)^2 \frac{\partial c}{\partial X_e} & \text{if } \frac{\partial c}{\partial X_e} < 0 \end{cases}
\end{aligned}
$$

That all the density sensitivities are negative can be derived from adjoint sensitivity equation. This simplifies the expression and resulted in:

$$
c \approx c^k + \sum_{e=1}^{N} -\frac{\left(X_e^k - L_e\right)^2}{X_e - L_e} \frac{\partial c}{\partial X_e}
$$

Then the optimization, on X_e, used in this iteration is defined as:

$$
\begin{aligned}
\min_{X_1, X_2, \ldots, X_N} \quad & c^k - \sum_{e=1}^{N} \frac{\left(X_e^k - L_e\right)^2}{X_e - L_e} \frac{\partial c}{\partial X_e} \\
\text{s.t. :} \quad & \sum_{e=1}^{N} v_e X_e \leq V \\
& 0 \geq X_e \geq 1 \quad \forall \quad e \in \{1, 2, \ldots, N\}
\end{aligned}
$$

here the moving asymptote, L_e, can be varied and is chosen to improve convergence and stability, choosing this wisely is important. In general the goal is to stabilize the process when it is oscillating, i.e. moving the asymptote closer. Or to relax the problem when it is monotone, i.e. moving the asymptote further and thus causing larger steps to be taken at that iteration. This can be done by including the behavior of previous iterations or calculating the second derivative of the optimization objective to the design variables. Several implementations exist, they are tuned to work for specific problems[11],[12].

The update scheme minimizes the local approximation to decide on the new densities. Starting with the minimalization of the Lagrange function:

$$
\mathcal{L} = c^k - \sum_{e=1}^{N} \frac{\left(X_e^k - L_e\right)^2}{X_e - L_e} \frac{\partial c}{\partial X_e} + \Lambda \left(\sum_{e=1}^{N} v_e X_e - V \right) + \sum_{e=1}^{N} \lambda_e^- \left(X_e - 0\right) + \sum_{e=1}^{N} \lambda_e^+ \left(1 - X_e\right)
$$

This separable and purely convex problem can be solved by a range of algorithms. It can easily be changed into a formulation with other or more constraints.

11 K. Svanberg, "The method of moving asymptotes - a new method for structural optimization," Int. J. Numer. Methods Eng., vol. 24, no. 2, pp. 359–373, Feb. 1987.

12 K. Svanberg, "MMA and GCMMA – two methods for nonlinear optimization," Stockholm, Sweden, 2007.

2.2.4 Filtering Techniques

Filtering the sensitivities was proposed by O. Sigmund[13] . The method is derived from image processing and uses a normalized convolution filter to blur the figure. The density distribution X_e and the gradient can be interpreted as a figure with gray scale pixels. The gradient itself is not filtered, but the gradient multiplied by the densities is filtered before the update scheme decides on the densities of the next iteration[14],[15].

The sensitivity filter can be described as,

$$\frac{\widehat{\partial C}}{\partial X_k} = \frac{1}{X_k \sum_{i=1}^{N} H_i} \sum_{i=1}^{N} H_i \, X_i \, \frac{\partial l(\boldsymbol{u})}{\partial X_i}$$

$$H_i = \begin{cases} r_{min} - \text{dist}(k,i) & \text{if} \quad \text{dist}(k,i) < r_{min} \\ 0 & \text{if} \quad \text{dist}(k,i) \geq r_{min} \end{cases}$$

where k is the element to be filtered. The value of the filtered compliance density gradient at element i is depended on three main things, the density, density gradient and the distance to the surrounding nodes i. All nodes that fall within radius r_{min} are contributing but the further the node is the lower its contribution. Note that the filter is normalized by dividing it by $\sum \hat{H}_i$. There is limited understanding why this filter works, there is no physical or theoretical basis for it. From experience, it was simply observed that it works well.

a)		
b)		
c)		

Fig. 2.2.2: Optimized cantilever beams at resolution, a) 250x50, b) 500x100 and c) 1000x200. A sensitivity filter of increasing filter radius is used to avoid checkerboard patterns for the figures at the rigth side.

Fig. 2.2.2 show the same simulations. The only difference is that the simulations is that the are filtered. It was observed that scaling the filter size r_{min} with the resolution results in similar designs. The main difference between the designs is that higher resolution simulations result in a smoother structure. But filtering this way leads to less discrete designs. Larger filters cause more pixels to have intermediate density values. Three solutions do exist; lowering the filter size for the last couple of iterations, increasing the SIMP penalty factor or applying extra post processing steps.

Another filter that can be considered is the linear density filter which was proposed by T.E. Bruns, D.A. Tortorelli and

[13] O. Sigmund, "Design of Material Structures Using Topology Optimization," PHD thesis, 1994, pp. 72-75.

[14] O. Sigmund, "On the design of compliant mechanisms using topology optimization," Mech. Struct. Mach., vol. 25, no. 4, pp. 493–524, 1997.

[15] O. Sigmund and J. Petersson, "Numerical instabilities in topology optimization: A survey on procedures dealing with checkerboards, mesh-dependencies and local minima," Struct. Optim., vol. 16, no. 1, pp. 68–75, Aug. 1998.

B. Bourdin[16],[17]. Here the blur filter,

$$\widehat{X_e} = \frac{1}{\sum_{i=1}^{N} H_i} \sum_{i=1}^{N} H_i \, X_i$$

$$H_i = \begin{cases} r_{min} - \text{dist}(k,i) & \text{if} \quad \text{dist}(k,i) < r_{min} \\ 0 & \text{if} \quad \text{dist}(k,i) \geq r_{min} \end{cases}$$

is applied directly on the densities. These filtered densities, $\widehat{X_e}$, are used in the FEA and SA. This means that the design variables X_e lose there physical meaning as the FEA gives it the relation to reality, therefore the final geometry should be based on the filtered densities[18].

A comparison between Fig. 2.2.3 shows that filtering the densities suppresses the finer features well. Comparing the performance difference of the sensitivity and density filters is difficult. Many criteria can be used such as, computational effort, how discrete the final design is, the magnitude of the final compliance and whether the volume constrained is still maintained. A small comparison was made by O. Sigmund[18]. The performance of the filters depends greatly on the design case used. The paper clearly shows that better filters exist then those presented in this communication however as the density and sensitivity filters are computational efficient and simple to implement they were chosen as the basic filters used in the code.

a)		
b)		
c)		

Fig. 2.2.3: Optimized cantilever beams at resolution, a) 250x50, b) 500x100 and c) 1000x200. A density filter of increasing filter radius is used to avoid checkerboard patterns for the figures at the rigth side.

2.2.5 Computational Implementation

The iterative implementation of topology optimization as proposed by M. Beckers,[19] or M.P. Bendsøe and O. Sigmund[3] are similar. It exists out of three parts, initialization, optimization and post processing. The flowchart for the methods used in this communication can be found in Fig. 2.2.4.

Fig. 2.2.4: Basic flowchart for compliance minimization[3].

16 T. E. Bruns and D. A. Tortorelli, "Topology optimization of non-linear elastic structures and compliant mechanisms," Comput. Methods Appl. Mech. Eng., vol. 190, no. 26–27, pp. 3443–3459, Mar. 2001.

17 B. Bourdin, "Filters in topology optimization," Int. J. Numer. Methods Eng., vol. 50, no. 9, pp. 2143–2158, Mar. 2001.

18 O. Sigmund, "Morphology-based black and white filters for topology optimization," Struct. Multidiscip. Optim., vol. 33, no. 4–5, pp. 401–424, Feb. 2007.

19 M. Beckers, "Topology optimization using a dual method with discrete variables," Struct. Optim., vol. 17, no. 1, pp. 14–24, Feb. 1999.

In the initialization phase the problem is set up. It defines the design domain, the loading conditions, the initial design and generates the finite element mesh that will be used in the optimization phase.

The optimization phase is the iterative method that solves the topology problem. It will analyze the current design with a FEA. After which it will calculate the sensitivity of the global compliance to the density of each element, this is the local gradient of which the calculation is discussed before The Method of Moving Asymptotes (MMA), developed by K. Svanberg[11], is used to formulate a simplified convex approximation of the problem which is optimized to formulate the updated design. These steps are performed in a loop until the design is converged, i.e. when the change in design between two iterations becomes negligible.

Post processing is required to remove the last elements with intermediate values and generate a shape out of the design, for example a CAD or STL file. This algorithm will not contain any of the post processing steps. The code used in this communication simply plots the final shape and load case.

2.2.6 Changing the Objective

Topology optimization can be used for several objectives; classical examples are, truss structure design, antenna/microphone design, heat convection problems[3],[20] and MEMS actuator designs[2],[3],[21],[22]. In general all these TO algorithms approach the optimization as a material distribution problem within a design space with a resource constraint witch is solved with an iterative gradient method.

When changing the objective and/or problem one should start with a formulation of the problem which consists of the objective, variables and constraints. Then the changes should be made in the calculation of the objective and sensitivity. Important therefor is the method used to link the optimization variables to the objective, in the case of compliance minimization it consists of the variables to density formulation (SIMP cref{eq:SIMP_Lit}) and the FEA that links stiffness to compliance. Beneficial would be a (self) adjoint formulation because it allows for an efficient calculation of the sensitivities. The parts of the method that are unlikely to change are; the overall methodology, described in cref{fig:Flowchart_Lit}, the method of moving asymptotes and its update scheme.

Sometimes optimization objectives are formulated in the form of several sub objectives resulting in multi objective optimization formulations. Optimizing for multiple objectives or load cases at once is common. For most structures several considerations, such as costs, weight and strength are taken in account. In addition do most structures experience multiple load-cases during their life. Several TO algorithms have been developed for this purpose. The most basic methods will be discussed here.

Fig. 2.2.5: Flowchart of the multi loadcase compliance minimization algorithm[3].

The method sets up multiple FEA as shown in Fig. 2.2.5. Then the total objective will be linked to sub objectives. For instance the goal might be to minimize the compliance due to n load cases. One could formulate the total objective (O) as the weighted sum of the compliance of all load cases,

$$O = \sum_{i=1}^{n} w_i c_i$$

20

S. Turteltaub, "Functionally graded materials for prescribed field evolution," Comput. Methods Appl. Mech. Eng., vol. 191, no. 21–22, pp. 2283–2296, Mar. 2002.

21

O. Sigmund, "Design of multiphysics actuators using topology optimization – Part I: One-material structures," Comput. Methods Appl. Mech. Eng., vol. 190, no. 49–50, pp. 6577–6604, Oct. 2001.

22

O. Sigmund, "Design of multiphysics actuators using topology optimization – Part II: Two-material structures," Comput. Methods Appl. Mech. Eng., vol. 190, no. 49–50, pp. 6605–6627, Oct. 2001.

resulting in a gradient function that can be formulated as,

$$\frac{\partial O}{\partial X_e} = \sum_{i=1}^{n} w_i \frac{\partial c_i}{\partial X_e}$$

Another example can be made with a similar method. Assume that adding up the objective is not what is wanted but that the goal is to prohibit two different failure modes. Hence, the design update is based on the most critical case resulting in objective,

$$O = \max\left(o_1, o_2, \ldots, o_n\right)$$

An example of such a formulation can be found in the TO based damage tolerance optimization algorithm presented by Z. Kang, P. Liu and M. Li[23]. Where they optimize geometries for the most cricital crack in every iteration. The sensitivities can then be formulated as:

$$\frac{\partial O}{\partial X_e} = \sum_{i=1}^{n} s_i \frac{\partial o_i}{\partial X_e} \tag{2.2.1}$$

$$\text{where} \quad s_i = \begin{cases} 1 & \text{if} \quad o_i = O \\ 0 & \text{if} \quad o_i \neq O \end{cases} \tag{2.2.2}$$

These basic multiple load case algorithms can be summarized in the flowchart shown in Fig. 2.2.5. In general the FEA requires most of the computational time therefore the method as shown here is computationally inefficient. More advanced algorithms have been developed but these are outside the scope of this communication[24],[25].

2.2.7 References

2.3 Different Objectives

2.4 Setup of the Code

2.5 MIT License

23

26. Kang, P. Liu, and M. Li, "Topology optimization considering fracture mechanics behaviors at specified locations," Struct. Multidiscip. Optim., vol. 55, no. 5, pp. 1847–1864, May 2017.

24

K. A. James, J. S. Hansen, and J. R. R. A. Martins, "Structural topology optimization for multiple load cases using a dynamic aggregation technique," Eng. Optim., vol. 41, no. 12, pp. 1103–1118, 2009.

25

E. Nutu, "Multiple load case topology optimization based on bone mechanical adaptation theory," UPB Sci. Bull. Ser. D Mech. Eng., vol. 77, no. 4, pp. 131–140, 2015.

CHAPTER 3

Theory and Examples

3.1 Global Compliance Minimization

This type of TO tries to minimize the global compliance. It will be the main example algorithm as it has been researched and documented extensively among others by the TopOpt group at the Technical University of Denmark (DTU)[1],[2],[3],[4],[5],[6]. The goal of the method is to minimize the compliance by distributing the assigned mass. It has to satisfy certain constraints, the volume constrain V limits the amount of mass available and the structure should be in equilibrium.

- *Continuum Formulation*
- *Discretization*

[1] M. P. Bendsøe, "Optimal shape design as a material distribution problem," Struct. Optim., vol. 1, no. 4, pp. 193–202, Dec. 1989.

[2] O. Sigmund, "A 99 line topology optimization code written in matlab," Struct. Multidiscip. Optim., vol. 21, no. 2, pp. 120–127, 2001.

[3] M. P. Bendsøe and O. Sigmund, Topology Optimization. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[4] B. S. Lazarov and O. Sigmund, "Filters in topology optimization based on Helmholtz-type differential equations," Int. J. Numer. Methods Eng., vol. 86, no. 6, pp. 765–781, May 2011.

[5] E. Andreassen, A. Clausen, M. Schevenels, B. S. Lazarov, and O. Sigmund, "Efficient topology optimization in MATLAB using 88 lines of code," Struct. Multidiscip. Optim., vol. 43, no. 1, pp. 1–16, Jan. 2011.

[6] J. Wu, N. Aage, R. Westermann, and O. Sigmund, "Infill Optimization for Additive Manufacturing—Approaching Bone-Like Porous Structures," IEEE Trans. Vis. Comput. Graph., vol. 24, no. 2, pp. 1127–1140, Feb. 2018.

- *Sensitivity analysis*
- *Computational Implementation*
- *Example and Results*
- *References*

3.1.1 Continuum Formulation

The linear elastic optimization for small deformation as presented by N. Olhoff and J.E. Taylor[7] is used. It considers a design region Ω that is in $\boldsymbol{R}^2$ or $\boldsymbol{R}^3$ of which a subregion Ω^m is filled with material[1]. The optimal topology is reached when the optimal stiffness tensor $\boldsymbol{E}_{ijkl}(\boldsymbol{x})$ is found.

As all space within Ω^m is filled an equation of the mass distribution X can be formulated as a discrete function,

$$X(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{x} \in \Omega^m \\ 0 & \text{if } \boldsymbol{x} \in \Omega \backslash \Omega^m \end{cases}$$

This can be used to define the stiffness tensor,

$$\boldsymbol{E}_{ijkl}(\boldsymbol{x}) = X(\boldsymbol{x}) \overline{\boldsymbol{E}}_{ijkl}$$

in terms of this mass distribution function and the constant rigidity tensor $\overline{\boldsymbol{E}}_{ijkl}$. The constant rigidity tensor is function of the material properties only. As X is a discrete function all admissible tensors are discrete and thus the optimization problem has a discrete valued parameter function.

The amount of work due of the deformation $\boldsymbol{u}$ can be calculated by cref{eq:LinLoad}. With the standard linearized strain formulation this results in,

$$l(\boldsymbol{u}) = \int_\Omega \boldsymbol{f}\boldsymbol{u} \, \mathrm{d}\Omega + \int_{\Gamma_T} \boldsymbol{t}\boldsymbol{u} \, \mathrm{d}\Gamma_T$$

A bi-linear energy equation with virtual work $a(\boldsymbol{u}, \hat{\boldsymbol{u}})$ is formulated,

$$a(\boldsymbol{u}, \hat{\boldsymbol{u}}) = \int_\Omega \boldsymbol{E}_{ijkl} \boldsymbol{\varepsilon}_{kl}(\boldsymbol{u}) \boldsymbol{\varepsilon}_{ij}(\hat{\boldsymbol{u}}) \, \mathrm{d}\Omega$$

$\hat{\boldsymbol{u}}$ is an arbitrary kinematically admissible deformation. Equilibrium is ensured when $l(\hat{\boldsymbol{u}}) = a(\boldsymbol{u}, \hat{\boldsymbol{u}})$ is satisfied for all admissible deformations $\hat{\boldsymbol{u}}$.

As minimizing the work, due to the traction forces for a given load, minimizes the deformation of a structure the problem can be formulated as:

$$\begin{aligned} \min_{\Omega^m} \ & l(\boldsymbol{u}) \\ \text{s.t.} : \ & a(\boldsymbol{u}, \hat{\boldsymbol{u}}) = l(\hat{\boldsymbol{u}}) \\ & \textstyle\int_\Omega X(\boldsymbol{x}) \mathrm{d}\Omega \ = \ \mathrm{Vol}(\Omega^m) \ \leq \ V \end{aligned}$$

[7] N. Olhoff and J. E. Taylor, "On Structural Optimization," J. Appl. Mech., vol. 50, no. 4b, p. 1139, 1983.

3.1.2 Discretization

To solve the continuum problem of the previous section it is discretized into a finite element analysis with N elements:

$$\begin{aligned} \min_{X_1,X_2,\ldots,X_N} \quad & c = \boldsymbol{f}^T \boldsymbol{u} \\ \text{s.t. :} \quad & \boldsymbol{K}\boldsymbol{u} = \boldsymbol{f} \\ & \sum_{e=1}^{N} v_e X_e \ \leq \ V \\ & X_e \in \{0,1\} \quad \forall \quad e \in \{1,2,\ldots,N\} \\ \text{where :} \quad & \boldsymbol{K} = \sum_{e=1}^{N} \boldsymbol{K}_e(X_e, \overline{E}) \end{aligned}$$

it shows that the element stiffness matrix $\boldsymbol{K_e}$ depends on the element material value X_e and the material stiffness $\overline{E}$. The problem becomes unstable towards the element type and mesh when the discrete formulation of cref{eq:conti mass distribution,eq:stiffness_Lit} are used. Such a distribution problem generally has no solution[8,9]. Iterative search methods would not work because they require the calculation of gradients. Therefore, the problem is changed so that the density becomes a continuous equation ranging from 0 to 1.

$$0 \leq X_e \leq 1$$

This method would result in a design with intermediate values. Although this makes sense for variable thickness plate design, see the work of M.P. Rossow and J.E. Taylor[10], for discrete topology design loses its direct physical representation. There is either material or there is not, intermediate values are meaningless. Changing cref{eq:stiffness_Lit} with a penalization that reduces the effectiveness of intermediate values results in a formulation that suppresses these intermediate values. The method used here, developed by E. Andreassen[5], is derived from the classical penalized proportional stiffness method (SIMP)[1,3]. Here $E_{\min}$ is a small artificial stiffness used to avoid elements with zero stiffness as that could make the FEA unstable.

$$\boldsymbol{E}_{ijkl}(\boldsymbol{x}) = \boldsymbol{E}_{ijkl,\min} + X(\boldsymbol{x})^p \left(\overline{\boldsymbol{E}}_{ijkl} - \boldsymbol{E}_{ijkl,\min}\right)$$

When $p > 1$ the intermediate density values are less effective as there stiffness is low in comparison to the volume occupied. When p is sufficiently large, generally $p \geq 3$, the design converges to a solution that is close to a discrete (0-1) design.

3.1.3 Sensitivity analysis

The gradient of one element in the discretized form is $\partial c / \partial X_e$. This derivative does not have to be explicitly calculated as the problem is self adjoint. This is used by the following proof. It starts with a new formulation of the work, the difference is the zero term at the end. Again $\hat{\boldsymbol{u}}$ is any arbitrary admissible deformation[3].

$$c = \boldsymbol{f}^T \boldsymbol{u} - \hat{\boldsymbol{u}}^T \left(\boldsymbol{K}\boldsymbol{u} - \boldsymbol{f}\right)$$

[8] G. Strang and R. V. Kohn, "Optimal design in elasticity and plasticity," Int. J. Numer. Methods Eng., vol. 22, no. 1, pp. 183–188, Jan. 1986.

[9] R. V. Kohn and G. Strang, "Optimal design and relaxation of variational problems, I," Commun. Pure Appl. Math., vol. 39, no. 1, pp. 113–137, 1986.

[10] M. P. Rossow and J. E. Taylor, "A Finite Element Method for the Optimal Design of Variable Thickness Sheets," AIAA J., vol. 11, no. 11, pp. 1566–1569, Nov. 1973.

taking the derivative to the density leads to:

$$\frac{\partial c}{\partial X_e} = \left(\boldsymbol{f}^T - \hat{\boldsymbol{u}}^T \boldsymbol{K}\right) \frac{\partial \boldsymbol{u}}{\partial X_e} - \hat{\boldsymbol{u}}^T \frac{\partial \boldsymbol{K}}{\partial X_e} \boldsymbol{u}$$

when $\hat{\boldsymbol{u}}$ satisfies the adjoint equation it becomes:

$$\frac{\partial c}{\partial X_e} = -\hat{\boldsymbol{u}}^T \frac{\partial \boldsymbol{K}}{\partial X_e} \boldsymbol{u}$$
$$\text{when} \quad \boldsymbol{f}^T - \hat{\boldsymbol{u}}^T \boldsymbol{K} = 0$$

Satisfying this adjoint equation is simple, just choose $\hat{\boldsymbol{u}} = \boldsymbol{u}$. The derivative of the stiffness matrix to the density of an element can be derived leading to the final expression of the gradient:

$$\frac{\partial c}{\partial X_e} = -p X_e^{p-1} \boldsymbol{u}^T \boldsymbol{K}_e \boldsymbol{u}$$

3.1.4 Computational Implementation

The iterative implementation of topology optimization as proposed by M. Beckers,[11] or M.P. Bendsøe and O. Sigmund[3] are similar. It exists out of three parts, initialization, optimization and post processing. The flowchart for the methods used in this algorithm can be found in Fig. 3.1.1.

Fig. 3.1.1: Basic flowchart for compliance minimization[3].

In the initialization phase the problem is set up. It defines the design domain, the loading conditions, the initial design and generates the finite element mesh that will be used in the optimization phase.

The optimization phase is the iterative method that solves the topology problem. It will analyze the current design with a FEA. After which it will calculate the sensitivity of the global compliance to the density of each element, this is the local gradient of which the calculation is discussed in *Sensitivity analysis and MMA*. The Method of Moving Asymptotes (MMA), developed by K. Svanberg[12], is used to formulate a simplified convex approximation of the problem which is optimized to formulate the updated design. These steps are performed in a loop until the design is converged, i.e. when the change in design between two iterations becomes negligible.

Post processing is required to remove the last elements with intermediate values and generate a shape out of the design, for example a CAD or STL file. This algorithm will not contain any of the post processing steps. The code used in this communication simply plots the final shape and load case.

3.1.5 Example and Results

Example code and results!!!!!!!!!!!!!!!!!!

[11] M. Beckers, "Topology optimization using a dual method with discrete variables," Struct. Optim., vol. 17, no. 1, pp. 14–24, Feb. 1999.

[12] K. Svanberg, "The method of moving asymptotes - a new method for structural optimization," Int. J. Numer. Methods Eng., vol. 24, no. 2, pp. 359–373, Feb. 1987.

3.1.6 References

3.2 Maximum Local Compliance

Maximizing the output displacement of one, or more, nodes for a given load case results in so called Compliant Mechanisms. These geometries will behave like a normal mechanism but without any hinges, the displacement and force are transfered by (elastic) deformations only. Avoiding hinges can be required for various reasons, think of manufacturing constraints or reliability issues. Compliant mechanisms are used in various occasions from MEMS actuators to space structures. For more information check this youtube video.

- *Continuum Formulation*
- *Discretization*
- *Sensitivity analysis*
- *Computational Implementation*
- *Example and Results*
- *References*

3.2.1 Continuum Formulation

The linear elastic optimization for small deformation as presented by N. Olhoff and J.E. Taylor[1] is used. It considers a design region Ω that is in $\boldsymbol{R}^2$ or $\boldsymbol{R}^3$ of which a subregion Ω^m is filled with material[2]. The optimal topology is reached when the optimal stiffness tensor $\boldsymbol{E}_{ijkl}(\boldsymbol{x})$ is found.

As all space within Ω^m is filled an equation of the mass distribution X can be formulated as a discrete function,

$$X(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{x} \in \Omega^m \\ 0 & \text{if } \boldsymbol{x} \in \Omega \backslash \Omega^m \end{cases}$$

This can be used to define the stiffness tensor,

$$\boldsymbol{E}_{ijkl}(\boldsymbol{x}) = X(\boldsymbol{x}) \overline{\boldsymbol{E}}_{ijkl}$$

in terms of this mass distribution function and the constant rigidity tensor $\overline{\boldsymbol{E}}_{ijkl}$. The constant rigidity tensor is function of the material properties only. As X is a discrete function all admissible tensors are discrete and thus the optimization problem has a discrete valued parameter function.

The amount of work due of the deformation $\boldsymbol{u}$ can be calculated by cref{eq:LinLoad}. With the standard linearized strain formulation this results in,

$$l(\boldsymbol{u}) = \int_\Omega \boldsymbol{f}\boldsymbol{u} \, \mathrm{d}\Omega + \int_{\Gamma_T} \boldsymbol{t}\boldsymbol{u} \, \mathrm{d}\Gamma_T$$

A bi-linear energy equation with virtual work $a(\boldsymbol{u}, \hat{\boldsymbol{u}})$ is formulated,

$$a(\boldsymbol{u}, \hat{\boldsymbol{u}}) = \int_\Omega \boldsymbol{E}_{ijkl} \boldsymbol{\varepsilon}_{kl}(\boldsymbol{u}) \boldsymbol{\varepsilon}_{ij}(\hat{\boldsymbol{u}}) \, \mathrm{d}\Omega$$

[1] N. Olhoff and J. E. Taylor, "On Structural Optimization," J. Appl. Mech., vol. 50, no. 4b, p. 1139, 1983.

[2] M. P. Bendsøe, "Optimal shape design as a material distribution problem," Struct. Optim., vol. 1, no. 4, pp. 193–202, Dec. 1989.

$\hat{\boldsymbol{u}}$ is an arbitrary kinematically admissible deformation. Equilibrium is ensured when $l(\hat{\boldsymbol{u}}) = a(\boldsymbol{u}, \hat{\boldsymbol{u}})$ is satisfied for all admissible deformations $\hat{\boldsymbol{u}}$.

$$\begin{aligned} \min_{\Omega^m} \quad & u_{\text{out}} \\ \text{s.t.} : \quad & a(\boldsymbol{u}, \hat{\boldsymbol{u}}) = l(\hat{\boldsymbol{u}}) \\ & \int_\Omega X(\boldsymbol{x}) \mathrm{d}\Omega \;=\; \text{Vol}(\Omega^m) \;\leq\; V \end{aligned}$$

3.2.2 Discretization

To solve the continuum problem of the previous section it is discretized into a finite element analysis with N elements:

$$\begin{aligned} \min_{X_1, X_2, \ldots, X_N} \quad & c = \boldsymbol{l}^T \boldsymbol{u} \\ \text{s.t.} : \quad & \boldsymbol{K}\boldsymbol{u} = \boldsymbol{f} \\ & \sum_{e=1}^{N} v_e X_e \;\leq\; V \\ & X_e \in \{0, 1\} \quad \forall \quad e \in \{1, 2, \ldots, N\} \\ \text{where} : \quad & \boldsymbol{K} = \sum_{e=1}^{N} \boldsymbol{K}_e(X_e, \overline{E}) \end{aligned}$$

it shows that the element stiffness matrix $\boldsymbol{K_e}$ depends on the element material value X_e and the material stiffness $\overline{E}$. In this equation $\boldsymbol{l}$ is a vector filled with 0 except a -1 at the degree of freedom of whith the displacement is maximized (use 1 to minimize the displacement). The problem becomes unstable towards the element type and mesh when the discrete formulation of cref{eq:conti mass distribution,eq:stiffness_Lit} are used. Such a distribution problem generally has no solution[3],[4]. Iterative search methods would not work because they require the calculation of gradients. Therefore, the problem is changed so that the density becomes a continuous equation ranging from 0 to 1.

$$0 \leq X_e \leq 1$$

This method would result in a design with intermediate values. Although this makes sense for variable thickness plate design, see the work of M.P. Rossow and J.E. Taylor[5], for discrete topology design loses its direct physical representation. There is either material or there is not, intermediate values are meaningless. Changing cref{eq:stiffness_Lit} with a penalization that reduces the effectiveness of intermediate values results in a formulation that suppresses these intermediate values. The method used here, developed by E. Andreassen[6], is derived from the classical penalized proportional stiffness method (SIMP)[2],[7]. Here $E_{\min}$ is a small artificial stiffness used to avoid elements with zero

[3] G. Strang and R. V. Kohn, "Optimal design in elasticity and plasticity," Int. J. Numer. Methods Eng., vol. 22, no. 1, pp. 183–188, Jan. 1986.

[4] R. V. Kohn and G. Strang, "Optimal design and relaxation of variational problems, I," Commun. Pure Appl. Math., vol. 39, no. 1, pp. 113–137, 1986.

[5] M. P. Rossow and J. E. Taylor, "A Finite Element Method for the Optimal Design of Variable Thickness Sheets," AIAA J., vol. 11, no. 11, pp. 1566–1569, Nov. 1973.

[6] E. Andreassen, A. Clausen, M. Schevenels, B. S. Lazarov, and O. Sigmund, "Efficient topology optimization in MATLAB using 88 lines of code," Struct. Multidiscip. Optim., vol. 43, no. 1, pp. 1–16, Jan. 2011.

[7] M. P. Bendsøe and O. Sigmund, Topology Optimization. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

stiffness as that could make the FEA unstable.

$$\boldsymbol{E}_{ijkl}(\boldsymbol{x}) = \boldsymbol{E}_{ijkl,\min} + X(\boldsymbol{x})^p \left(\overline{\boldsymbol{E}}_{ijkl} - \boldsymbol{E}_{ijkl,\min}\right)$$

When $p > 1$ the intermediate density values are less effective as there stiffness is low in comparison to the volume occupied. When p is sufficiently large, generally $p \geq 3$, the design converges to a solution that is close to a discrete (0-1) design.

3.2.3 Sensitivity analysis

The gradient of one element in the discretized form is $\partial u_{\text{out}}/\partial X_e$. This derivative has to be calculated explicitly as the problem is not self adjoint. The derivation starts with a new formulation of the displacement, the difference is the zero term at the end. Here $\boldsymbol{\lambda}$ this is a arbitrary admissible deformation. This is similar to what $\hat{\boldsymbol{u}}$ would be for the global compliance case, here the symbol $\boldsymbol{\lambda}$ because the adoint problem will link it to the vector $\boldsymbol{l}$[2].

$$u_{\text{out}} = \boldsymbol{l}^T\boldsymbol{u} - \boldsymbol{\lambda}^T\left(\boldsymbol{K}\boldsymbol{u} - \boldsymbol{f}\right)$$

taking the derivative to the density leads to:

$$\frac{\partial u_{\text{out}}}{\partial X_e} = \left(\boldsymbol{l}^T - \boldsymbol{\lambda}^T\boldsymbol{K}\right)\frac{\partial \boldsymbol{u}}{\partial X_e} - \boldsymbol{\lambda}^T\frac{\partial \boldsymbol{K}}{\partial X_e}\boldsymbol{u}$$

when $\boldsymbol{\lambda}$ satisfies the adjoint equation it becomes:

$$\begin{aligned}\frac{\partial u_{\text{out}}}{\partial X_e} =& -\boldsymbol{\lambda}^T\frac{\partial \boldsymbol{K}}{\partial X_e}\boldsymbol{u} \\ &\text{when} \quad \boldsymbol{l}^T - \boldsymbol{\lambda}^T\boldsymbol{K} = 0\end{aligned}$$

Satisfying this adjoint equation is simple, just solve $\boldsymbol{K}\boldsymbol{\lambda} = \boldsymbol{l}$. The derivative of the stiffness matrix to the density of an element can be derived leading to the final expression of the gradient:

$$\frac{\partial u_{\text{out}}}{\partial X_e} = -pX_e^{p-1}\boldsymbol{\lambda}^T\boldsymbol{K}_e\boldsymbol{u}$$

3.2.4 Computational Implementation

The iterative implementation of topology optimization as proposed by M. Beckers,[8] or M.P. Bendsøe and O. Sigmund[2] are similar. It exists out of three parts, initialization, optimization and post processing. The flowchart of the local compliance algorithm can be found in Fig. 3.2.1.

Fig. 3.2.1: Flowchart for local compliance maximization[7].

In the initialization phase the problem is set up. It defines the design domain, the loading conditions, the initial design and generates the finite element mesh that will be used in the optimization phase.

The optimization phase is the iterative method that solves the topology problem. It will analyze the current design with a FEA. After which it will calculate the sensitivity of the local compliance to the density of each element, this is the local gradient of which the calculation is discussed in *Sensitivity analysis and MMA*. The Method of Moving

[8] M. Beckers, "Topology optimization using a dual method with discrete variables," Struct. Optim., vol. 17, no. 1, pp. 14–24, Feb. 1999.

Asymptotes (MMA), developed by K. Svanberg[9], is used to formulate a simplified convex approximation of the problem which is optimized to formulate the updated design. These steps are performed in a loop until the design is converged, i.e. when the change in design between two iterations becomes negligible.

Post processing is required to remove the last elements with intermediate values and generate a shape out of the design, for example a CAD or STL file. This algorithm will not contain any of the post processing steps. The code used in this communication simply plots the final shape and load case.

3.2.5 Example and Results

Example code and results!!!!!!!!!!!!!!!!!

3.2.6 References

3.3 Stress Intensity Factor Minimization

The objective of the research was to explore how topology optimization can be used to optimized for damage tolerance objectives such as fatigue crack growth rate. It was hypothesized that the difficulties would lay in the formulation an objective function and the adjoint equation. There formulation should be based upon linear fracture mechanics with the use of FEA.

- *Continuum formulation*
- *Discretisation*
- *Sensitivity analysis*
- *Computational implementation*
- *Examples and Results*
- *References*

3.3.1 Continuum formulation

The problem formulation, required for optimization problems, should contain the optimization objective, its link to the design variables and the constraints. Because the goal is design a geometry with the lowest crack growth rate and the Paris-Erdogan law[1] minimizing stress intensity factor K_I was chosen as the objective. Due to this formulation the design geometry, topology, is the optimization variable.

Fig. 3.3.1: Design domain Ω with a crack, arbitrary boundary conditions and a density X which is dependent on the position vector $\boldsymbol{x}$.

[9] K. Svanberg, "The method of moving asymptotes - a new method for structural optimization," Int. J. Numer. Methods Eng., vol. 24, no. 2, pp. 359–373, Feb. 1987.

[1] P. C. Paris and F. Erdogan, "A Critical Analysis of Crack Propagation Laws," J. Basic Eng., vol. 85, no. 4, p. 528, 1963.

Assuming a general problem, shown in Fig. 3.3.1, which minimizes the stress intensity by changing the material distribution, $X(\boldsymbol{x})$ within the design domain Ω, the following mathematical formulation is proposed,

$$\begin{aligned} \min_{X(\boldsymbol{x})} \quad & K_I(X(\boldsymbol{x})) \\ \text{s.t. :} \quad & a(\boldsymbol{u}(X(\boldsymbol{x})), \hat{\boldsymbol{u}}) = l(\hat{\boldsymbol{u}}) \\ & \int_{\Omega} X(\boldsymbol{x}) \, \mathrm{d}\Omega \;=\; \mathrm{Vol}(\Omega^m) \;\leq\; V \\ & X_{\min} \leq X(\boldsymbol{x}) \leq X_{\max} \end{aligned}$$

it enforces equilibrium with a virtual work method while the problem is subjected to a resource constraint. This constraint limits the volume within the design domain that can be filled with a material beside setting a minimum and maximum density value.

For any optimization a link between the objective and the design variables must be made. The method proposed here can be used for two cases, variable thickness plate and discrete material distribution. The honeycomb infill problem is a type of discrete material distribution and will not be discussed separately. In the first case the optimization variables X are interpreted as the local plate thickness. As the thickness influences the local stiffness properties it affects the stress intensity values at the crack tip. For this variable thickness sheet a linear relation,

$$\boldsymbol{E}_{ijkl}(\boldsymbol{x}) = \boldsymbol{E}_{ijkl,\min} + X(\boldsymbol{x}) \left(\overline{\boldsymbol{E}}_{ijkl} - \boldsymbol{E}_{ijkl,\min} \right)$$

between local stiffness and thickness is used. This equation was proposed by M.P. Rossow and J.E. Taylor[2] and discussed by O. Sigmund[3], and causes the stiffness to become twice as high when the thickness is doubled. Here $\overline{\boldsymbol{E}}_{ijkl}$ is a constant stiffness tensor related to the material it unity thickness while $\boldsymbol{E}_{ijkl,\min}$ a tensor is with very small stiffness. Which enforces the total stiffness to be larger than zero. One cannot allow the stiffness to become zero as it would cause the FEA to fail. This relation might be inaccurate due to out of plane effects at thickness changes and it will be necessary to measure under what circumstances this equation is invalid.

When the goal is to obtain a discrete design the density values can be either 0 (no material) or 1 (material). This however causes the objective equation to become discrete as well as the method used a gradient approach and requires a continuous function of density. To ensure a discrete final design while maintaining a continuous objective function a penalization method was implemented. The method used was based upon the penalized proportional stiffness method (SIMP),

$$\boldsymbol{E}_{ijkl}(\boldsymbol{x}) = \boldsymbol{E}_{ijkl,\min} + X(\boldsymbol{x})^p \left(\overline{\boldsymbol{E}}_{ijkl} - \boldsymbol{E}_{ijkl,\min} \right)$$

it causes designs to converge to a 0-1 solution when the penalty factor p is chosen sufficiently high. Values of $p \geq 3$ are required for designs to become discrete.

3.3.2 Discretisation

The previous section linked the design variables to the stiffness distribution no official formulation of the stress intensity factors in terms of design variables was made. This formulation is indirectly made through the equilibrium constraint as stiffness distribution influences the stress/displacement field of the loaded part, these stress/displacement distribution can be related to the stress intensity factor. The original equilibrium equation is in a continuum formulation but to simplify the problem a discretized version will be solved using FEA.

2 M. P. Rossow and J. E. Taylor, "A Finite Element Method for the Optimal Design of Variable Thickness Sheets," AIAA J., vol. 11, no. 11, pp. 1566–1569, Nov. 1973.

3 O. Sigmund, N. Aage, and E. Andreassen, "On the (non-)optimality of Michell structures," Struct. Multidiscip. Optim., vol. 54, no. 2, pp. 361–373, 2016.

To ensure a direct and efficient calculation of the stress intensity factor while using a finite element analysis an enrichment method was used for elements close to the crack tip. The method used was developed by S.E. Benzley[4] and improved by L.N. Gifford[5]. It uses a linear summation of a continuous displacement field and a near crack tip displacement field capturing both the discrete behavior at the crack tip and the continuous one around it. The discrete solution was derived with the Westergaard function method[6]. This type of tip element enrichment allows accurate predictions of stress intensity directly from the FEA without any post processing as it can be found in the displacement vector.

Crack tip element

The method uses special elements around the crack tip of which the stiffness matrix needs to be derived. As these enriched elements based upon an addition of the continuous and singularity displacement field these are discussed separately at first.

Fig. 3.3.2: Nodal definition of the crack tip element.

The enrichment method shown here was based upon the crack tip element developed my L.N. Gifford[5]. Who based the enriched elements on a bicubic serendipity elements, see Fig. 3.3.2. The algorithm presented here keeps the local coordinate system $(\xi,\, \eta)$ as only a regular mesh with square elements will be used. For a more general element that can contain cracks under an angle and that transforms elements from $(\xi,\, \eta)$ to $(x,\, y)$ see the original paper[5].

The displacement field within the bicubic serendipity 12-node element can be described by:

$$\boldsymbol{u} = \sum_{i=0}^{11} N^i(\xi,\, \eta)\boldsymbol{u}^i$$

[4] S. E. Benzley, "Representation of singularities with isoparametric finite elements," Int. J. Numer. Methods Eng., vol. 8, no. 3, pp. 537–545, 1974.

[5] L. Nash Gifford and P. D. Hilton, "Stress intensity factors by enriched finite elements," Eng. Fract. Mech., vol. 10, no. 3, pp. 485–496, Jan. 1978.

[6] H. M. Westergaard, "Bearing pressures and cracks," J. Appl. Mech., vol. 6, pp. A49-53, 1939.

where the shape functions N^i are,

$$
\begin{aligned}
N^0 &= \frac{1}{32}(1-\eta)(1-\xi)\left(9\eta^2+9\xi^2-10\right)\\
N^1 &= \frac{9}{32}(1-\eta)(1-3\xi)\left(1-\xi^2\right)\\
N^2 &= \frac{9}{32}(1-\eta)(1+3\xi)\left(1-\xi^2\right)\\
N^3 &= \frac{1}{32}(1-\eta)(1+\xi)\left(9\eta^2+9\xi^2-10\right)\\
N^4 &= \frac{9}{32}(1-3\eta)(1+\xi)\left(1-\eta^2\right)\\
N^5 &= \frac{9}{32}(1+3\eta)(1+\xi)\left(1-\eta^2\right)\\
N^6 &= \frac{1}{32}(1+\eta)(1+\xi)\left(9\eta^2+9\xi^2-10\right)\\
N^7 &= \frac{9}{32}(1+\eta)(1+3\xi)\left(1-\xi^2\right)\\
N^8 &= \frac{9}{32}(1+\eta)(1-3\xi)\left(1-\xi^2\right)\\
N^9 &= \frac{1}{32}(1+\eta)(1-\xi)\left(9\eta^2+9\xi^2-10\right)\\
N^{10} &= \frac{9}{32}(1+3\eta)(1-\xi)\left(1-\eta^2\right)\\
N^{11} &= \frac{9}{32}(1-3\eta)(1-\xi)\left(1-\eta^2\right)
\end{aligned}
$$

Added to this will be the crack tip singularity displacement field which derivation starts from the definition of stress intensity factors in a simplified 2D space,

$$
\begin{aligned}
K_I &= \lim_{r\to 0}\sqrt{2\pi r}\sigma_{xx}\\
K_{II} &= \lim_{r\to 0}\sqrt{2\pi r}\sigma_{xy}
\end{aligned}
$$

and the crack tip stresses derived with the Westergaard method[6],

$$
\begin{aligned}
\sigma_{xx} =& \frac{K_I}{\sqrt{2\pi r}}\cos\frac{\theta}{2}\left(1-\sin\frac{\theta}{2}\sin\frac{3\theta}{2}\right)\\
&-\frac{K_{II}}{\sqrt{2\pi r}}\sin\frac{\theta}{2}\left(2+\cos\frac{\theta}{2}\cos\frac{3\theta}{2}\right)\\
\sigma_{yy} =& \frac{K_I}{\sqrt{2\pi r}}\cos\frac{\theta}{2}\left(1+\sin\frac{\theta}{2}\sin\frac{3\theta}{2}\right)\\
&+\frac{K_{II}}{\sqrt{2\pi r}}\cos\frac{\theta}{2}\sin\frac{\theta}{2}\cos\frac{3\theta}{2}\\
\tau_{xy} =& \frac{K_I}{\sqrt{2\pi r}}\cos\frac{\theta}{2}\sin\frac{\theta}{2}\cos\frac{3\theta}{2}\\
&+\frac{K_{II}}{\sqrt{2\pi r}}\cos\frac{\theta}{2}\left(1-\sin\frac{\theta}{2}\sin\frac{3\theta}{2}\right)
\end{aligned}
$$

which are accurate approximations of the stresses close to the crack tip, i.e. r is small. Fig. 3.3.3 shows the axis system definition for the calculation around the crack tip.

Fig. 3.3.3: Definition of the axis systems around the crack tip.

A formulation of the displacement field can be found by integration leading to,

$$\begin{aligned}
u_x =& K_I f_x(r,\,\theta) + K_{II} g_x(r,\,\theta) \\
=& \frac{K_I}{4G}\sqrt{\frac{r}{2\pi}}\left(-1+\gamma-2\sin^2\frac{\theta}{2}\right)\cos\frac{\theta}{2} \\
&+ \frac{K_{II}}{4G}\sqrt{\frac{r}{2\pi}}\left(1+\gamma+2\cos^2\frac{\theta}{2}\right)\sin\frac{\theta}{2} \\
u_y =& K_I f_y(r,\,\theta) + K_{II} g_y(r,\,\theta) \\
=& \frac{K_I}{4G}\sqrt{\frac{r}{2\pi}}\left(1+\gamma+2\cos^2\frac{\theta}{2}\right)\sin\frac{\theta}{2} \\
&+ \frac{K_{II}}{4G}\sqrt{\frac{r}{2\pi}}\left(1-\gamma+2\sin^2\frac{\theta}{2}\right)\cos\frac{\theta}{2}
\end{aligned}$$

where $\gamma = (3-\nu)/(1+\nu)$ for plane stress and $\gamma = 3-4\nu$ for plane strain[7]. When assuming linear fracture mechanics one can describe the displacement field of this element as summation of the continuums and the singularity displacement fields resulting in:

$$\begin{aligned}
u_x &= K_I f_x(r,\,\theta) + K_{II} g_x(r,\,\theta) + \sum N^i(\xi,\,\eta) u_x^i \\
u_y &= K_I f_y(r,\,\theta) + K_{II} g_y(r,\,\theta) + \sum N^i(\xi,\,\eta) u_y^i
\end{aligned}$$

The singularity equations need to be transformed from the $(r,\,\theta)$ axis into the local $(\xi,\,\eta)$ system. This transformation is dependent of the relative location of the crack tip to the local element axis system.

The enriched displacement functions can cause discontinuities at the border to normal elements, this can be repaired by multiplying the enrichment terms of the displacement function with an equation that is 1 at the crack tip and 0 at the border to non enriched elements[4]. It has however been reported that the effects of discontinuities are minor and this solution was therefore not implemented[5].

Following a definition of FE by Zienkiewicz[8] an element stiffness matrix can be calculated with,

$$\boldsymbol{K} = \int_{-1}^{1}\int_{-1}^{1} \boldsymbol{B}^T \boldsymbol{D} \boldsymbol{B}\ \det \boldsymbol{J}\ \mathrm{d}\xi \mathrm{d}\eta$$

where $\boldsymbol{D}$ the material stiffness matrix is, $\boldsymbol{J}$ the Jacobian of axis system transformation $(\xi,\,\eta)$ into the global $(x,\,y)$ axis system is and $\boldsymbol{B}$ the matrix is that converts displacement into strain. The integration was performed with a Gauss-Legendre quadrature function with 8x8 integration points as was found sufficient by L.N. Gifford[5].

For a standard bicubic serendipity element this $\boldsymbol{B}$ matrix is of shape $(3,\,24)$ however due to the enrichment it becomes $(3,\,26)$. Which results in a final stiffness matrix of $(26,\,26)$. Where

$$\boldsymbol{f} = \boldsymbol{K}\boldsymbol{u} = \begin{pmatrix} f_x^0 \\ \vdots \\ f_x^* \\ f_y^* \end{pmatrix} = \begin{bmatrix} \boldsymbol{k} & \vdots & \boldsymbol{k}_{12} \\ \dots & \vdots & \dots \\ \boldsymbol{k}_{21} & \vdots & \boldsymbol{k}_{22} \end{bmatrix} \begin{pmatrix} u_x^0 \\ \vdots \\ K_I \\ K_{II} \end{pmatrix}$$

[7] A. F. Bower, "Modeling Material Failure," in Applied Mechanics of Solids, 1st ed., Baton Rouge (LA): CRC Press, 2009, pp. 569.

[8] O. C. Zienkiewicz, The Finite Element Method In Engineering Science. New York (NY): McGraw-Hill, 1971.

Here $\boldsymbol{k}$ is similar to the stiffness matrix of a normal bicubic element, the enrichment is in the parts $\boldsymbol{k}_{12}$, $\boldsymbol{k}_{21}$ and $\boldsymbol{k}_{22}$. New terms do also appear in the force vector, where f_x^* and f_y^* are so-called singular loads. They describe the external forces applied on the crack boundary[4], in general these values are zero.

Meshing strategy

To reduce computational costs these enriched elements are only used at the crack tip and conventional linear elements are used throughout the rest of the mesh. It uses the hanging node method to connect the elements as can be seen in Fig. 3.3.4.

Fig. 3.3.4: Top section of mesh around a crack tip, $\oplus$ is the enrichment node with K_I and K_{II}, while solid circles represent the linear ones and the open circle the higher order ones.

This mesh is not conform which can potentially cause the displacement field to become discontinuous. To avoid this one could use normal bicubic serendipity elements throughout the entire mesh which is computational inefficient. However, using a multi-resolution interpretation of topology optimization its performance might be improved[9].

Currently the linear system of the FEA, $\boldsymbol{f} = \boldsymbol{K}\boldsymbol{u}$, and the adjoint equation, $\boldsymbol{l} = \boldsymbol{K}\boldsymbol{\lambda}$, are solved with a complete Cholesky decomposition. A more efficient methods can be formulated with a Multi Grid Conjugate Gradient method as proposed by O. Amir[10].

Objective formulation

As a spacial discretized method (FEA) was used to calculate the objective the problem formulation needs to become discretized as well. For a mesh of N elements the optimization objective becomes;

$$
\begin{aligned}
\min_{X_1, X_2, \ldots, X_N} \quad & K_I = \boldsymbol{l}^T \boldsymbol{u} \\
\text{s.t. :} \quad & \boldsymbol{K}\boldsymbol{u} = \boldsymbol{f} \\
& \sum_{e=1}^{N} v_e X_e \ \leq \ V \\
& X_{\min} \leq X_e \leq X_{\max} \ \ \forall \ \ e \in \{1, 2, \ldots, N\} \\
\text{where :} \quad & \boldsymbol{K} = \sum_{e=1}^{N} \boldsymbol{K}_e(X_e, \overline{E})
\end{aligned}
$$

which minimizes the stress intensity factor while ensuring equilibrium and setting constraints to the density distribution. Here $\boldsymbol{u}$ is the enriched displacement vector, f the force vector and v_e is the (relative) element volume. $\boldsymbol{l}$ is zero vector except for the degree of freedom linked to the stress intensity factor, and the multiplication of $\boldsymbol{l}^T\boldsymbol{u}$ will return the stress intensity factor. This is similar to the compliant mechanism optimization mentioned by O. Sigmund[11] where the displacement of a specific degree of freedom is maximized.

9 J. P. Groen, M. Langelaar, O. Sigmund, and M. Ruess, "Higher-order multi-resolution topology optimization using the finite cell method," Int. J. Numer. Methods Eng., vol. 110, no. 10, pp. 903–920, Jun. 2017.

10 O. Amir, N. Aage, and B. S. Lazarov, "On multigrid-CG for efficient topology optimization," Struct. Multidiscip. Optim., vol. 49, no. 5, pp. 815–829, May 2014.

11 O. Sigmund, "On the design of compliant mechanisms using topology optimization," Mech. Struct. Mach., vol. 25, no. 4, pp. 493–524, 1997.

3.3.3 Sensitivity analysis

The local convex approximation requires the calculation of the sensitivity of K_I to a density change in any element. This can be measured by $\partial K_I / \partial X_e$, which can be calculated with the following steps and starts with adding a zero term after the known function $K_I = \boldsymbol{l}^T \boldsymbol{u}$, where $\boldsymbol{\lambda}$ is an arbitrary vector:

$$K_I = \boldsymbol{l}^T \boldsymbol{u} - \boldsymbol{\lambda}^T (\boldsymbol{K}\boldsymbol{u} - \boldsymbol{f})$$

$$\frac{\partial K_I}{\partial X_e} = \left(\boldsymbol{l}^T - \boldsymbol{\lambda}^T \boldsymbol{K}\right) \frac{\partial \boldsymbol{u}}{\partial X_e} - \boldsymbol{\lambda}^T \frac{\partial \boldsymbol{K}}{\partial X_e} \boldsymbol{u}$$

Now choosing a convenient vector for $\boldsymbol{\lambda}$ which causes $\boldsymbol{l}^T - \boldsymbol{\lambda}^T \boldsymbol{K}$ to be zero leads to the following expression for the sensitivity,

$$\frac{\partial K_I}{\partial X_e} = - \boldsymbol{\lambda}^T \frac{\partial \boldsymbol{K}}{\partial X_e} \boldsymbol{u}$$

$$\text{where:} \qquad \boldsymbol{l} = \boldsymbol{K}\boldsymbol{\lambda}$$

This means that $\boldsymbol{\lambda}$ can be calculated with the FEA, where $\boldsymbol{l}$ is seen as a sort force vector, by solving $\boldsymbol{l} = \boldsymbol{K}\boldsymbol{u}$. The sensitivity of $\boldsymbol{K}$ to the element density can be calculated, resulting in the following gradient:

$$\frac{\partial K_I}{\partial X_e} = -p X_e^{p-1} \boldsymbol{\lambda}^T \boldsymbol{K}_e \boldsymbol{u}$$

3.3.4 Computational implementation

The iterative implementation of topology optimization as proposed by M. Beckers,[8] or M.P. Bendsøe and O. Sigmund[2] are similar. It exists out of three parts, initialization, optimization and post processing. The flowchart of the local compliance algorithm can be found in Fig. 3.3.5.

Fig. 3.3.5: Flowchart for fatigue crack growth rate minimization[7].

In the initialization phase the problem is set up. It defines the design domain, the loading conditions, the initial design and generates the finite element mesh that will be used in the optimization phase.

The optimization phase is the iterative method that solves the topology problem. It will analyze the current design with a FEA. After which it will calculate the sensitivity of the stress intensity factor to the density of each element, this is the local gradient of which the calculation is discussed in *Sensitivity analysis and MMA*. The Method of Moving Asymptotes (MMA), developed by K. Svanberg[9], is used to formulate a simplified convex approximation of the problem which is optimized to formulate the updated design. These steps are performed in a loop until the design is converged, i.e. when the change in design between two iterations becomes negligible.

Post processing is required to remove the last elements with intermediate values and generate a shape out of the design, for example a CAD or STL file. This algorithm will not contain any of the post processing steps. The code used in this communication simply plots the final shape and load case.

3.3.5 Examples and Results

3.3.6 References

3.4 Fatigue Crack Growth Life Maximization

The objective of the research was to explore how topology optimization can be used to optimized for damage tolerance objectives such as fatigue crack growth life (FCGL). It was hypothesized that the difficulties would lay in the

formulation an objective function and the adjoint equation. There formulation should be based upon linear fracture mechanics combining the Paris rule and FEA.

- *Continuum formulation*
- *Discretisation*
- *Sensitivity Analysis*
- *Computational implementation*
- *Exaples and results*
- *References*

3.4.1 Continuum formulation

The problem formulation, required for optimization problems, should contain the optimization objective, its link to the design variables and the constraints.

Fig. 3.4.1: Design domain Ω with a crack, arbitrary boundary conditions and a density X which is dependent on the position vector $\boldsymbol{x}$.

Because the goal is design a geometry with the most crack growth cycles and uses the Paris-Erdogan rule $da/dN = CK_I^m$[1]. Due to this formulation the design geometry, is the optimization objective was formulated as an integral, $N = \int 1/(da/dN)da$. This integral is only valid in the socalled Paris region, hence the integral starts at $a_0 > 0$ and ends it ends at a chosen maximum length a_{end}. This a_{end} should be a crack length that can be observed during inspection while it is not long enougth for failure. Assuming a general problem, shown in Fig. 3.4.1, which maximized the FCGL by changing the material distribution, $X(\boldsymbol{x})$ within the design domain Ω, the following mathematical formulation is proposed,

$$
\begin{aligned}
\min_{X(\boldsymbol{x})} \quad & N(X(\boldsymbol{x})) = \int_{a_0}^{a_{\text{end}}} \frac{1}{C} \frac{1}{K_I(X(\boldsymbol{x}), a)}^m \mathrm{d}a \\
\text{s.t.}: \quad & a(\boldsymbol{u}(X(\boldsymbol{x})), \hat{\boldsymbol{u}}) = l(\hat{\boldsymbol{u}}) \\
& \int_\Omega X(\boldsymbol{x}) \, \mathrm{d}\Omega \;=\; \text{Vol}(\Omega^m) \;\leq\; V \\
& X_{\min} \leq X(\boldsymbol{x}) \leq X_{\max}
\end{aligned}
$$

it enforces equilibrium with a virtual work method while the problem is subjected to a resource constraint. This constraint limits the volume within the design domain that can be filled with a material beside setting a minimum and maximum density value.

For any optimization a link between the objective and the design variables must be made. The method proposed here can be used for two cases, variable thickness plate and discrete material distribution. The honeycomb infill problem is a type of discrete material distribution and will not be discussed separately. In the first case the optimization variables X are interpreted as the local plate thickness. As the thickness influences the local stiffness properties it affects the stress intensity values at the crack tip, these stress intensity factors are related to the crack growth rate. Hence the thickness does influece the amount of load cycles required for the crack to grow from math:*a_0* to math:*a_{text{end}}*. For this variable thickness sheet a linear relation,

$$
\boldsymbol{E}_{ijkl}(\boldsymbol{x}) = \boldsymbol{E}_{ijkl,\min} + X(\boldsymbol{x}) \left(\overline{\boldsymbol{E}}_{ijkl} - \boldsymbol{E}_{ijkl,\min} \right)
$$

[1] P. C. Paris and F. Erdogan, "A Critical Analysis of Crack Propagation Laws," J. Basic Eng., vol. 85, no. 4, p. 528, 1963.

between local stiffness and thickness is used. This equation was proposed by M.P. Rossow and J.E. Taylor[2] and discussed by O. Sigmund[3], and causes the stiffness to become twice as high when the thickness is doubled. Here $\overline{\boldsymbol{E}}_{ijkl}$ is a constant stiffness tensor related to the material it unity thickness while $\boldsymbol{E}_{ijkl,\min}$ a tensor is with very small stiffness. Which enforces the total stiffness to be larger than zero. One cannot allow the stiffness to become zero as it would cause the FEA to fail. This relation might be inaccurate due to out of plane effects at thickness changes and it will be necessary to measure under what circumstances this equation is invalid.

When the goal is to obtain a discrete design the density values can be either 0 (no material) or 1 (material). This however causes the objective equation to become discrete as well as the method used a gradient approach and requires a continuous function of density. To ensure a discrete final design while maintaining a continuous objective function a penalization method was implemented. The method used was based upon the penalized proportional stiffness method (SIMP),

$$\boldsymbol{E}_{ijkl}(\boldsymbol{x}) = \boldsymbol{E}_{ijkl,\min} + X(\boldsymbol{x})^p \left(\overline{\boldsymbol{E}}_{ijkl} - \boldsymbol{E}_{ijkl,\min}\right)$$

it causes designs to converge to a 0-1 solution when the penalty factor p is chosen sufficiently high. Values of $p \geq 3$ are required for designs to become discrete.

3.4.2 Discretisation

The previous section linked the design variables to the stiffness distribution no official formulation of the FCGL in terms of design variables was made. This formulation is indirectly made through the equilibrium constraint as stiffness distribution influences the stress/displacement field of the loaded part, these stress/displacement distribution can be related to the stress intensity factor and the fatigue crack growth rate. The original equilibrium equation is in a continuum formulation but to simplify the problem a discretized version will be solved using FEA.

To calculate the FCGL one has to use the Paris rule resulting in:

$$N(X(\boldsymbol{x})) = \int_{a_0}^{a_{\text{end}}} \frac{1}{C} \frac{1}{K_I(X(\boldsymbol{x}), a)}^m \mathrm{d}a$$

K_I is dependend on the design variables $X(\boldsymbol{x})$, both C and m can be interpetated as material constants. Notice that K_I is also depending on the actual crack length (a), hence the integral is replaced by the following discrete summation,

$$N(X(\boldsymbol{x})) = \frac{1}{C} \sum_{l=1}^{L-1} \frac{(a_{l+1} + a_l)}{\left(\frac{1}{2}\left(K_I(X(\boldsymbol{x}), a_{l+1}) + K_I(X(\boldsymbol{x}), a_l)\right)\right)^m}$$

to compute this summation L different values for K_I have to be computed each with a different crack length. To ensure a direct and efficient calculation of the stress intensity factor while using a finite element analysis an enrichment method was used for elements close to the crack tip. The method used was developed by S.E. Benzley[4] and improved by L.N. Gifford[5]. It uses a linear summation of a continuous displacement field and a near crack tip displacement

2 M. P. Rossow and J. E. Taylor, "A Finite Element Method for the Optimal Design of Variable Thickness Sheets," AIAA J., vol. 11, no. 11, pp. 1566–1569, Nov. 1973.

3 O. Sigmund, N. Aage, and E. Andreassen, "On the (non-)optimality of Michell structures," Struct. Multidiscip. Optim., vol. 54, no. 2, pp. 361–373, 2016.

4 S. E. Benzley, "Representation of singularities with isoparametric finite elements," Int. J. Numer. Methods Eng., vol. 8, no. 3, pp. 537–545, 1974.

5 L. Nash Gifford and P. D. Hilton, "Stress intensity factors by enriched finite elements," Eng. Fract. Mech., vol. 10, no. 3, pp. 485–496, Jan. 1978.

field capturing both the discrete behavior at the crack tip and the continuous one around it. The discrete solution was derived with the Westergaard function method[6]. This type of tip element enrichment allows accurate predictions of stress intensity directly from the FEA without any post processing as it can be found in the displacement vector.

Crack tip element

The method uses special elements around the crack tip of which the stiffness matrix needs to be derived. As these enriched elements based upon an addition of the continuous and singularity displacement field these are discussed separately at first.

Fig. 3.4.2: Nodal definition of the crack tip element.

The enrichment method shown here was based upon the crack tip element developed my L.N. Gifford[5]. Who based the enriched elements on a bicubic serendipity elements, see Fig. 3.4.2. The algorithm presented here keeps the local coordinate system $(\xi,\, \eta)$ as only a regular mesh with square elements will be used. For a more general element that can contain cracks under an angle and that transforms elements from $(\xi,\, \eta)$ to $(x,\, y)$ see the original paper[5].

The displacement field within the bicubic serendipity 12-node element can be described by:

$$\boldsymbol{u} = \sum_{i=0}^{11} N^i(\xi,\, \eta)\boldsymbol{u}^i$$

where the shape functions N^i are,

$$\begin{aligned}
N^0 &= \frac{1}{32}\,(1-\eta)\,(1-\xi)\,\left(9\eta^2+9\xi^2-10\right)\\
N^1 &= \frac{9}{32}\,(1-\eta)\,(1-3\xi)\,\left(1-\xi^2\right)\\
N^2 &= \frac{9}{32}\,(1-\eta)\,(1+3\xi)\,\left(1-\xi^2\right)\\
N^3 &= \frac{1}{32}\,(1-\eta)\,(1+\xi)\,\left(9\eta^2+9\xi^2-10\right)\\
N^4 &= \frac{9}{32}\,(1-3\eta)\,(1+\xi)\,\left(1-\eta^2\right)\\
N^5 &= \frac{9}{32}\,(1+3\eta)\,(1+\xi)\,\left(1-\eta^2\right)\\
N^6 &= \frac{1}{32}\,(1+\eta)\,(1+\xi)\,\left(9\eta^2+9\xi^2-10\right)\\
N^7 &= \frac{9}{32}\,(1+\eta)\,(1+3\xi)\,\left(1-\xi^2\right)\\
N^8 &= \frac{9}{32}\,(1+\eta)\,(1-3\xi)\,\left(1-\xi^2\right)\\
N^9 &= \frac{1}{32}\,(1+\eta)\,(1-\xi)\,\left(9\eta^2+9\xi^2-10\right)\\
N^{10} &= \frac{9}{32}\,(1+3\eta)\,(1-\xi)\,\left(1-\eta^2\right)\\
N^{11} &= \frac{9}{32}\,(1-3\eta)\,(1-\xi)\,\left(1-\eta^2\right)
\end{aligned}$$

[6] H. M. Westergaard, "Bearing pressures and cracks," J. Appl. Mech., vol. 6, pp. A49-53, 1939.

Added to this will be the crack tip singularity displacement field which derivation starts from the definition of stress intensity factors in a simplified 2D space,

$$K_I = \lim_{r\to 0} \sqrt{2\pi r}\sigma_{xx}$$
$$K_{II} = \lim_{r\to 0} \sqrt{2\pi r}\sigma_{xy}$$

and the crack tip stresses derived with the Westergaard method[6],

$$\begin{aligned}
\sigma_{xx} =& \frac{K_I}{\sqrt{2\pi r}} \cos\frac{\theta}{2}\left(1 - \sin\frac{\theta}{2}\sin\frac{3\theta}{2}\right) \\
& - \frac{K_{II}}{\sqrt{2\pi r}} \sin\frac{\theta}{2}\left(2 + \cos\frac{\theta}{2}\cos\frac{3\theta}{2}\right) \\
\sigma_{yy} =& \frac{K_I}{\sqrt{2\pi r}} \cos\frac{\theta}{2}\left(1 + \sin\frac{\theta}{2}\sin\frac{3\theta}{2}\right) \\
& + \frac{K_{II}}{\sqrt{2\pi r}} \cos\frac{\theta}{2}\sin\frac{\theta}{2}\cos\frac{3\theta}{2} \\
\tau_{xy} =& \frac{K_I}{\sqrt{2\pi r}} \cos\frac{\theta}{2}\sin\frac{\theta}{2}\cos\frac{3\theta}{2} \\
& + \frac{K_{II}}{\sqrt{2\pi r}} \cos\frac{\theta}{2}\left(1 - \sin\frac{\theta}{2}\sin\frac{3\theta}{2}\right)
\end{aligned}$$

which are accurate approximations of the stresses close to the crack tip, i.e. r is small. Fig. 3.4.3 shows the axis system definition for the calculation around the crack tip.

Fig. 3.4.3: Definition of the axis systems around the crack tip.

A formulation of the displacement field can be found by integration leading to,

$$\begin{aligned}
u_x =& K_I f_x(r,\,\theta) + K_{II} g_x(r,\,\theta) \\
=& \frac{K_I}{4G}\sqrt{\frac{r}{2\pi}}\left(-1 + \gamma - 2\sin^2\frac{\theta}{2}\right)\cos\frac{\theta}{2} \\
& + \frac{K_{II}}{4G}\sqrt{\frac{r}{2\pi}}\left(1 + \gamma + 2\cos^2\frac{\theta}{2}\right)\sin\frac{\theta}{2} \\
u_y =& K_I f_y(r,\,\theta) + K_{II} g_y(r,\,\theta) \\
=& \frac{K_I}{4G}\sqrt{\frac{r}{2\pi}}\left(1 + \gamma + 2\cos^2\frac{\theta}{2}\right)\sin\frac{\theta}{2} \\
& + \frac{K_{II}}{4G}\sqrt{\frac{r}{2\pi}}\left(1 - \gamma + 2\sin^2\frac{\theta}{2}\right)\cos\frac{\theta}{2}
\end{aligned}$$

where $\gamma = (3-\nu)/(1+\nu)$ for plane stress and $\gamma = 3 - 4\nu$ for plane strain[7]. When assuming linear fracture mechanics one can describe the displacement field of this element as summation of the continuums and the singularity displacement fields resulting in:

$$u_x = K_I f_x(r,\,\theta) + K_{II} g_x(r,\,\theta) + \sum N^i(\xi,\,\eta) u_x^i$$
$$u_y = K_I f_y(r,\,\theta) + K_{II} g_y(r,\,\theta) + \sum N^i(\xi,\,\eta) u_y^i$$

[7] A. F. Bower, "Modeling Material Failure," in Applied Mechanics of Solids, 1st ed., Baton Rouge (LA): CRC Press, 2009, pp. 569.

The singularity equations need to be transformed from the $(r,\ \theta)$ axis into the local $(\xi,\ \eta)$ system. This transformation is dependent of the relative location of the crack tip to the local element axis system.

The enriched displacement functions can cause discontinuities at the border to normal elements, this can be repaired by multiplying the enrichment terms of the displacement function with an equation that is 1 at the crack tip and 0 at the border to non enriched elements[4]. It has however been reported that the effects of discontinuities are minor and this solution was therefore not implemented[5].

Following a definition of FE by Zienkiewicz[8] an element stiffness matrix can be calculated with,

$$\boldsymbol{K} = \int_{-1}^{1}\int_{-1}^{1} \boldsymbol{B}^T\boldsymbol{D}\boldsymbol{B}\ \det \boldsymbol{J}\ \ \mathrm{d}\xi\mathrm{d}\eta$$

where $\boldsymbol{D}$ the material stiffness matrix is, $\boldsymbol{J}$ the Jacobian of axis system transformation $(\xi,\ \eta)$ into the global $(x,\ y)$ axis system is and $\boldsymbol{B}$ the matrix is that converts displacement into strain. The integration was performed with a Gauss-Legendre quadrature function with 8x8 integration points as was found sufficient by L.N. Gifford[5].

For a standard bicubic serendipity element this $\boldsymbol{B}$ matrix is of shape $(3,\ 24)$ however due to the enrichment it becomes $(3,\ 26)$. Which results in a final stiffness matrix of $(26,\ 26)$. Where

$$\boldsymbol{f} = \boldsymbol{K}\boldsymbol{u} = \begin{pmatrix} f_x^0 \\ \vdots \\ f_x^* \\ f_y^* \end{pmatrix} = \begin{bmatrix} \boldsymbol{k} & \vdots & \boldsymbol{k}_{12} \\ \dots & \vdots & \dots \\ \boldsymbol{k}_{21} & \vdots & \boldsymbol{k}_{22} \end{bmatrix} \begin{pmatrix} u_x^0 \\ \vdots \\ K_I \\ K_{II} \end{pmatrix}$$

Here $\boldsymbol{k}$ is similar to the stiffness matrix of a normal bicubic element, the enrichment is in the parts $\boldsymbol{k}_{12}$, $\boldsymbol{k}_{21}$ and $\boldsymbol{k}_{22}$. New terms do also appear in the force vector, where f_x^* and f_y^* are so-called singular loads. They describe the external forces applied on the crack boundary[4], in general these values are zero.

Meshing strategy

To reduce computational costs these enriched elements are only used at the crack tip and conventional linear elements are used throughout the rest of the mesh. It uses the hanging node method to connect the elements as can be seen in Fig. 3.4.4.

Fig. 3.4.4: Top section of mesh around a crack tip, $\oplus$ is the enrichment node with K_I and K_{II}, while solid circles represent the linear ones and the open circle the higher order ones.

This mesh is not conform which can potentially cause the displacement field to become discontinuous. To avoid this one could use normal bicubic serendipity elements throughout the entire mesh which is computational inefficient. However, using a multi-resolution interpretation of topology optimization its performance might be improved[9].

Currently the linear system of the FEA, $\boldsymbol{f} = \boldsymbol{K}\boldsymbol{u}$, and the adjoint equation, $\boldsymbol{l} = \boldsymbol{K}\boldsymbol{\lambda}$, are solved with a complete Cholesky decomposition. A more efficient methods can be formulated with a Multi Grid Conjugate Gradient method as proposed by O. Amir[10].

8 O. C. Zienkiewicz, The Finite Element Method In Engineering Science. New York (NY): McGraw-Hill, 1971.

9 J. P. Groen, M. Langelaar, O. Sigmund, and M. Ruess, "Higher-order multi-resolution topology optimization using the finite cell method," Int. J. Numer. Methods Eng., vol. 110, no. 10, pp. 903–920, Jun. 2017.

10 O. Amir, N. Aage, and B. S. Lazarov, "On multigrid-CG for efficient topology optimization," Struct. Multidiscip. Optim., vol. 49, no. 5, pp. 815–829, May 2014.

Objective formulation

As a spacial discretized method (FEA) was used to calculate the objective the problem formulation needs to become discretized as well. For a mesh of N elements the optimization objective becomes;

$$\begin{aligned}
\max_{X_1,X_2,\ldots,X_N} \quad & N = \frac{1}{C}\sum_{l=1}^{L-1}\frac{(a_{l+1}+a_l)}{\left(\frac{1}{2}\left(K_I(a_{l+1})+K_I(a_l)\right)\right)^m} \\
\text{s.t. :} \quad & \boldsymbol{K}\boldsymbol{u} = \boldsymbol{f} \\
& \sum_{e=1}^{N} v_e X_e \ \leq \ V \\
& X_{\min} \leq X_e \leq X_{\max} \ \ \forall \ \ e \in \{1,2,\ldots,N\} \\
\text{where :} \quad & \boldsymbol{K} = \sum_{e=1}^{N} \boldsymbol{K}_e(X_e,\overline{E}) \\
& K_I(a_l) = K_I^l = \left(\boldsymbol{l}^l\right)^T \boldsymbol{u}^l
\end{aligned}$$

which maximized the FCGL while ensuring equilibrium and setting constraints to the density distribution. Here $\boldsymbol{u}$ is the enriched displacement vector, f the force vector and v_e is the (relative) element volume. $\boldsymbol{l}$ is zero vector except for the degree of freedom linked to the stress intensity factor, and the multiplication of $\boldsymbol{l}^T\boldsymbol{u}$ will return the stress intensity factor at one crack length. This is similar to the compliant mechanism optimization mentioned by O. Sigmund[11] where the displacement of a specific degree of freedom is maximized.

This formulation of the objective can not be combined with Method of Moving Asymptotes because MMA requires the derivatives of the objective function and constraints to have the same order of magnitude. Hence the obective function is scaled linearly to be in the same order as the density constraint, this resulted in the discrete objective:

$$\begin{aligned}
\max_{X_1,X_2,\ldots,X_N} \quad & O = \frac{1}{m2^m\sum_{l=1}^{L-1}(a_{l+1}+a_l)}\sum_{l=1}^{L-1}\frac{(a_{l+1}+a_l)}{\left(\frac{1}{2}\left(K_I(a_{l+1})+K_I(a_l)\right)\right)^m} \\
\text{s.t. :} \quad & \boldsymbol{K}\boldsymbol{u} = \boldsymbol{f} \\
& \sum_{e=1}^{N} v_e X_e \ \leq \ V \\
& X_{\min} \leq X_e \leq X_{\max} \ \ \forall \ \ e \in \{1,2,\ldots,N\} \\
\text{where :} \quad & \boldsymbol{K} = \sum_{e=1}^{N} \boldsymbol{K}_e(X_e,\overline{E}) \\
& K_I(a_l) = K_I^l = \left(\boldsymbol{l}^l\right)^T \boldsymbol{u}^l
\end{aligned}$$

3.4.3 Sensitivity Analysis

The local convex approximation requires the calculation of the sensitivity of O to a density change in any element. Because K_I is the only thing dependend on the design variables the objective gradient is formulated as a function of $\partial K_I/\partial X_e$.

$$\frac{\partial O}{\partial X_e} = -\frac{1}{\sum_{l=1}^{L-1}(a_{l+1}+a_l)}\sum\nolimits_{l=1}^{L-1}\frac{(a_{l+1}+a_l)\left(\frac{\partial K_I(a_{l+1})}{\partial X_e}+\frac{\partial K_I(a_l)}{\partial X_e}\right)}{\left(K_I(a_{l+1})+K_I(a_l)\right)^{m+1}}$$

[11] J. Lu, N. Kashaev, and N. Huber, "Crenellation Patterns for Fatigue Crack Retardation in Fuselage Panels Optimized via Genetic Algorithm," Procedia Eng., vol. 114, pp. 248–254, 2016.

$\partial K_I(a_l)/\partial X_e$ has to be calculated for all crack lengths. The derivative derivation for a specific crack length starts with adding a zero term after the known function $K_I = \boldsymbol{l}^T \boldsymbol{u}$, where $\boldsymbol{\lambda}$ is an arbitrary vector:

$$K_I = \boldsymbol{l}^T \boldsymbol{u} - \boldsymbol{\lambda}^T (\boldsymbol{K}\boldsymbol{u} - \boldsymbol{f})$$

$$\frac{\partial K_I}{\partial X_e} = \left(\boldsymbol{l}^T - \boldsymbol{\lambda}^T \boldsymbol{K}\right) \frac{\partial \boldsymbol{u}}{\partial X_e} - \boldsymbol{\lambda}^T \frac{\partial \boldsymbol{K}}{\partial X_e} \boldsymbol{u}$$

Now choosing a convenient vector for $\boldsymbol{\lambda}$ which causes $\boldsymbol{l}^T - \boldsymbol{\lambda}^T \boldsymbol{K}$ to be zero leads to the following expression for the sensitivity,

$$\frac{\partial K_I}{\partial X_e} = -\boldsymbol{\lambda}^T \frac{\partial \boldsymbol{K}}{\partial X_e} \boldsymbol{u}$$

$$\text{where:} \qquad \boldsymbol{l} = \boldsymbol{K}\boldsymbol{\lambda}$$

This means that $\boldsymbol{\lambda}$ can be calculated with the FEA, where $\boldsymbol{l}$ is seen as a sort force vector, by solving $\boldsymbol{l} = \boldsymbol{K}\boldsymbol{u}$. The sensitivity of $\boldsymbol{K}$ to the element density can be calculated, resulting in the following gradient:

$$\frac{\partial K_I}{\partial X_e} = -pX_e^{p-1} \boldsymbol{\lambda}^T \boldsymbol{K}_e \boldsymbol{u}$$

3.4.4 Computational implementation

The iterative implementation of topology optimization as proposed by M. Beckers,[8] or M.P. Bendsøe and O. Sigmund[2] are similar. It exists out of three parts, initialization, optimization and post processing. The flowchart of the local compliance algorithm can be found in Fig. 3.4.5.

Fig. 3.4.5: Flowchart for maximum fatigue crack growth life[7].

In the initialization phase the problem is set up. It defines the design domain, the loading conditions, the initial design and generates the finite element mesh that will be used in the optimization phase.

The optimization phase is the iterative method that solves the topology problem. It will analyze the current design with multiple FEA, for each crack length increment one. After which it will calculate the sensitivity of the stress intensity factor to the density of each element, for each crack length increments. Then the over all performance and sensitivity is calculate, this is used in the local approximation and update scheme which is discussed in *Sensitivity analysis and MMA*. The Method of Moving Asymptotes (MMA), developed by K. Svanberg[9], is used to formulate a simplified convex approximation of the problem which is optimized to formulate the updated design. These steps are performed in a loop until the design is converged, i.e. when the change in design between two iterations becomes negligible.

Post processing is required to remove the last elements with intermediate values and generate a shape out of the design, for example a CAD or STL file. This algorithm will not contain any of the post processing steps. The code used in this communication simply plots the final shape and load case.

Limitations

The limitations of the fatigue crack growth life maximization are inherited from the stress intensity minimization one. Two of these limitations are discussed again, as they have more impact on this FCGL maximization than they had on the SIF minimization.

That the thickness of crack tip elements cannot be changed is a significant problem for fatigue life maximization of variable thickness plates. The fatigue crack growth analysis requires the crack to propagate. In the fatigue maximization all elements around the crack are forced to have unit thickness. Literature shows that creating patterns of varying

thickness/stiffness in front and after the crack tip influences the crack growth rate and the overall fatigue live[11],[12]. These kinds of crenelation patterns cannot be created by the optimization algorithm.

That the crack geometry needs to be determined in advance does also have a larger impact in this crack growth life maximization algorithm. The fatigue life optimization assumes a crack path and does not consider that the crack might deviate from it. It might very well be possible that a better design, one in which more load cycles are required for the crack to grow a certain length, can be obtained by enquote{crack steering}. It is recommended to investigate how the method can be expanded such that crack steering becomes possible.

Computational efficiency

In this thesis little attention was payed to the computational efficiency, stress intensity minimization was fast enough to run on a simple laptop anyway. This is different for fatigue life maximization. The difference in computational requirements comes from the fact that information of the stress intensity and its sensitivity are required as a function of crack length. The fatigue growth model requires calculating stress intensity factors for the crack at different values of a. For each stress intensity calculation a mesh needs to be generated on which a FEA and adjoint problem will be solved.

In the current, simple but inefficient, implementation the following steps are taken:

- During the problem initialization the meshes for the crack at all lengths are generated.
- During each iteration the following steps are performed for all these meshes:
  - Assemble the stiffness matrix.
  - Solve both the linear elastic and adjoint problems with a complete Cholesky factorization, which has a computational complexity of $O(n^3/3)$.

All meshes are generated ones and reused throughout all iterations, which compared to regenerating them, reduces the computational requirements. This causes an increase of the memory requirements, because all the meshes generated need to be saved untill they are used. The size of all these arrays becomes significant. Take for example a problem with a mesh of 500 by 240 elements, each mesh required 0.3 GB memory to store. For fatigue life maximization many of these meshes need to be saved. For an optimization with a crack that growths from element 220 to 430 around 210 crack length increments are required, just saving the meshes requires 63 GB of RAM already.

No attempt to improve the mesh generation and saving was made because the current implementation is incompatible with any method that allows for crack steering. When the crack path can be changed by the optimization variables, the mesh of the current crack increment can only be determined after finishing the FEA calculation of the previous increment. This means that the mesh can only be generated in each increment.

Besides the memory requirement, the optimization requires a large computational effort as it needs to solve two systems of linear equations per crack length considered. For a mesh of 500 by 240 elements every iteration required around 13 minutes on a pc with a Intel Xeon E5-1620 v2. The optimization required 12 days to converge, this is significantly longer than the 4 to 8 hours which is used in stress intensity minimization at the same resolution. To reduce both the memory and computational requirements one could use a crack increment that are larger than one element between every stress intensity calculation. Performing the calculation every two elements will already half the memory and computational requirements.

Taking crack length increments that are far greater than the element size will result in inaccurate fatigue life predictions which has a large effect on the optimization results. An optimization with large increments will design a structure that preforms well at the location where the stress intensity factors are calculated and neglect the rest. Cref{fig:increments_geometry} the result of an optimization with a crack increment of 25 elements is shown. A more accurate FEA with used crack increments of 1 element was run. The area under the dN/da curves in

[12] C. D. Rans, R. Rodi, and R. Alderliesten, "Analytical prediction of mode I stress intensity factors for cracked panels containing bonded stiffeners," Eng. Fract. Mech., vol. 97, no. 1, pp. 12–29, 2012.

cref{fig:inaccuracy dN/da} of the smaller crack increments is lower. This proves that taking to large increments will lead to degenerate designs of with performance is overestimated by the optimization. From experience a crack increment of two elements can always be used without any artifacts appearing. This is also why the lines shown in cref{fig:StressIntensity_FL,fig:crack_growth,fig:Cycles} are generated by calculating the stress intensity values every two elements.

Improving the computational efficiency should be a major focus before expanding the capabilities to higher resolution or 3D problems. One could consider improving the currently algorithm by using efficient FE problem solvers[13] and creating a parallel implementation[14] with for example the PETSc frameworkfootnote{Look for an example at href{http://www.topopt.mek.dtu.dk/Apps-and-software/Large-scale-topology-optimization-code-using-PETSc}{TopOpt_in_PETSc} or[15]}. Another solution to reduce the computational requirement is to reduce the amount of FEA that need to be performed, for example by replacing them with more simple algebraic approximations. B. Herremans showed that an algebraic approximation of the fatigue performance could replace the FE model used in optimization algorithm, wile retaining accuracy. The original model (developed by J. Lu[11]) was to slow for high resolution problem, while the improved version could be run in a matter of seconds[16].

3.4.5 Exaples and results

3.4.6 References

13

O. Amir, N. Aage, and B. S. Lazarov, "On multigrid-CG for efficient topology optimization," Struct. Multidiscip. Optim., vol. 49, no. 5, pp. 815–829, May 2014.

14

N. Aage and B. S. Lazarov, "Parallel framework for topology optimization using the method of moving asymptotes," Struct. Multidiscip. Optim., vol. 47, no. 4, pp. 493–505, Apr. 2013.

15

N. Aage, E. Andreassen, and B. S. Lazarov, "Topology optimization using PETSc: An easy-to-use, fully parallel, open source topology optimization framework," Struct. Multidiscip. Optim., vol. 51, no. 3, pp. 565–572, Mar. 2015.

16

B. Herremans, "Thickness distribution optimisation in flat panels for damage tolerance using genetic algorithms," Technical University of Delft, 2019.

CHAPTER 4

Docstrings

4.1 Global Compliance Minimization

The total compliance minimization does design structures with maximum stiffness as is discussed at *Global Compliance Minimization*. An example as how to use the optimization is shown in an example optimization example.py

- *Density Constraints*
- *Load Cases*
- *Finite Element Solvers*
- *Optimization Module*
- *Plotting Module*

4.1.1 Density Constraints

Constraints class used to specify the density constraints of the topology optimisation problem. It contains functions for minimum and maximum element density in the upcomming iteration and the magnitude of the volume constraint function itself of the current design. This version of the code is used for the global compliance minimization.

Bram Lagerweij Aerospace Structures and Materials Department TU Delft 2018

class src_Compliance.constraints.**DensityConstraint**(*nelx*, *nely*, *move*, *volume_frac*, *density_min=0.0*, *density_max=1.0*)

This object relates to the constraints used in this optimization. It can be used for the MMA updatescheme to derive what the limit is for all element densities at every itteration. The class itself is not changed by the itterations.

Parameters

- **nelx** (*int*) – Number of elements in x direction.

- **nely** (*int*) – Number of elements in y direction.
- **move** (*float*) – Maximum change in density of an element over 1 itteration.
- **volume_frac** (*float*) – Maximum volume that can be filled with material.
- **volume_derivative** (*2D array size(1, nelx*nely)*) – Sensityvity of the density constraint to the density in each element.
- **density_min** (*float, optional*) – Minumum density, set at 0.0 if not specified.
- **density_max** (*float, optional*) – Maximum density, set at 0.0 if not specified.

nelx
Number of elements in x direction.

Type int

nely
Number of elements in y direction.

Type int

move
Maximum change in density of an element over 1 itteration.

Type float

volume_frac
Maximum volume that can be filled with material.

Type float

volume_derivative
Sensityvity of the density constraint to the density in each element.

Type 2D array size(1, nelx*nely)

density_min
Minumum density, set at 0.0 if not specified.

Type float, optional

density_max
Maximum density, set at 0.0 if not specified.

current_volconstrain(*x*)
Calculates the current magnitude of the volume constraint funcion:

$$V_{\text{constraint}} = \frac{\sum v_e X_e}{V_{\max}} - 1$$

Parameters **x** (*2D array size(nely, nelx)*) – Density distribution of this itteration.

Returns **curvol** – Curent value of the density constraint function.

Return type float

xmax(*x*)
This function calculates the maximum density value of all ellements of this itteration.

Parameters **x** (*2D array size(nely, nelx)*) – Density distribution of this itteration.

Returns **xmax** – Maximum density values of this itteration after updating.

Return type 2D array size(nely, nelx)

xmin(*x*)

This function calculates the minimum density value of all ellements of this itteration.

Parameters **x** (*2D array size(nely, nelx)*) – Density distribution of this itteration.

Returns **xmin** – Minimum density values of this itteration for the update scheme.

Return type 2D array size(nely, nelx)

4.1.2 Load Cases

This file containts the Load class that allows the generation of an object that contains geometric, mesh, loads and boundary conditions that belong to the load case. This version of the code is meant for global compliance minimization.

Bram Lagerweij Aerospace Structures and Materials Department TU Delft 2018

Parent Load Case

class src_Compliance.loads.**Load**(*nelx*, *nely*, *young*, *Emin*, *poisson*)

Load parent class that contains the basic functions used in all load cases. This class and its children do cantain information about the load case conciderd in the optimisation. The load case consists of the mesh, the loads, and the boundaries conditions. The class is constructed such that new load cases can be generated simply by adding a child and changing the function related to the geometry, loads and boundaries.

Parameters

- **nelx** (*int*) – Number of elements in x direction.
- **nely** (*int*) – Number of elements in y direction.
- **young** (*float*) – Youngs modulus of the materias.
- **Emin** (*float*) – Artifical Youngs modulus of the material to ensure a stable FEA. It is used in the SIMP based material model.
- **poisson** (*float*) – Poisson ration of the material.

nelx

Number of elements in x direction.

Type int

nely

Number of elements in y direction.

Type int

young

Youngs modulus of the materias.

Type float

Emin

Artifical Youngs modulus of the material to ensure a stable FEA. It is used in the SIMP based material model.

Type float

poisson

Poisson ration of the material.

Type float

dim
Amount of dimensions conciderd in the problem, set at 2.

Type int

alldofs()
Returns a list with all degrees of freedom.

Returns **all** – List with numbers from 0 to the maximum degree of freedom number.

Return type 1-D list

edof()
Generates an array with the position of the nodes of each element in the global stiffness matrix.

Returns

- **edof** (*2-D array size(nelx*nely, 8)*) – The list with all elements and their degree of freedom numbers.
- **x_list** (*1-D array len(nelx*nely*8*8)*) – The list with the x indices of all ellements to be inserted into the global stiffniss matrix.
- **y_list** (*1-D array len(nelx*nely*8*8)*) – The list with the y indices of all ellements to be inserted into the global stiffniss matrix.

fixdofs()
Returns a list with indices that are fixed by the boundary conditions.

Returns **fix** – List with all the numbers of fixed degrees of freedom. This list is empty in this parrent class.

Return type 1-D list

force()
Returns an 1D array, the force vector of the loading condition.

Returns **f** – Empy force vector.

Return type 1-D array length covering all degrees of freedom

freedofs()
Returns a list of arr indices that are not fixed

Returns **free** – List containing all elemens of alldogs except those that appear in the freedofs list.

Return type 1-D list

lk(*E*, *nu*)
Calculates the local siffness matrix depending on E and nu.

Parameters

- **E** (*float*) – Youngs modulus of the material.
- **nu** (*float*) – Poisson ratio of the material.

Returns **ke** – Local stiffness matrix.

Return type 2-D array size(8, 8)

node(*elx*, *ely*)
Calculates the topleft node number of the requested element

Parameters

- **elx** (*int*) – X position of the conciderd element.
- **ely** (*int*) – Y position of the conciderd element.

Returns **topleft** – The node number of the top left node

Return type int

nodes(*elx*, *ely*)
Calculates all node numbers of the requested element

Parameters

- **elx** (*int*) – X position of the conciderd element.
- **ely** (*int*) – Y position of the conciderd element.

Returns

- **n1** (*int*) – The node number of the top left node.
- **n2** (*int*) – The node number of the top right node.
- **n3** (*int*) – The node number of the bottom right node.
- **n4** (*int*) – The node number of the bottom left node.

passive()
Retuns three lists containing the location and magnitude of fixed density values

Returns

- **elx** (*1-D list*) – X coordinates of all passive elements, empty for the parrent class.
- **ely** (*1-D list*) – Y ccordinates of all passive elements, empty for the parrent class.
- **values** (*1-D list*) – Density values of all passive elements, empty for the parrent class.
- **fix_ele** (*1-D list*) – List with all element numbers that are allowed to change.

Child Load Cases

class src_Compliance.loads.**HalfBeam**(*nelx*, *nely*, *young*, *Emin*, *poisson*)
Bases: *src_Compliance.loads.Load*

This child of the Loads class represents the loading conditions of a half mbb-beam. Only half of the beam is considerd due to the symetry around the y axis.

No methods are added compared to the parrent class. The force and fixdofs functions are changed to output the correct force vector and boundary condition used in this specific load case. See the functions themselfs for more details

fixdofs()
The boundary conditions of the half mbb-beam fix the x displacments of all the nodes at the outer left side and the y displacement of the bottom right element.

Returns **fix** – List with all the numbers of fixed degrees of freedom.

Return type 1-D list

force()
The force vector containts a load in negative y direction at the top left corner.

Returns **f** – A -1 is placed at the index of the y direction of the top left node.

Return type 1-D array length covering all degrees of freedom

class src_Compliance.loads.**Beam**(*nelx*, *nely*, *young*, *Emin*, *poisson*)
Bases: *src_Compliance.loads.Load*

This child of the Loads class represents the full mbb-beam without assuming an axis of symetry. To enforce an node in the middle nelx needs to be an even number.

No methods are added compared to the parrent class. The force and fixdofs functions are changed to output the correct force vector and boundary condition used in this specific load case. See the functions themselfs for more details

fixdofs()
The boundary conditions of the full mbb-beam fix the x and y displacment of the bottom left node ande the y displacement of the bottom right node.

Returns **fix** – List with all the numbers of fixed degrees of freedom.

Return type 1-D list

force()
The force vector containts a load in negative y direction at the mid top node.

Returns **f** – Where at the inndex relating to the y direction of the top mid node a -1 is placed.

Return type 1-D array length covering all degrees of freedom

class src_Compliance.loads.**Canti**(*nelx*, *nely*, *young*, *Emin*, *poisson*)
Bases: *src_Compliance.loads.Load*

This child of the Loads class represents the loading conditions of a cantilever beam. The beam is encasted on the left an the load is applied at the middel of the right side. To do this an even number for nely is required.

No methods are added compared to the parrent class. The force and fixdofs functions are changed to output the correct force vector and boundary condition used in this specific load case. See the functions themselfs for more details

fixdofs()
The boundary conditions of the cantileverbeam fix the x and y displacment of all nodes on the left side.

Returns **fix** – List with all the numbers of fixed degrees of freedom.

Return type 1-D list

force()
The force vector containts a load in negative y direction at the mid most rigth node.

Returns **f** – Where at the inndex relating to the y direction of the mid right node a -1 is placed.

Return type 1-D array length covering all degrees of freedom

class src_Compliance.loads.**Michell**(*nelx*, *nely*, *young*, *Emin*, *poisson*)
Bases: *src_Compliance.loads.Load*

This child of the Loads class represents the loading conditions of a half a Michell structure. A load is applied in the mid left of the design space and the boundary conditions fixes the x and y direction of the middle right node. Due to symetry all nodes at the left side are constraint in x direction. This class requires nely to be even.

No methods are added compared to the parrent class. The force and fixdofs functions are changed to output the correct force vector and boundary condition used in this specific load case. See the functions themselfs for more details

fixdofs()
The boundary conditions of the half mbb-beam fix the x displacments of all the nodes at the outer left side and the y displacement of the mid right element.

Returns **fix** – List with all the numbers of fixed degrees of freedom.

Return type 1-D list

force()
The force vector containts a load in negative y direction at the mid most left node.

Returns **f** – Where at the inndex relating to the y direction of the mid left node a -1 is placed.

Return type 1-D array length covering all degrees of freedom

class src_Compliance.loads.**BiAxial**(*nelx*, *nely*, *young*, *Emin*, *poisson*)
Bases: *src_Compliance.loads.Load*

This child of the Loads class represents the loading conditions of a bi-axial loaded plate. All outer nodes have a load applied that goes outward. This class is made to show the checkerboard problem that generaly occeurs with topology optimisation.

No methods are added compared to the parrent class. The force, fixdofs and passive functions are changed to output the correct force vector, boundary condition and passive elements used in this specific load case. See the functions themselfs for more details

fixdofs()
The boundary conditions fix the top left node in x direction, the bottom left node in x and y direction and the bottom right node in y direction only.

Returns **fix** – List with all the numbers of fixed degrees of freedom.

Return type 1-D list

force()
The force vector containing loads that act outward from the edge.

Returns **f** – Where at the indices related to the outside nodes an outward force of 1 is inserted.

Return type 1-D array length covering all degrees of freedom

passive()
The Bi-Axial load case requires fully dense elements around the border. This is done to enforce propper load introduction.

Returns

- **elx** (*1-D list*) – X coordinates of all passive elements, empty for the parrent class.
- **ely** (*1-D list*) – Y ccordinates of all passive elements, empty for the parrent class.
- **values** (*1-D list*) – Density values of all passive elements, empty for the parrent class.

4.1.3 Finite Element Solvers

Finite element solvers for the displacement from stiffness matrix and force vector. This version of the code is meant for global compliance minimization.

Bram Lagerweij Aerospace Structures and Materials Department TU Delft 2018

Parent Solver

class src_Compliance.fesolvers.**FESolver**(*verbose=False*)
This parent FEA class can only assemble the global stiffness matrix and exclude all fixed degrees of freedom from it. This stiffenss csc-sparse stiffness matrix is assebled in the gk_freedof method. This class solves the FE

problem with a sparse LU-solver based upon umfpack. This solver is slow and inefficient. It is however more robust.

Parameters **verbose** (*bool, optional*) – False if the FEA should not print updates

verbose
False if the FEA does not print updates.

Type bool

displace(*load*, *x*, *ke*, *kmin*, *penal*)
FE solver based upon the sparse SciPy solver that uses umfpack.

Parameters

- **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.
- **x** (*2-D array size(nely, nelx)*) – Current density distribution.
- **ke** (*2-D array size(8, 8)*) – Local fully dense stiffnes matrix.
- **kmin** (*2-D array size(8, 8)*) – Local stiffness matrix for an empty element.
- **penal** (*float*) – Material model penalisation (SIMP).

Returns **u** – Displacement of all degrees of freedom

Return type 1-D array len(max(edof)+1)

gk_freedofs(*load*, *x*, *ke*, *kmin*, *penal*)
Generates the global stiffness matrix with deleted fixed degrees of freedom. It generates a list with stiffness values and their x and y indices in the global stiffness matrix. Some combination of x and y appear multiple times as the degree of freedom might apear in multiple elements of the FEA. The SciPy coo_matrix function adds them up at the background.

Parameters

- **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.
- **x** (*2-D array size(nely, nelx)*) – Current density distribution.
- **ke** (*2-D array size(8, 8)*) – Local fully dense stiffnes matrix.
- **kmin** (*2-D array size(8, 8)*) – Local stiffness matrix for an empty element.
- **penal** (*float*) – Material model penalisation (SIMP).

Returns **k** – Global stiffness matrix without fixed degrees of freedom.

Return type 2-D sparse csc matrix

Child Solvers

class src_Compliance.fesolvers.**CvxFEA**(*verbose=False*)
Bases: *src_Compliance.fesolvers.FESolver*

This parent FEA class is used to assemble the global stiffness matrix while this class solves the FE problem with a Supernodal Sparse Cholesky Factorization.

verbose
False if the FEA should not print updates.

Type bool

displace(*load*, *x*, *ke*, *kmin*, *penal*)

FE solver based upon a Supernodal Sparse Cholesky Factorization. It requires the instalation of the cvx module.[1]

Parameters

- **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.
- **x** (*2-D array size(nely, nelx)*) – Current density distribution.
- **ke** (*2-D array size(8, 8)*) – Local fully dense stiffnes matrix.
- **kmin** (*2-D array size(8, 8)*) – Local stiffness matrix for an empty element.
- **penal** (*float*) – Material model penalisation (SIMP).

Returns **u** – Displacement of all degrees of freedom

Return type 1-D array len(max(edof))

References

class src_Compliance.fesolvers.**CGFEA**(*verbose=False*)

Bases: *src_Compliance.fesolvers.FESolver*

This parent FEA class can assemble the global stiffness matrix and this class solves the FE problem with a sparse solver based upon a preconditioned conjugate gradient solver. The preconditioning is based upon the inverse of the diagonal of the stiffness matrix.

verbose

False if the FEA should not print updates.

Type bool

ufree_old

Displacement field of previous CG iteration

Type array len(freedofs)

displace(*load*, *x*, *ke*, *kmin*, *penal*)

FE solver based upon the sparse SciPy solver that uses a preconditioned conjugate gradient solver, preconditioning is based upon the inverse of the diagonal of the stiffness matrix. Currently the relative tolerance is hardcoded as 1e-3.

Recomendations

- Make the tolerance change over the iterations, low accuracy is required for first itteration, more accuracy for the later ones.
- Add more advanced preconditioner.
- Add gpu accerelation.

Parameters

- **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.
- **x** (*2-D array size(nely, nelx)*) – Current density distribution.

[1] Y. Chen, T. A. Davis, W. W. Hager, S. Rajamanickam, "Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate", ACM Transactions on Mathematical Software, 35(3), 22:1-22:14, 2008.

- **ke** (*2-D array size(8, 8)*) – Local fully dense stiffnes matrix.
- **kmin** (*2-D array size(8, 8)*) – Local stiffness matrix for an empty element.
- **penal** (*float*) – Material model penalisation (SIMP).

Returns **u** – Displacement of all degrees of freedom

Return type 1-D array len(max(edof)+1)

4.1.4 Optimization Module

Topology Optimization class that handles the itterations, objective functions, filters and update scheme. It requires to call upon a constraint, load case and FE solver classes. This version of the code is meant for global compliance minimization.

Bram Lagerweij Aerospace Structures and Materials Department TU Delft 2018

class src_Compliance.topopt.**Topopt**(*constraint*, *load*, *fesolver*, *verbose=False*)

This is the optimisation object itself. It contains the initialisation of the density distribution.

Parameters

- **constraint** (*object of DensityConstraint class*) – The constraints for this optimization problem.
- **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.
- **fesolver** (*object, child of the CSCStiffnessMatrix class*) – The finite element solver.
- **verbose** (*bool, optional*) – Printing itteration results.

constraint

The constraints for this optimization problem.

Type object of DensityConstraint class

load

The loadcase(s) considerd for this optimisation problem.

Type object, child of the Loads class

fesolver

The finite element solver.

Type object, child of the CSCStiffnessMatrix class

verbose

Printing itteration results.

Type bool, optional

itr

Number of iterations performed

Type int

x

Array containing the current densities of every element.

Type 2-D array size(nely, nelx)

xold1
Flattend density distribution one iteration ago.

Type 1D array len(nelx*nely)

xold2
Flattend density distribution two iteration ago.

Type 1D array len(nelx*nely)

low
Column vector with the lower asymptotes, calculated and used in the MMA subproblem of the previous itteration.

Type 1D array len(nelx*nely)

upp
Column vector with the lower asymptotes, calculated and used in the MMA subproblem of the previous itteration.

Type 1D array len(nelx*nely)

comp(*x*, *u*, *ke*, *penal*)
This funcion calculates compliance and compliance density derivative.

Parameters

- **x** (*2-D array size(nely, nelx)*) – Possibly filterd density distribution.
- **u** (*1-D array len(max(edof)+1)*) – Displacement of all degrees of freedom.
- **ke** (*2-D array size(8, 8)*) – Element stiffness matrix with full density.
- **penal** (*float*) – Material model penalisation (SIMP).

Returns

- **c** (*float*) – Compliance for the current design.
- **dc** (*2-D array size(nely, nelx)*) – Compliance sensitivity to density changes.

densityfilt(*rmin*, *filt*)
Filters with a normalized convolution on the densities with a radius of rmin if:

```
>>> filt=='density'
```

Parameters

- **rmin** (*float*) – Filter size.
- **filt** (*str*) – The filter type that is selected, either 'sensitivity' or 'density'.

Returns **xf** – Filterd density distribution.

Return type 2-D array size(nely, nelx)

iter(*penal*, *rmin*, *filt*)
This funcion performs one itteration of the topology optimisation problem. It

- loads the constraints,
- calculates the stiffness matrices,
- executes the density filter,
- executes the FEA solver,

- calls upon the compliance and compliance sensitivity calculation,
- executes the sensitivity filter,
- executes the MMA update scheme,
- and finaly updates density distribution (design).

Parameters

- **penal** (*float*) – Material model penalisation (SIMP).
- **rmin** (*float*) – Filter size.
- **filt** (*str*) – The filter type that is selected, either 'sensitivity' or 'density'.

Returns

- **change** (*float*) – Largest difference between the new and old density distribution.
- **c** (*float*) – Compliance for the current design.

layout (*penal*, *rmin*, *delta*, *loopy*, *filt*, *history=False*)

Solves the topology optimisation problem by looping over the iter function.

Parameters

- **penal** (*float*) – Material model penalisation (SIMP).
- **rmin** (*float*) – Filter size.
- **delta** (*float*) – Convergence is roached when delta > change.
- **loopy** (*int*) – Amount of iteration allowed.
- **filt** (*str*) – The filter type that is selected, either 'sensitivity' or 'density'.
- **history** (*bool, optional*) – Do the intermediate results need to be stored.

Returns

- **xf** (*array size(nely, nelx)*) – Density distribution resulting from the optimisation.
- **xf_history** (*list of arrays len(itterations size(nely, nelx), float16)*) – List with the density distributions of all itterations, None when history != True.

mma (*m*, *n*, *itr*, *xval*, *xmin*, *xmax*, *xold1*, *xold2*, *f0val*, *df0dx*, *fval*, *dfdx*, *low*, *upp*, *a0*, *a*, *c*, *d*)

This function mmasub performs one MMA-iteration, aimed at solving the nonlinear programming problem:

$$\min f_0(x)$$

$$+a_0 z + \sum_{i=1}^{m} \left(c_i y_i + \frac{1}{2} d_i y_i^2 \right)$$

$$\text{s.t.}$$

$$f_i(x) - a_i z - y_i \leq 0 \qquad i \in \{1, 2, \ldots, m\}$$

$$x_{\min} \geq x_j \geq x_{\max} j \in \{1, 2, \ldots, n\}$$

$$y_i \leq 0 i \in \{1, 2, \ldots, m\}$$

$$z \geq 0$$

Parameters

- **m** (*int*) – The number of general constraints.
- **n** (*int*) – The number of variables x_j.
- **itr** (*int*) – Current iteration number (=1 the first time mmasub is called).
- **xval** (*1-D array len(n)*) – Vector with the current values of the variables x_j.
- **xmin** (*1-D array len(n)*) – Vector with the lower bounds for the variables x_j.
- **xmax** (*1-D array len(n)*) – Vector with the upper bounds for the variables x_j.
- **xold1** (*1-D array len (n)*) – xval, one iteration ago when iter>1, zero othewise.
- **xold2** (*1-D array len (n)*) – xval, two iteration ago when iter>2, zero othewise.
- **f0val** (*float*) – The value of the objective function f_0 at xval.
- **df0dx** (*1-D array len(n)*) – Vector with the derivatives of the objective function f_0 with respect to the variables x_j, calculated at xval.
- **fval** (*1-D array len(m)*) – Vector with the values of the constraint functions f_i, calculated at xval.
- **dfdx** (*2-D array size(m x n)*) – (m x n)-matrix with the derivatives of the constraint functions f_i. with respect to the variables x_j, calculated at xval.
- **low** (*1-D array len(n)*) – Vector with the lower asymptotes from the previous iteration (provided that iter>1).
- **upp** (*1-D array len(n)*) – Vector with the upper asymptotes from the previous iteration (provided that iter>1).
- **a0** (*float*) – The constants a_0 in the term $a_0 z$.
- **a** (*1-D array len(m)*) – Vector with the constants $a_i 1 in the terms : math :$.
- **c** (*1-D array len(m)*) – Vector with the constants c_i in the terms $c_i * y_i$.
- **d** (*1-D array len(m)*) – Vector with the constants d_i in the terms $0.5 d_i (y_i)^2$.

Returns

- **xmma** (*1-D array len(n)*) – Column vector with the optimal values of the variables x_j in the current MMA subproblem.
- **low** (*1-D array len(n)*) – Column vector with the lower asymptotes, calculated and used in the current MMA subproblem.
- **upp** (*1-D array len(n)*) – Column vector with the upper asymptotes, calculated and used in the current MMA subproblem.
- *Version September 2007 (and a small change August 2008)*
- *Krister Svanberg <krille@math.kth.se>*
- *Department of Mathematics KTH, SE-10044 Stockholm, Sweden.*
- *Translated to python 3 by A.J.J. Lagerweij TU Delft June 2018*

sensitivityfilt(*x*, *rmin*, *dc*, *filt*)

Filters with a normalized convolution on the sensitivity with a radius of rmin if:

```
>>> filt=='sensitivity'
```

Parameters

- **x** (*2-D array size(nely, nelx)*) – Current density ditribution.
- **dc** (*2-D array size(nely, nelx*) – Compliance sensitivity to density changes.
- **rmin** (*float*) – Filter size.
- **filt** (*str*) – The filter type that is selected, either 'sensitivity' or 'density'.

Returns **dcf** – Filterd sensitivity distribution.

Return type 2-D array size(nely, nelx)

solvemma(*m*, *n*, *epsimin*, *low*, *upp*, *alfa*, *beta*, *p0*, *q0*, *P*, *Q*, *a0*, *a*, *b*, *c*, *d*)

This function solves the MMA subproblem with a primal-dual Newton method.

$$\min \sum_{j-1}^{n} \left(\frac{p_{0j}^{(k)}}{U_j^{(k)} - x_j} + \frac{q_{0j}^{(k)}}{x_j - L_j^{(k)}} \right) + a_0 z + \sum_{i=1}^{m} \left(c_i y_i + \frac{1}{2} d_i y_i^2 \right)$$

s.t.

$$\sum_{j-1}^{n} \left(\frac{p_{ij}^{(k)}}{U_j^{(k)} - x_j} + \frac{q_{ij}^{(k)}}{x_j - L_j^{(k)}} \right) - a_i z - y_i \leq b_i$$

$$\alpha_j \geq x_j \geq \beta_j$$

$$z \geq 0$$

Returns **x** – Column vector with the optimal values of the variables x_j in the current MMA subproblem.

Return type 1-D array len(n)

4.1.5 Plotting Module

Plotting the simulated TopOpt geometry with boundery conditions and loads.

Bram Lagerweij Aerospace Structures and Materials Department TU Delft 2018

class src_Compliance.plotting.**Plot**(*load*, *title=None*)

This class contains functions that allows the visualisation of the TopOpt algorithem. It can print the density distribution, the boundary conditions and the forces.

Parameters

- **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.
- **title** (*str, optional*) – Title of the plot if required.

nelx

Number of elements in x direction.

Type int

nely
Number of elements in y direction.

Type int

fig
An empty figure of size nelx/10 and nely/10*1.2 inch.

Type matplotlib.pyplot figure

ax
The axis system that belongs to fig.

Type matplotlib.pyplot axis

images
This list contains all density distributions that need to be plotted.

Type 1-D list with imshow objects

add(*x*, *animated=False*)
Adding a plot of the density distribution to the figure.

Parameters

- **x** (*2-D array size(nely, nelx)*) – The density distribution.
- **animated** (*bool, optional*) – An animated figure is genereted when history = True.

boundary(*load*)
Plotting the boundary conditions.

Parameters **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.

loading(*load*)
Plotting the loading conditions.

Parameters **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.

save(*filename*, *fps=10*)
Saving an plot in svg or mp4 format, depending on the length of the images list. The FasterFFMpegWriter is used when videos are generated. These videos are encoded with a hardware accelerated h264 codec with the .mp4 file format. Other codecs and encoders can be set within the function itself.

Parameters

- **filename** (*str*) – Name of the file, excluding the file exstension.
- **fps** (*int, optional*) – Amount of frames per second if the plots are animations.

show()
Showing the plot in a window.

class src_Compliance.plotting.**FasterFFMpegWriter**(***kwargs*)
Bases: matplotlib.animation.FFMpegWriter

FFMpeg-pipe writer bypassing figure.savefig. To improofs speed with respect to the matplotlib.animation.FFMpegWriter

classmethod bin_path()
Return the binary path to the commandline tool used by a specific subclass. This is a class method so that the tool can be looked for before making a particular MovieWriter subclass available.

cleanup()
Clean-up and collect the process used to write the movie file.

finish()
Finish any processing for writing the movie.

frame_size
A tuple `(width, height)` in pixels of a movie frame.

grab_frame(***savefig_kwargs*)
Grab the image information from the figure and save as a movie frame.

Doesn't use savefig to be faster: savefig_kwargs will be ignored.

classmethod isAvailable()
Check to see if a MovieWriter subclass is actually available.

saving(*fig*, *outfile*, *dpi*, **args*, ***kwargs*)
Context manager to facilitate writing the movie file.

`*args, **kw` are any parameters that should be passed to *setup*.

setup(*fig*, *outfile*, *dpi=None*)
Perform setup for writing the movie file.

Parameters

- **fig** (*~matplotlib.figure.Figure*) – The figure object that contains the information for frames
- **outfile** (*str*) – The filename of the resulting movie file
- **dpi** (*int, optional*) – The DPI (or resolution) for the file. This controls the size in pixels of the resulting movie file. Default is fig.dpi.

4.2 Maximum Local Compliance

This loal compliance maximization designs structures with the maximum displacement in one node. This can be used to design MEMS actuators as is discussed at *Maximum Local Compliance*. An example as how to use the optimization is shown in an example optimization example.py

- *Density Constraints*
- *Load Cases*
- *Finite Element Solvers*
- *Optimization Module*
- *Plotting Module*

4.2.1 Density Constraints

Constraints class used to specify the density constraints of the topology optimisation problem. It contains functions for minimum and maximum element density in the upcoming iteration and the magnitude of the volume constraint function itself of the current design. This version of the code is used for the compliant design, local displacement maximisation.

Bram Lagerweij Aerospace Structures and Materials Department TU Delft 2018

class src_Actuator.constraints.**DensityConstraint**(*nelx*, *nely*, *move*, *volume_frac*, *density_min=0.0*, *density_max=1.0*)

This object relates to the constraints used in this optimization. It can be used for the MMA update scheme to derive what the limit is for all element densities at every iteration. The class itself is not changed by the iterations.

Parameters

- **nelx** (*int*) – Number of elements in x direction.
- **nely** (*int*) – Number of elements in y direction.
- **move** (*float*) – Maximum change in density of an element over 1 itteration.
- **volume_frac** (*float*) – Maximum volume that can be filled with material.
- **volume_derivative** (*2D array size(1, nelx*nely)*) – Sensitivity of the density constraint to the density in each element.
- **density_min** (*float (optional)*) – Minimum density, set at 0.0 if not specified.
- **density_max** (*float (optional)*) – Maximum density, set at 0.0 if not specified.

nelx
Number of elements in x direction.

Type int

nely
Number of elements in y direction.

Type int

move
Maximum change in density of an element over 1 iteration.

Type float

volume_frac
Maximum volume that can be filled with material.

Type float

volume_derivative
Sensitivity of the density constraint to the density in each element.

Type 2D array size(1, nelx*nely)

density_min
Minimum density, set at 0.0 if not specified.

Type float, optional

density_max
Maximum density, set at 0.0 if not specified.

Type float, optional

current_volconstrain(*x*)
Calculates the current magnitude of the volume constraint function:

$$V_{\text{constraint}} = \frac{\sum v_e X_e}{V_{\max}} - 1$$

Parameters **x** (*2D array size(nely, nelx)*) – Density distribution of this iteration.

Returns **curvol** – Current value of the density constraint function.

Return type float

xmax(*x*)
This function calculates the maximum density value of all elements of this iteration.

Parameters **x** (*2D array size(nely, nelx)*) – Density distribution of this iteration.

Returns **xmax** – Maximum density values of this itteration after updating.

Return type 2D array size(nely, nelx)

xmin(*x*)
This function calculates the minimum density value of all elements of this iteration.

Parameters **x** (*2D array size(nely, nelx)*) – Density distribution of this iteration.

Returns **xmin** – Minimum density values of this iteration for the update scheme.

Return type 2D array size(nely, nelx)

4.2.2 Load Cases

This file contains the Load class that allows the generation of an object that contains geometric, mesh, loads and boundary conditions that belong to the load case. This version of the code is meant for local compliant maximization.

Bram Lagerweij Aerospace Structures and Materials Department TU Delft 2018

Parent Load Case

class src_Actuator.loads.**Load**(*nelx*, *nely*, *young*, *Emin*, *poisson*, *ext_stiff*)
Load parent class that contains the basic functions used in all load cases. This class and its children do cantain information about the load case conciderd in the optimisation. The load case consists of the mesh, the loads, and the boundaries conditions. The class is constructed such that new load cases can be generated simply by adding a child and changing the function related to the geometry, loads and boundaries.

Parameters

- **nelx** (*int*) – Number of elements in x direction.
- **nely** (*int*) – Number of elements in y direction.
- **young** (*float*) – Youngs modulus of the materias.
- **Emin** (*float*) – Artifical Youngs modulus of the material to ensure a stable FEA. It is used in the SIMP based material model.
- **poisson** (*float*) – Poisson ration of the material.

nelx
Number of elements in x direction.

Type int

nely
Number of elements in y direction.

Type int

young
Youngs modulus of the materias.

Type float

Emin
Artifical Youngs modulus of the material to ensure a stable FEA. It is used in the SIMP based material model.

Type float

poisson
Poisson ration of the material.

Type float

dim
Amount of dimensions concidered in the problem, set at 2.

Type int

ext_stiff
Extra stiffness to be added to global stiffness matrix. Due to interactions with meganisms outside design domain.

Type float

alldofs()
Returns a list with all degrees of freedom.

Returns **all** – List with numbers from 0 to the maximum degree of freedom number.

Return type 1-D list

displaceloc()
Returns a zero vector, there is supposed to be an 1 implemented at the index where displacment output should be maximised, such that u·l = u_out

Returns **l** – Empty for the governing class.

Return type 1-D column array length covering all degrees of freedom

edof()
Generates an array with the position of the nodes of each element in the global stiffness matrix.

Returns

- **edof** (*2-D array size(nelx*nely, 8)*) – The list with all elements and their degree of freedom numbers.
- **x_list** (*1-D array len(nelx*nely*8*8)*) – The list with the x indices of all ellements to be inserted into the global stiffniss matrix.
- **y_list** (*1-D array len(nelx*nely*8*8)*) – The list with the y indices of all ellements to be inserted into the global stiffniss matrix.

fixdofs()
Returns a list with indices that are fixed by the boundary conditions.

Returns **fix** – List with all the numbers of fixed degrees of freedom. This list is empty in this parrent class.

Return type 1-D list

force()
Returns an 1D array, the force vector of the loading condition.

Returns **f** – Empy force vector.

Return type 1-D column array length covering all degrees of freedom

freedofs()
Returns a list of arr indices that are not fixed

Returns free – List containing all elemens of alldogs except those that appear in the freedofs list.

Return type 1-D list

lk(*E*, *nu*)
Calculates the local siffness matrix depending on E and nu.

Parameters

- **E** (*float*) – Youngs modulus of the material.
- **nu** (*float*) – Poisson ratio of the material.

Returns ke – Local stiffness matrix.

Return type 2-D array size(8, 8)

node(*elx*, *ely*)
Calculates the topleft node number of the requested element.

Parameters

- **elx** (*int*) – X position of the conciderd element.
- **ely** (*int*) – Y position of the conciderd element.

Returns topleft – The node number of the top left node.

Return type int

nodes(*elx*, *ely*)
Calculates all node numbers of the requested element

Parameters

- **elx** (*int*) – X position of the conciderd element.
- **ely** (*int*) – Y position of the conciderd element.

Returns

- **n1** (*int*) – The node number of the top left node.
- **n2** (*int*) – The node number of the top right node.
- **n3** (*int*) – The node number of the bottom right node.
- **n4** (*int*) – The node number of the bottom left node.

passive()
Retuns three lists containing the location and magnitude of fixed density values

Returns

- **elx** (*1-D list*) – X coordinates of all passive elements, empty for the parrent class.
- **ely** (*1-D list*) – Y ccordinates of all passive elements, empty for the parrent class.
- **values** (*1-D list*) – Density values of all passive elements, empty for the parrent class.
- **fix_ele** (*1-D list*) – List with all element numbers that are allowed to change.

Child Load Cases

class src_Actuator.loads.**Inverter**(*nelx*, *nely*, *young*, *Emin*, *poisson*, *ext_stiff*)
Bases: *src_Actuator.loads.Load*

This child of the Load class represents a top half of the symetric inverter design used for MEMS actuators. It contains an positive horizontal force at the bottom left corner which causes a negative displacement at the bottom right corner.

No methods are added compared to the parrent class. Only the force, displaceloc and fixdof equations are changed to contain the propper values for the boundary conditions and optimisation objective.

displaceloc()
The maximisation should occur in negative x direction at the bottom right corner. Positive movement is thus in negative x direction.

Returns **l** – Value of -1 at the index related to the bottom right node.

Return type 1-D column array length covering all degrees of freedom

fixdofs()
The boundary conditions of this problem fixes the bottom of the desing space in y direction (due to symetry). While the topleft element is fixed in both x and y direction on the left side.

Returns **fix** – List with all the numbers of fixed degrees of freedom.

Return type 1-D list

force()
The force vector containts a load in positive x direction at the bottom left corner node.

Returns **f** – Value of 1 at the index related to the bottom left node.

Return type 1-D column array length covering all degrees of freedom

4.2.3 Finite Element Solvers

Finite element solvers for the displacement from stiffness matrix, force and adjoin vector. This version of the code is meant for local compliant maximization.

Bram Lagerweij Aerospace Structures and Materials Department TU Delft 2018

Parent Solver

class src_Actuator.fesolvers.**FESolver**(*verbose=False*)
This parent FEA class can only assemble the global stiffness matrix and exclude all fixed degrees of freedom from it. This stiffness csc-sparse stiffness matrix is assembled in the gk_freedof method. This class solves the FE problem with a sparse LU-solver based upon umfpack. This solver is slow and inefficient. It is however more robust.

For this local compliance (actuator) maximization this solver solves two problems, the equilibrium and the adjoint problem which will be required to compute the gradients.

Parameters **verbose** (*bool, optional*) – False if the FEA should not print updates

verbose
False if the FEA should not print updates.

Type bool

displace(*load*, *x*, *ke*, *kmin*, *penal*)
FE solver based upon the sparse SciPy solver that uses umfpack.

Parameters

- **load** (*object, child of the Loads class*) – The loadcase(s) considered for this optimisation problem.
- **x** (*2-D array size(nely, nelx)*) – Current density distribution.
- **ke** (*2-D array size(8, 8)*) – Local fully dense stiffness matrix.
- **kmin** (*2-D array size(8, 8)*) – Local stiffness matrix for an empty element.
- **penal** (*float*) – Material model penalisation (SIMP).

Returns

- **u** (*1-D column array shape(max(edof), 1)*) – The displacement vector.
- **lamba** (*1-D column array shape(max(edof), 1)*) – Adjoint equation solution.

gk_freedofs(*load*, *x*, *ke*, *kmin*, *penal*)
Generates the global stiffness matrix with deleted fixed degrees of freedom. It generates a list with stiffness values and their x and y indices in the global stiffness matrix. Some combination of x and y appear multiple times as the degree of freedom might appear in multiple elements of the FEA. The SciPy coo_matrix function adds them up at the background. At the location of the force introduction and displacement output an external stiffness is added due to stability reasons.

Parameters

- **load** (*object, child of the Loads class*) – The loadcase(s) considered for this optimisation problem.
- **x** (*2-D array size(nely, nelx)*) – Current density distribution.
- **ke** (*2-D array size(8, 8)*) – Local fully dense stiffness matrix.
- **kmin** (*2-D array size(8, 8)*) – Local stiffness matrix for an empty element.
- **penal** (*float*) – Material model penalisation (SIMP).

Returns **k** – Global stiffness matrix without fixed degrees of freedom.

Return type 2-D sparse csc matrix

Child Solvers

class src_Actuator.fesolvers.**CvxFEA**(*verbose=False*)
Bases: *src_Actuator.fesolvers.FESolver*

This parent FEA class can assemble the global stiffness matrix and solve the FE problem with a Supernodal Sparse Cholesky Factorization. It solves for both the equilibrium and adjoin problems.

verbose
False if the FEA should not print updates.

Type bool

displace(*load*, *x*, *ke*, *kmin*, *penal*)
FE solver based upon a Supernodal Sparse Cholesky Factorization. It requires the installation of the cvx module. It solves both the FEA equilibrium and adjoint problems.[1]

[1] Y. Chen, T. A. Davis, W. W. Hager, S. Rajamanickam, "Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate", ACM Transactions on Mathematical Software, 35(3), 22:1-22:14, 2008.

Parameters

- **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.
- **x** (*2-D array size(nely, nelx)*) – Current density distribution.
- **ke** (*2-D array size(8, 8)*) – Local fully dense stiffness matrix.
- **kmin** (*2-D array size(8, 8)*) – Local stiffness matrix for an empty element.
- **penal** (*float*) – Material model penalisation (SIMP).

Returns

- **u** (*1-D column array shape(max(edof), 1)*) – The displacement vector.
- **lamba** (*1-D column array shape(max(edof), 1)*) – Adjoint equation solution.

References

class src_Actuator.fesolvers.**CGFEA**(*verbose=False*)

Bases: *src_Actuator.fesolvers.FESolver*

This parent FEA class can assemble the global stiffness matrix and solve the FE problem with a sparse solver based upon a preconditioned conjugate gradient solver. The preconditioning is based upon the inverse of the diagonal of the stiffness matrix.

Recommendations

- Make the tolerance change over the iterations, low accuracy is required for first iteration, more accuracy for the later ones.
- Add more advanced preconditioned.
- Add gpu acceleration.

verbose
False if the FEA should not print updates.

Type bool

ufree_old
Displacement field of previous iteration.

Type array len(freedofs)

lambafree_old
Ajoint equation result of previous iteration.

Type array len(freedofs)

displace(*load*, *x*, *ke*, *kmin*, *penal*)
FE solver based upon the sparse SciPy solver that uses a preconditioned conjugate gradient solver, preconditioning is based upon the inverse of the diagonal of the stiffness matrix. Currently the relative tolerance is hardcoded as 1e-5. It solves both the equilibrium and adjoint problems.

Parameters

- **load** (*object, child of the Loads class*) – The loadcase(s) considered for this optimisation problem.
- **x** (*2-D array size(nely, nelx)*) – Current density distribution.
- **ke** (*2-D array size(8, 8)*) – Local fully dense stiffness matrix.

- **kmin** (*2-D array size(8, 8)*) – Local stiffness matrix for an empty element.
- **penal** (*float*) – Material model penalisation (SIMP).

Returns

- **u** (*1-D array len(max(edof)+1)*) – Displacement of all degrees of freedom
- **lamba** (*1-D column array shape(max(edof), 1)*) – Adjoint equation solution.

4.2.4 Optimization Module

Topology Optimization class that handles the iterations, objective functions, filters and update scheme. It requires to call upon a constraint, load case and FE solver classes. This version of the code is meant for local compliant maximization (Actuator design).

Bram Lagerweij Aerospace Structures and Materials Department TU Delft 2018

class src_Actuator.topopt.**Topopt**(*constraint*, *load*, *fesolver*, *verbose=False*)

This is the optimisation object itself. It contains the initialisation of the density distribution.

Parameters

- **constraint** (*object of DensityConstraint class*) – The constraints for this optimization problem.
- **load** (*object, child of the Loads class*) – The loadcase(s) considered for this optimisation problem.
- **fesolver** (*object, child of the CSCStiffnessMatrix class*) – The finite element solver.
- **verbose** (*bool, optional*) – Printing itteration results.

constraint

The constraints for this optimization problem.

Type object of DensityConstraint class

load

The loadcase(s) considered for this optimisation problem.

Type object, child of the Loads class

fesolver

The finite element solver.

Type object, child of the CSCStiffnessMatrix class

verbose

Printing iteration results.

Type bool

itr

Number of iterations performed

Type int

x

Array containing the current densities of every element.

Type 2-D array size(nely, nelx)

xold1
Flattened density distribution one iteration ago.

Type 1D array len(nelx*nely)

xold2
Flattened density distribution two iteration ago.

Type 1D array len(nelx*nely)

low
Column vector with the lower asymptotes, calculated and used in the MMA subproblem of the previous iteration.

Type 1D array len(nelx*nely)

upp
Column vector with the lower asymptotes, calculated and used in the MMA subproblem of the previous iteration.

Type 1D array len(nelx*nely)

densityfilt(*rmin*, *filt*)
Filters with a normalized convolution on the densities with a radius of rmin if:

```
>>> filt=='density'
```

Parameters

- **rmin** (*float*) – Filter size.
- **filt** (*str*) – The filter type that is selected, either 'sensitivity' or 'density'.

Returns **xf** – Filterd density distribution.

Return type 2-D array size(nely, nelx)

disp(*x*, *u*, *lamba*, *ke*, *penal*)
This function calculates displacement of the objective node and its sensitivity to the densities.

Parameters

- **x** (*2-D array size(nely, nelx)*) – Possibly filtered density distribution.
- **u** (*1-D array size(max(edof), 1)*) – Displacement of all degrees of freedom.
- **lamba** (*2-D array size(max(edof), 1)*) –
- **ke** (*2-D array size(8, 8)*) – Element stiffness matrix with full density.
- **penal** (*float*) – Material model penalisation (SIMP).

Returns

- **uout** (*float*) – Displacement objective.
- **duout** (*2-D array size(nely, nelx)*) – Displacement objective sensitivity to density changes.

iter(*penal*, *rmin*, *filt*)
This function performs one iteration of the topology optimisation problem. It

- loads the constraints,
- calculates the stiffness matrices,
- executes the density filter,

- executes the FEA solver,
- calls upon the displacement objective and its sensitivity calculation,
- executes the sensitivity filter,
- executes the MMA update scheme,
- and finally updates density distribution (design).

Parameters

- **penal** (*float*) – Material model penalisation (SIMP).
- **rmin** (*float*) – Filter size.
- **filt** (*str*) – The filter type that is selected, either 'sensitivity' or 'density'.

Returns

- **change** (*float*) – Largest difference between the new and old density distribution.
- **uout** (*float*) – Displacement at the objective node for the current design.

layout (*penal*, *rmin*, *delta*, *loopy*, *filt*, *history=False*)

Solves the topology optimisation problem by looping over the iter function.

Parameters

- **penal** (*float*) – Material model penalisation (SIMP).
- **rmin** (*float*) – Filter size.
- **delta** (*float*) – Convergence is roached when delta > change.
- **loopy** (*int*) – Amount of iteration allowed.
- **filt** (*str*) – The filter type that is selected, either 'sensitivity' or 'density'.
- **history** (*bool, optional*) – Do the intermediate results need to be stored.

Returns

- **xf** (*array size(nely, nelx)*) – Density distribution resulting from the optimisation.
- **xf_history** (*list of arrays len(iterations size(nely, nelx))*) – List with the density distributions of all iterations, None when history != True.

mma (*m*, *n*, *itr*, *xval*, *xmin*, *xmax*, *xold1*, *xold2*, *f0val*, *df0dx*, *fval*, *dfdx*, *low*, *upp*, *a0*, *a*, *c*, *d*)

This function mmasub performs one MMA-iteration, aimed at solving the nonlinear programming prob-

lem:

$$\min f_0(x)$$

$$+a_0 z + \sum_{i=1}^{m} \left(c_i y_i + \frac{1}{2} d_i y_i^2 \right)$$

$$\text{s.t.}$$

$$f_i(x) - a_i z - y_i \leq 0 \qquad i \in \{1, 2, \ldots, m\}$$

$$x_{\min} \geq x_j \geq x_{\max} j \in \{1, 2, \ldots, n\}$$

$$y_i \leq 0 i \in \{1, 2, \ldots, m\}$$

$$z \geq 0$$

Parameters

- **m** (*int*) – The number of general constraints.
- **n** (*int*) – The number of variables x_j.
- **itr** (*int*) – Current iteration number (=1 the first time mmasub is called).
- **xval** (*1-D array len(n)*) – Vector with the current values of the variables x_j.
- **xmin** (*1-D array len(n)*) – Vector with the lower bounds for the variables x_j.
- **xmax** (*1-D array len(n)*) – Vector with the upper bounds for the variables x_j.
- **xold1** (*1-D array len (n)*) – xval, one iteration ago when iter>1, zero othewise.
- **xold2** (*1-D array len (n)*) – xval, two iteration ago when iter>2, zero othewise.
- **f0val** (*float*) – The value of the objective function f_0 at xval.
- **df0dx** (*1-D array len(n)*) – Vector with the derivatives of the objective function f_0 with respect to the variables x_j, calculated at xval.
- **fval** (*1-D array len(m)*) – Vector with the values of the constraint functions f_i, calculated at xval.
- **dfdx** (*2-D array size(m x n)*) – (m x n)-matrix with the derivatives of the constraint functions f_i. with respect to the variables x_j, calculated at xval.
- **low** (*1-D array len(n)*) – Vector with the lower asymptotes from the previous iteration (provided that iter>1).
- **upp** (*1-D array len(n)*) – Vector with the upper asymptotes from the previous iteration (provided that iter>1).
- **a0** (*float*) – The constants a_0 in the term $a_0 z$.
- **a** (*1-D array len(m)*) – Vector with the constants $a_i 1 in the terms : math :$.
- **c** (*1-D array len(m)*) – Vector with the constants c_i in the terms $c_i * y_i$.
- **d** (*1-D array len(m)*) – Vector with the constants d_i in the terms $0.5 d_i (y_i)^2$.

Returns

- **xmma** (*1-D array len(n)*) – Column vector with the optimal values of the variables x_j in the current MMA subproblem.

- **low** (*1-D array len(n)*) – Column vector with the lower asymptotes, calculated and used in the current MMA subproblem.
- **upp** (*1-D array len(n)*) – Column vector with the upper asymptotes, calculated and used in the current MMA subproblem.
- *Version September 2007 (and a small change August 2008)*
- *Krister Svanberg <krille@math.kth.se>*
- *Department of Mathematics KTH, SE-10044 Stockholm, Sweden.*
- *Translated to python 3 by A.J.J. Lagerweij TU Delft June 2018*

sensitivityfilt (*x*, *rmin*, *duout*, *filt*)

Filters with a normalized convolution on the sensitivity with a radius of rmin if:

```
>>> filt=='sensitivity'
```

Parameters

- **x** (*2-D array size(nely, nelx)*) – Current density ditribution.
- **duout** (*2-D array size(nely, nelx)*) – Displacement objective sensitivity to density changes.
- **rmin** (*float*) – Filter size.
- **filt** (*str*) – The filter type that is selected, either 'sensitivity' or 'density'.

Returns **duoutf** – Filterd sensitivity distribution.

Return type 2-D array size(nely, nelx)

solvemma (*m*, *n*, *epsimin*, *low*, *upp*, *alfa*, *beta*, *p0*, *q0*, *P*, *Q*, *a0*, *a*, *b*, *c*, *d*)

This function solves the MMA subproblem with a primal-dual Newton method.

$$\min \sum_{j-1}^{n} \left(\frac{p_{0j}^{(k)}}{U_j^{(k)} - x_j} + \frac{q_{0j}^{(k)}}{x_j - L_j^{(k)}} \right) + a_0 z + \sum_{i=1}^{m} \left(c_i y_i + \frac{1}{2} d_i y_i^2 \right)$$

s.t.

$$\sum_{j-1}^{n} \left(\frac{p_{ij}^{(k)}}{U_j^{(k)} - x_j} + \frac{q_{ij}^{(k)}}{x_j - L_j^{(k)}} \right) - a_i z - y_i \leq b_i$$

$$\alpha_j \geq x_j \geq \beta_j$$

$$z \geq 0$$

Returns **x** – Column vector with the optimal values of the variables x_j in the current MMA subproblem.

Return type 1-D array len(n)

4.2.5 Plotting Module

Plotting the simulated TopOpt geometry with boundary conditions and loads.

Bram Lagerweij Aerospace Structures and Materials Department TU Delft 2018

class src_Actuator.plotting.**Plot**(*load*, *title=None*)

This class contains functions that allows the visualisation of the TopOpt algorithm. It can print the density distribution, the boundary conditions and the forces.

Parameters

- **load** (*object, child of the Loads class*) – The loadcase(s) considered for this optimisation problem.
- **title** (*str*) – Title of the plot if required.

nelx

Number of elements in x direction.

Type int

nely

Number of elements in y direction.

Type int

fig

An empty figure of size nelx/10 and nely/10*1.2 inch.

Type matplotlib.pyplot figure

ax

The axis system that belongs to fig.

Type matplotlib.pyplot axis

images

This list contains all density distributions that need to be plotted.

Type 1-D list with imshow objects

add(*x*, *animated=False*)

Adding a plot of the density distribution to the figure.

Parameters

- **x** (*2-D array size(nely, nelx)*) – The density distribution.
- **animated** (*bool, optional*) – An animated figure is generated when history = True.

boundary(*load*)

Plotting the boundary conditions.

Parameters **load** (*object, child of the Loads class*) – The loadcase(s) considered for this optimisation problem.

loading(*load*)

Plotting the loading conditions.

Parameters **load** (*object, child of the Loads class*) – The loadcase(s) considered for this optimisation problem.

save (*filename*, *fps=10*)
Saving an plot in svg or mp4 format, depending on the length of the images list. The FasterFFMpegWriter is used when videos are generated. These videos are encoded with a hardware accelerated h264 codec with the .mp4 file format. Other codecs and encoders can be set within the function itself.

Parameters

- **filename** (*str*) – Name of the file, excluding the file extension.
- **fps** (*int, optional*) – Amount of frames per second if the plots are animations.

show()
Showing the plot in a window.

class src_Actuator.plotting.**FasterFFMpegWriter**(***kwargs*)
Bases: matplotlib.animation.FFMpegWriter

FFMpeg-pipe writer bypassing figure.savefig. To improof saving speed

classmethod bin_path()
Return the binary path to the commandline tool used by a specific subclass. This is a class method so that the tool can be looked for before making a particular MovieWriter subclass available.

cleanup()
Clean-up and collect the process used to write the movie file.

finish()
Finish any processing for writing the movie.

frame_size
A tuple (width, height) in pixels of a movie frame.

grab_frame(***savefig_kwargs*)
Grab the image information from the figure and save as a movie frame.

Doesn't use savefig to be faster: savefig_kwargs will be ignored.

classmethod isAvailable()
Check to see if a MovieWriter subclass is actually available.

saving (*fig*, *outfile*, *dpi*, **args*, ***kwargs*)
Context manager to facilitate writing the movie file.

**args, **kw* are any parameters that should be passed to *setup*.

setup (*fig*, *outfile*, *dpi=None*)
Perform setup for writing the movie file.

Parameters

- **fig** (*~matplotlib.figure.Figure*) – The figure object that contains the information for frames
- **outfile** (*str*) – The filename of the resulting movie file
- **dpi** (*int, optional*) – The DPI (or resolution) for the file. This controls the size in pixels of the resulting movie file. Default is fig.dpi.

4.3 Stress Intensity Factor Minimization

In this stress intensity factor minimization a structure with crack is optimized to have minimal crack growth rate. Thow this works is discussed in *Stress Intensity Factor Minimization* An example as how to use the optimization is shown in

an example optimization example.py

- *Density Constraints*
- *Load Cases*
- *Finite Element Solvers*
- *Optimization Module*
- *Plotting Module*

4.3.1 Density Constraints

Constraints class used to specify the density constraints of the topology optimisation problem. It contains functions for minimum and maximum element density in the upcomming iteration and the magnitude of the volume constraint function itself of the current design. This version of the code is used for stress intensity minimisation.

Bram Lagerweij Aerospace Structures and Materials Department TU Delft 2018

class src_StressIntensity.constraints.**DensityConstraint**(*load*, *move*, *volume_frac*, *density_min=0.0*, *density_max=1.0*)

This object relates to the constraints used in this optimization. It can be used for the MMA updatescheme to derive what the limit is for all element densities at every itteration. The class itself is not changed by the itterations.

Parameters

- **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem
- **move** (*float*) – Maximum change in density of an element over 1 itteration.
- **volume_frac** (*float*) – Maximum volume that can be filled with material.
- **volume_derivative** (*2D array size(1, nelx*nely)*) – Sensityvity of the density constraint to the density in each element.
- **density_min** (*float (optional)*) – Minumum density, set at 0.0 if not specified.
- **density_max** (*float (optional)*) – Maximum density, set at 0.0 if not specified.

nelx
Number of elements in x direction.

Type int

nely
Number of elements in y direction.

Type int

move
Maximum change in density of an element over 1 itteration.

Type float

volume_frac
Maximum volume that can be filled with material.

Type float

volume_derivative
Sensityvity of the density constraint to the density in each element.

Type 2D array size(1, nelx*nely)

density_min
Minumum density, set at 0.0 if not specified.

Type float, optional

density_max
Maximum density, set at 0.0 if not specified.

Type float, optional

current_volconstrain(*x*)
Calculates the current magnitude of the volume constraint funcion:

$$V_{\text{constraint}} = \frac{\sum v_e X_e}{V_{\max}} - 1$$

Parameters **x** (*2D array size(nely, nelx)*) – Density distribution of this itteration.

Returns **curvol** – Curent value of the density constraint function.

Return type float

xmax(*x*)
This function calculates the maximum density value of all ellements of this itteration.

Parameters **x** (*2D array size(nely, nelx)*) – Density distribution of this itteration.

Returns **xmax** – Maximum density values of this itteration after updating.

Return type 2D array size(nely, nelx)

xmin(*x*)
This function calculates the minimum density value of all ellements of this itteration.

Parameters **x** (*2D array size(nely, nelx)*) – Density distribution of this itteration.

Returns **xmin** – Minimum density values of this itteration for the update scheme.

Return type 2D array size(nely, nelx)

4.3.2 Load Cases

This file containts the Load class that allows the generation of an object that contains geometric, mesh, loads and boundary conditions that belong to the load case. This version of the code is meant for stress intensity minimization.

Bram Lagerweij Aerospace Structures and Materials Department TU Delft 2018

Parent Load Case

class src_StressIntensity.loads.**Load**(*nelx*, *nely*, *young*, *Emin*, *poisson*, *ext_stiff*, *hoe*)
Load parent class that contains the basic functions used in all load cases. This class and its children do cantain information about the load case concicerd in the optimisation. The load case consists of the mesh, the loads, and the boundaries conditions. The class is constructed such that new load cases can be generated simply by adding a child and changing the function related to the geometry, loads and boundaries.

Parameters

- **nelx** (*int*) – Number of elements in x direction.

- **nely** (*int*) – Number of elements in y direction.
- **young** (*float*) – Youngs modulus of the materias.
- **Emin** (*float*) – Artifical Youngs modulus of the material to ensure a stable FEA. It is used in the SIMP based material model.
- **poisson** (*float*) – Poisson ration of the material.
- **ext_stiff** (*float*) – Extra stiffness to be added to global stiffness matrix. Due to interactions with meganisms outside design domain.
- **hoe** (*list*) – List of lists with for every cracklength the x end y element locations that need to be enriched.

nelx
Number of elements in x direction.

Type int

nely
Number of elements in y direction.

Type int

dim
Amount of dimensions conciderd in the problem, set at 2.

Type int

edof
The list with all elements and their degree of freedom numbers.

Type 2-D list size(nelx*nely, # degrees of freedom per element)

x_list
The list with the x indices of all ellements to be inserted into the global stiffniss matrix.

Type 1-D array

y_list
The list with the y indices of all ellements to be inserted into the global stiffniss matrix.

Type 1-D array

num_dofs
Amount of degrees of freedom.

Type int

young
Youngs modulus of the materias.

Type float

Emin
Artifical Youngs modulus of the material to ensure a stable FEA. It is used in the SIMP based material model.

Type float

poisson
Poisson ration of the material.

Type float

k_list
List with element stiffness matrices of full density.

Type list len(nelx*nely)

kmin_list
List with element stifness matrices at 0 density.

Type list len(nelx*nely)

ext_stiff
Extra stiffness to be added to global stiffness matrix. Due to interactions with meganisms outside design domain.

Type float

alldofs()
Returns a list with all degrees of freedom.

Returns **all** – List with numbers from 0 to the maximum degree of freedom number.

Return type 1-D list

edofcalc(*hoe*)
Generates an array with the position of the nodes of each element in the global stiffness matrix. This takes the Higher Order Elements in account.

Returns

- **edof** (*2-D list size(nelx*nely, # degrees of freedom per element)*) – The list with all elements and their degree of freedom numbers.
- **x_list** (*1-D array*) – The list with the x indices of all ellements to be inserted into the global stiffniss matrix.
- **y_list** (*1-D array*) – The list with the y indices of all ellements to be inserted into the global stiffniss matrix.
- **num_dofs** (*int*) – The amount of degrees of freedom.

fixdofs()
Returns a list with indices that are fixed by the boundary conditions.

Returns **fix** – List with all the numbers of fixed degrees of freedom. This list is empty in this parrent class.

Return type 1-D list

force()
Returns an 1D array, the force vector of the loading condition. Note that the possitive y direction is downwards, thus a negative force in y direction is required for a upward load.

Returns **f** – Empy force vector.

Return type 1-D column array length covering all degrees of freedom

freedofs()
Returns a list of arr indices that are not fixed

Returns **free** – List containing all elemens of alldogs except those that appear in the freedofs list.

Return type 1-D list

import_stiffness(*elementtype*, *E*, *nu*)
This function imports a matrix from a csv file that has variables to the material properties. The correct material properties are added.

Parameters

- **elementtype** (*str*) – Describes what .csv file should be used for the import.
- **E** (*float*) – Youngs modulus of the material.
- **nu** (*float*) – Poissons ratio of the material.

Returns **lk** – Element stiffness matrix

Return type array size(dofs, dofs)

kiloc()
The location of the stress intensity factor KI can be found at the second last index.

Returns **l** – Zeros except for the second last index.

Return type 1-D column array length covering all degrees of freedom

lk(*E*, *nu*)
Generates a list with all element stiffness matrices. It differenciates between the element types used.

Parameters

- **E** (*float*) – Youngs modulus of the material.
- **nu** (*float*) – Poissons ratio of the material.
- **Returns** –
- **k** (*list len(nelx*nely)*) – Returns a list with all local stiffness matrices.

node(*elx*, *ely*)
Calculates the topleft node number of the requested element. Does not toke Higher Order Elements in account.

Parameters

- **elx** (*int*) – X position of the conciderd element.
- **ely** (*int*) – Y position of the conciderd element.

Returns **topleft** – The node number of the top left node.

Return type int

nodes(*elx*, *ely*)
Calculates all node numbers of the requested element. Does not take Higher Order Elements in account.

Parameters

- **elx** (*int*) – X position of the conciderd element.
- **ely** (*int*) – Y position of the conciderd element.

Returns

- **n0** (*int*) – The node number of the bottom left node.
- **n1** (*int*) – The node number of the bottom right node.
- **n2** (*int*) – The node number of the top left node.
- **n3** (*int*) – The node number of the top right node.

passive()
Retuns three lists containing the location and magnitude of fixed density values

Returns

- **elx** (*1-D list*) – X coordinates of all passive elements, empty for the parrent class.
- **ely** (*1-D list*) – Y ccordinates of all passive elements, empty for the parrent class.
- **values** (*1-D list*) – Density values of all passive elements, empty for the parrent class.
- **fix_ele** (*1-D list*) – List with all element numbers that are allowed to change.

reset_Kij()
Resets the global variable Kij. This is neccesary as function import_stiffness will not clean up its local variables itself.

Child Load Cases

class src_StressIntensity.loads.**EdgeCrack**(*nelx*, *nely*, *crack_length*, *young*, *Emin*, *poisson*, *ext_stiff*)
Bases: *src_StressIntensity.loads.Load*

This child class of Load class represents the symetric top half of an edge crack. The crack is positioned to the bottom left and propegates towards the right. Special elements are placed around the crack tip. The plate is subjected to a distributed tensile load ($\sigma = 1$) on the top.

For a perfectly flat plate analytical expressions for K_I are known.[2]

The stress intensity factors calculated can be be interperted in two ways:

1. Without schaling. This means that all elements have a size of 2 length units.
2. With schaling, comparison to reality should be based upon.

$$K^{\text{Real}} = K^{\text{FEA}}(\sigma = 1)\sigma^{\text{Real}}\sqrt{\frac{a^{\text{Real}}}{2a^{\text{FEA}}}}$$

where a^{FEA} is the cracklength in number of elements.

Parameters

- **nelx** (*int*) – Number of elements in x direction.
- **nely** (*int*) – Number of elements in y direction.
- **crack_length** (*int*) – Crack lengs conciderd.
- **young** (*float*) – Youngs modulus of the materias.
- **Emin** (*float*) – Artifical Youngs modulus of the material to ensure a stable FEA. It is used in the SIMP based material model.
- **poisson** (*float*) – Poisson ration of the material.
- **ext_stiff** (*float*) – Extra stiffness to be added to global stiffness matrix. Due to interactions with meganisms outside design domain.

crack_length
Is the amount of elements that the crack is long.

[2] Tada, H., Paris, P., & Irwin, G. (2000). "Part II 2.10-2.12 The Single Edge Notch Test Specimen", The stress analysis of cracks handbook (3rd ed.). New York: ASME Press, pp:52-54.

Type int

hoe_type
List containing element type for each enriched element.

Type list len(2)

References

fixdofs()
The boundary conditions limmit y-translation at the bottom of the design space (due to symetry) and x-translations at the top (due to the clamps)

Returns **fix** – List with all the numbers of fixed degrees of freedom.

Return type 1-D list

force()
The top of the design space is pulled upwards by 1MPa. This means that the nodal forces are 2 upwards, except for the top corners they have a load of 1 only.

Returns **f** – Force vector.

Return type 1-D column array length covering all degrees of freedom

passive()
Retuns three lists containing the location and magnitude of fixed density values. The elements around the crack tip are fixed at a density of one.

Returns

- **elx** (*1-D list*) – X coordinates of all passive elements, empty for the parrent class.
- **ely** (*1-D list*) – Y ccordinates of all passive elements, empty for the parrent class.
- **values** (*1-D list*) – Density values of all passive elements, empty for the parrent class.
- **fix_ele** (*1-D list*) – List with all element numbers that are allowed to change.

class src_StressIntensity.loads.**DoubleEdgeCrack**(*nelx*, *young*, *Emin*, *poisson*, *ext_stiff*)
Bases: *src_StressIntensity.loads.Load*

This child class of Load class represents the symetric top rigth quarter of an double edge crack plate. The crack is positioned to the bottom left and propegatestowards the right. Special elements are placed around the crack tip. The plate is subjected to a distributed tensile load (σ=1) on the top.

For a perfectly flat plate analytical expressions for K_I are known.[3]

The stress intensity factors calculated can be be interperted in two ways:

1. Without schaling. This means that all elements have a size of 2 length units.
2. With schaling, comparison to reality should be based upon.

$$K^{\text{Real}} = K^{\text{FEA}}(\sigma = 1)\sigma^{\text{Real}}\sqrt{\frac{a^{\text{Real}}}{2a^{\text{FEA}}}}$$

where a^{FEA} is the cracklength in number of elements.

Parameters

[3] Tada, H., Paris, P., & Irwin, G. (2000). "Part II 2.6-2.9a The Double Edge Notch Test Specimen", The stress analysis of cracks handbook (3rd ed.). New York: ASME Press, pp:46-51.

- **nelx** (*int*) – Number of elements in x direction.
- **young** (*float*) – Youngs modulus of the materias.
- **Emin** (*float*) – Artifical Youngs modulus of the material to ensure a stable FEA. It is used in the SIMP based material model.
- **poisson** (*float*) – Poisson ration of the material.
- **ext_stiff** (*float*) – Extra stiffness to be added to global stiffness matrix. Due to interactions with meganisms outside design domain.

nely
Number of y elements, this is now a function of nelx.

Type int

crack_length
Is the amount of elements that the crack is long, this is a function of nelx.

Type int

hoe_type
List containging the type of enriched element.

Type list len(2)

References

fixdofs()
The right side is fixed in x direction (symetry around the y axis) while the bottom side is fixed in y direction (symetry around the x axis).

Returns **fix** – List with all the numbers of fixed degrees of freedom.

Return type 1-D list

force()
The top of the design space is pulled upwards by 1MPa. This means that the nodal forces are 2 upwards, except for the top left corner has a load of 1 only.

Returns **f** – Force vector

Return type 1-D column array length covering all degrees of freedom

passive()
Retuns three lists containing the location and magnitude of fixed density values. The elements around the crack tip are fixed at a density of one.

Returns

- **elx** (*1-D list*) – X coordinates of all passive elements, empty for the parrent class.
- **ely** (*1-D list*) – Y ccordinates of all passive elements, empty for the parrent class.
- **values** (*1-D list*) – Density values of all passive elements, empty for the parrent class.
- **fix_ele** (*1-D list*) – List with all element numbers that are allowed to change.

class src_StressIntensity.loads.**CompactTension**(*nelx*, *crack_length*, *young*, *Emin*, *poisson*, *ext_stiff*, *pas_loc=None*)

Bases: *src_StressIntensity.loads.Load*

This child class of Load class represents the symetric top half of an compact tension specimen. The crack is positioned to the bottom left and propegatestowards the right. Special elements are placed around the crack tip. The plate is subjected to upwards load of one. The design follows the ASTM standard.[4]

For a perfectly flat plate analytical expressions for K_I do exist.[5]

The stress intensity factors calculated can be be interperted in two ways: 1. Without schaling. This means that all elements have a size of 2 length units. 2. With schaling, comparison to reality should be based upon.

$$K^{\text{Real}} = K^{\text{FEA}}(F = 1)F^{\text{Real}}\sqrt{\frac{2W^{\text{FEA}}}{W^{\text{Real}}}}$$

where W^{FEA} is the width in number of elements.

Parameters

- **nelx** (*int*) – Number of elements in x direction.
- **crack_length** (*int*) – Crack length conciderd
- **young** (*float*) – Youngs modulus of the materias.
- **Emin** (*float*) – Artifical Youngs modulus of the material to ensure a stable FEA. It is used in the SIMP based material model.
- **poisson** (*float*) – Poisson ration of the material.
- **ext_stiff** (*float*) – Extra stiffness to be added to global stiffness matrix. Due to interactions with meganisms outside design domain.
- **pas_loc** (*string*) – Location/Name of the .npy file that contains passive background.

nely
Number of y elements, this is now a function of nelx.

Type int

crack_length
Is the amount of elements that the crack is long.

Type int

hoe
List containing the x end y element locations that need to be enriched.

Type list len(2)

hoe_type
List containging the type of enriched element.

Type list len(2)

References

fixdofs()
The bottom of the design space is fixed in y direction (due to symetry around the x axis). While at the location that the load is introduced x translations are constraint.

[4] ASTM Standard E647-15e1, "Standard Test Method for Measurement of Fatigue Crack Growth Rates," ASTM Book of Standards, vol. 0.30.1, 2015.

[5] Tada, H., Paris, P., & Irwin, G. (2000). "Part II 2.19-2.21 The Compact Tension Test Specimen", The stress analysis of cracks handbook (3rd ed.). New York: ASME Press, pp:61-63.

Returns **fix** – List with all the numbers of fixed degrees of freedom.

Return type 1-D list

force()
The ASTM standard requires the force to be located approx. 1/5 of nelx and at 0.195 * nely from the top.

Returns **f** – Force vector

Return type 1-D column array length covering all degrees of freedom

passive()
Retuns three lists containing the location and magnitude of fixed density values. The elements around the crack tip are fixed at a density of one.

Returns

- **elx** (*1-D list*) – X coordinates of all passive elements, empty for the parrent class.
- **ely** (*1-D list*) – Y ccordinates of all passive elements, empty for the parrent class.
- **values** (*1-D list*) – Density values of all passive elements, empty for the parrent class.
- **fix_ele** (*1-D list*) – List with all element numbers that are allowed to change.

4.3.3 Finite Element Solvers

Finite element solvers for the displacement from stiffness matrix, force and adjoint vector. This version of the code is meant for stress intensity minimization.

Bram Lagerweij Aerospace Structures and Materials Department TU Delft 2018

Parent Solver

class src_StressIntensity.fesolvers.**FESolver**(*verbose=False*)
This parent FEA class can only assemble the global stiffness matrix and exclude all fixed degrees of freedom from it. This stiffenss csc-sparse stiffness matrix is assebled in the gk_freedof method. This class solves the FE problem with a sparse LU-solver based upon umfpack. This solver is slow and inefficient. It is however more robust.

For this local compliance (actuator) maximization this solver solves two problems, the equalibrum and the adjoint problem which will be required to compute the gradients.

Parameters **verbose** (*bool, optional*) – False if the FEA should not print updates

verbose
False if the FEA should not print updates.

Type bool

displace(*load*, *x*, *ke*, *kmin*, *penal*)
FE solver based upon the sparse SciPy solver that uses umfpack.

Parameters

- **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.
- **x** (*2-D array size(nely, nelx)*) – Current density distribution.
- **ke** (*2-D array size(8, 8)*) – Local fully dense stiffnes matrix.
- **kmin** (*2-D array size(8, 8)*) – Local stiffness matrix for an empty element.

- **penal** (*float*) – Material model penalisation (SIMP).

Returns

- **u** (*1-D column array shape(max(edof), 1)*) – The displacement vector.
- **lamba** (*1-D column array shape(max(edof), 1)*) – Adjoint equation solution.

gk_freedofs(*load*, *x*, *ke*, *kmin*, *penal*)

Generates the global stiffness matrix with deleted fixed degrees of freedom. It generates a list with stiffness values and their x and y indices in the global stiffness matrix. Some combination of x and y appear multiple times as the degree of freedom might apear in multiple elements of the FEA. The SciPy coo_matrix function adds them up at the background. At the location of the force introduction and displacement output an external stiffness is added due to stability reasons.

Parameters

- **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.
- **x** (*2-D array size(nely, nelx)*) – Current density distribution.
- **ke** (*list len(nelx*nely)*) – List with all element stiffness matrixes for full dense material.
- **kmin** (*list len(nelx*nely)*) – List with all element stiffness matrixes for empty material.
- **penal** (*float*) – Material model penalisation (SIMP).

Returns **k** – Global stiffness matrix without fixed degrees of freedom.

Return type 2-D sparse csc matrix

Child Solvers

class src_StressIntensity.fesolvers.**CvxFEA**(*verbose=False*)

Bases: *src_StressIntensity.fesolvers.FESolver*

This parent FEA class can assemble the global stiffness matrix and solve the FE problem with a Supernodal Sparse Cholesky Factorization. It solves for both the equalibrium and adjoint problems.

verbose

False if the FEA should not print updates.

Type bool

displace(*load*, *x*, *ke*, *kmin*, *penal*)

FE solver based upon a Supernodal Sparse Cholesky Factorization. It requires the instalation of the cvx module. It solves both the FEA equalibrium and adjoint problems.[1]

Parameters

- **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.
- **x** (*2-D array size(nely, nelx)*) – Current density distribution.
- **ke** (*2-D array size(8, 8)*) – Local fully dense stiffnes matrix.
- **kmin** (*2-D array size(8, 8)*) – Local stiffness matrix for an empty element.

[1] Y. Chen, T. A. Davis, W. W. Hager, S. Rajamanickam, "Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate", ACM Transactions on Mathematical Software, 35(3), 22:1-22:14, 2008.

- **penal** (*float*) – Material model penalisation (SIMP).

Returns

- **u** (*1-D column array shape(max(edof), 1)*) – The displacement vector.
- **lamba** (*1-D column array shape(max(edof), 1)*) – Adjoint equation solution.

References

class src_StressIntensity.fesolvers.**CGFEA**(*verbose=False*)
Bases: *src_StressIntensity.fesolvers.FESolver*

This parent FEA class can assemble the global stiffness matrix and solve the FE problem with a sparse solver based upon a preconditioned conjugate gradient solver. The preconditioning is based upon the inverse of the diagonal of the stiffness matrix.

Recomendations

- Make the tolerance change over the iterations, low accuracy is required for first itteration, more accuracy for the later ones.
- Add more advanced preconditioner.
- Add gpu accerelation.

verbose
False if the FEA should not print updates.

Type bool

ufree_old
Displacement field of previous iteration.

Type array len(freedofs)

lambafree_old
Ajoint equation result of previous iteration.

Type array len(freedofs)

displace(*load*, *x*, *ke*, *kmin*, *penal*)
FE solver based upon the sparse SciPy solver that uses a preconditioned conjugate gradient solver, preconditioning is based upon the inverse of the diagonal of the stiffness matrix. Currently the relative tolerance is hardcoded as 1e-5. It solves both the equalibrium and adjoint problems.

Parameters

- **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.
- **x** (*2-D array size(nely, nelx)*) – Current density distribution.
- **ke** (*2-D array size(8, 8)*) – Local fully dense stiffnes matrix.
- **kmin** (*2-D array size(8, 8)*) – Local stiffness matrix for an empty element.
- **penal** (*float*) – Material model penalisation (SIMP).

Returns

- **u** (*1-D array len(max(edof)+1)*) – Displacement of all degrees of freedom
- **lamba** (*1-D column array shape(max(edof), 1)*) – Adjoint equation solution.

4.3.4 Optimization Module

Topology Optimization class that handles the itterations, objective functions, filters and update scheme. It requires to call upon a constraint, load case and FE solver classes. This version of the code is meant for stress intensity factor minimization.

Bram Lagerweij Aerospace Structures and Materials Department TU Delft 2018

class src_StressIntensity.topopt.**Topopt**(*constraint*, *load*, *fesolver*, *verbose=False*, *history=False*, *x0_loc=None*)

This is the optimisation object itself. It contains the initialisation of the density distribution.

Parameters

- **constraint** (*object of DensityConstraint class*) – The constraints for this optimization problem.
- **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.
- **fesolver** (*object, child of the CSCStiffnessMatrix class*) – The finite element solver.
- **verbose** (*bool, optional*) – Printing itteration results.
- **x0_loc** (*str, optional*) – Set initial design with numpy '.npy' file location.
- **history** (*boolean, optional*) – Saving a history array or not.

constraint
The constraints for this optimization problem.

Type object of DensityConstraint class

load
The loadcase(s) considerd for this optimisation problem.

Type object, child of the Loads class

fesolver
The finite element solver.

Type object, child of the CSCStiffnessMatrix class

verbose
Printing itteration results.

Type bool

itr
Number of iterations performed

Type int

free_ele
All element nubers that ar allowed to change.

Type 1-D list

x
Array containing the current densities of every element.

Type 2-D array size(nely, nelx)

xold1
Flattend density distribution one iteration ago.

Type 1D array len(nelx*nely)

xold2
Flattend density distribution two iteration ago.

Type 1D array len(nelx*nely)

low
Column vector with the lower asymptotes, calculated and used in the MMA subproblem of the previous itteration.

Type 1D array len(nelx*nely)

upp
Column vector with the lower asymptotes, calculated and used in the MMA subproblem of the previous itteration.

Type 1D array len(nelx*nely)

densityfilt (*rmin*, *filt*)
Filters with a normalized convolution on the densities with a radius of rmin if:

```
>>> filt=='density'
```

The relusting geometry retains passive elements.

Parameters

- **rmin** (*float*) – Filter size.
- **filt** (*str*) – The filter type that is selected, either 'sensitivity' or 'density'.

Returns **xf** – Filterd density distribution.

Return type 2-D array size(nely, nelx)

iter (*penal*, *rmin*, *filt*)
This funcion performs one itteration of the topology optimisation problem. It

- loads the constraints,
- calculates the stiffness matrices,
- executes the density filter,
- executes the FEA solver,
- calls upon the displacment objective and its sensitivity calculation,
- executes the sensitivity filter,
- executes the MMA update scheme,
- and finaly updates density distribution (design).

Parameters

- **penal** (*float*) – Material model penalisation (SIMP).
- **rmin** (*float*) – Filter size.
- **filt** (*str*) – The filter type that is selected, either 'sensitivity' or 'density'.

Returns

- **change** (*float*) – Largest difference between the new and old density distribution.
- **ki** (*float*) – Stress intensity factor for the current design.

ki(*x*, *u*, *lamba*, *ke*, *penal*)
This fuction calculates displacement of the objective node and its sensitivity to the densities.

Parameters

- **x** (*2-D array size(nely, nelx)*) – Possibly filterd density distribution.
- **u** (*1-D array size(max(edof), 1)*) – Displacement of all degrees of freedom.
- **lamba** (*2-D array size(max(edof), 1)*) –
- **ke** (*2-D array size(8, 8)*) – Element stiffness matrix with full density.
- **penal** (*float*) – Material model penalisation (SIMP).

Returns

- **ki** (*float*) – Displacement objective.
- **dki** (*2-D array size(nely, nelx)*) – Displacement objective sensitivity to density changes.

layout(*penal*, *rmin*, *delta*, *loopy*, *filt*)
Solves the topology optimisation problem by looping over the iter function.

Parameters

- **penal** (*float*) – Material model penalisation (SIMP).
- **rmin** (*float*) – Filter size.
- **delta** (*float*) – Convergence is roached when delta > change.
- **loopy** (*int*) – Amount of iteration allowed.
- **filt** (*str*) – The filter type that is selected, either 'sensitivity' or 'density'.
- **history** (*bool*) – Do the intermediate results need to be stored.

Returns

- **xf** (*array size(nely, nelx)*) – Density distribution resulting from the optimisation.
- **xf_history** (*list of arrays len(itterations size(nely, nelx))*) – List with the density distributions of all itterations, None when history != True.
- **ki** (*float*) – Stress intensity factor final design.

mma(*m*, *n*, *itr*, *xval*, *xmin*, *xmax*, *xold1*, *xold2*, *f0val*, *df0dx*, *fval*, *dfdx*, *low*, *upp*, *a0*, *a*, *c*, *d*)
This function mmasub performs one MMA-iteration, aimed at solving the nonlinear programming problem:

$$\min f_0(x)$$

$$+a_0 z + \sum_{i=1}^{m} \left(c_i y_i + \frac{1}{2} d_i y_i^2 \right)$$

$$\text{s.t.}$$

$$f_i(x) - a_i z - y_i \leq 0 \qquad i \in \{1, 2, \ldots, m\}$$

$$x_{\min} \geq x_j \geq x_{\max} j \in \{1, 2, \ldots, n\}$$

$$y_i \leq 0 i \in \{1, 2, \ldots, m\}$$

$$z \geq 0$$

Parameters

- **m** (*int*) – The number of general constraints.
- **n** (*int*) – The number of variables x_j.
- **itr** (*int*) – Current iteration number (=1 the first time mmasub is called).
- **xval** (*1-D array len(n)*) – Vector with the current values of the variables x_j.
- **xmin** (*1-D array len(n)*) – Vector with the lower bounds for the variables x_j.
- **xmax** (*1-D array len(n)*) – Vector with the upper bounds for the variables x_j.
- **xold1** (*1-D array len (n)*) – xval, one iteration ago when iter>1, zero othewise.
- **xold2** (*1-D array len (n)*) – xval, two iteration ago when iter>2, zero othewise.
- **f0val** (*float*) – The value of the objective function f_0 at xval.
- **df0dx** (*1-D array len(n)*) – Vector with the derivatives of the objective function f_0 with respect to the variables x_j, calculated at xval.
- **fval** (*1-D array len(m)*) – Vector with the values of the constraint functions f_i, calculated at xval.
- **dfdx** (*2-D array size(m x n)*) – (m x n)-matrix with the derivatives of the constraint functions f_i. with respect to the variables x_j, calculated at xval.
- **low** (*1-D array len(n)*) – Vector with the lower asymptotes from the previous iteration (provided that iter>1).
- **upp** (*1-D array len(n)*) – Vector with the upper asymptotes from the previous iteration (provided that iter>1).
- **a0** (*float*) – The constants a_0 in the term $a_0 z$.
- **a** (*1-D array len(m)*) – Vector with the constants $a_i 1 in the terms : math :$.
- **c** (*1-D array len(m)*) – Vector with the constants c_i in the terms $c_i * y_i$.
- **d** (*1-D array len(m)*) – Vector with the constants d_i in the terms $0.5 d_i (y_i)^2$.

Returns

- **xmma** (*1-D array len(n)*) – Column vector with the optimal values of the variables x_j in the current MMA subproblem.
- **low** (*1-D array len(n)*) – Column vector with the lower asymptotes, calculated and used in the current MMA subproblem.
- **upp** (*1-D array len(n)*) – Column vector with the upper asymptotes, calculated and used in the current MMA subproblem.
- *Version September 2007 (and a small change August 2008)*
- *Krister Svanberg <krille@math.kth.se>*
- *Department of Mathematics KTH, SE-10044 Stockholm, Sweden.*
- *Translated to python 3 by A.J.J. Lagerweij TU Delft June 2018*

sensitivityfilt(*x*, *rmin*, *dki*, *filt*)

Filters with a normalized convolution on the sensitivity with a radius of rmin if:

```
>>> filt=='sensitivity'
```

Parameters

- **x** (*2-D array size(nely, nelx)*) – Current density ditribution.
- **dki** (*2-D array size(nely, nelx*) – Stress intensity sensitivity to density changes.
- **rmin** (*float*) – Filter size.
- **filt** (*str*) – The filter type that is selected, either 'sensitivity' or 'density'.

Returns **dkif** – Filterd sensitivity distribution.

Return type 2-D array size(nely, nelx)

solvemma(*m*, *n*, *epsimin*, *low*, *upp*, *alfa*, *beta*, *p0*, *q0*, *P*, *Q*, *a0*, *a*, *b*, *c*, *d*)

This function solves the MMA subproblem with a primal-dual Newton method.

$$\min \sum_{j-1}^{n} \left(\frac{p_{0j}^{(k)}}{U_j^{(k)} - x_j} + \frac{q_{0j}^{(k)}}{x_j - L_j^{(k)}} \right) + a_0 z + \sum_{i=1}^{m} \left(c_i y_i + \frac{1}{2} d_i y_i^2 \right)$$

s.t.

$$\sum_{j-1}^{n} \left(\frac{p_{ij}^{(k)}}{U_j^{(k)} - x_j} + \frac{q_{ij}^{(k)}}{x_j - L_j^{(k)}} \right) - a_i z - y_i \leq b_i$$

$$\alpha_j \geq x_j \geq \beta_j$$

$$z \geq 0$$

Returns **x** – Column vector with the optimal values of the variables x_j in the current MMA subproblem.

Return type 1-D array len(n)

4.3.5 Plotting Module

Plotting the simulated TopOpt geometry with boundery conditions and loads. This version of the code is meant for mixed element types problems. Such as the stress intensity minimization.

Bram Lagerweij Aerospace Structures and Materials Department TU Delft 2018

class src_StressIntensity.plotting.**Plot**(*load*, *directory*, *title=None*)

This class contains functions that allows the visualisation of the TopOpt algorithem. It can print the density distribution, the boundary conditions and the forces.

Parameters

- **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.
- **directory** (*str*) – Relative directory that the results should be saved in.
- **title** (*str, optional*) – Title of the plot, optionaly.

nelx

Number of elements in x direction.

Type int

nely
Number of elements in y direction.

Type int

fig
An empty figure of size nelx/10 and nely/10 inch.

Type matplotlib.pyplot figure

ax
The axis system that belongs to fig.

Type matplotlib.pyplot axis

images
This list contains all density distributions that need to be plotted.

Type 1-D list with imshow objects

directory
Location where the results need to be saved.

Type str

add(*x*, *animated=False*)
Adding a plot of the density distribution to the figure.

Parameters

- **x** (*2-D array size(nely, nelx)*) – The density distribution.
- **animated** (*bool*) – An animated figure is genereted when history = True.

boundary(*load*)
Plotting the boundary conditions.

Parameters **load** (*object, child of the Loads class*) – The loadcase(s) considered for this optimisation problem.

find(*dof*)
This function returns the location, x,y of any degree of freedom by corresponding it with the edof array.

Parameters **dof** (*int*) – Degree of freedom number of unknown location.

Returns

- **x** (*float*) – x location of the dof.
- **y** (*float*) – y location of the dof.

loading(*load*)
Plotting the loading conditions.

Parameters **load** (*object, child of the Loads class*) – The loadcase(s) considered for this optimisation problem.

save(*filename*, *fps=10*)
Saving an plot in svg or mp4 format, depending on the length of the images list. The FasterFFMpegWriter is used when videos are generated. These videos are encoded with a hardware accelerated h264 codec with the .mp4 file format. Other codecs and encoders can be set within the function itself.

Parameters

- **filename** (*str*) – Name of the file, excluding the file exstension.

- **fps** (*int*) – Amount of frames per second if the plots are animations.

saveXYZ(*x*, *x_size*, *thickness=1*)

This function allows the export of the density distribution as a point cloud. This can be used to create .stl files in the following steps:

1. Open meshlab and 'import mesh' on all .xyz files.
2. Use 'Per Vertex Normal Fnction' on all point clouds.
   - bot with [nx, ny, nz] = [0, 0,-1]
   - top with [nx, ny, nz] = [0, 0, 1]
   - x- with [nx, ny, nz] = [-1, 0, 0]
   - x+ with [nx, ny, nz] = [1, 0, 0]
   - y- with [nx, ny, nz] = [0,-1, 0]
   - y+ with [nx, ny, nz] = [0, 1, 0]
3. Apply the 'Screened Poisson Surface Reconstruction' filter with the option of 'Merge all visible layers' as True

Parameters

- **x** (*2-D array*) – Density array.
- **x_size** (*float*) – X dimension of the mesh.
- **thickness** (*foat*) – Thickness of the mesh.

show()

Showing the plot in a window.

class src_StressIntensity.plotting.**FasterFFMpegWriter**(***kwargs*)

Bases: matplotlib.animation.FFMpegWriter

FFMpeg-pipe writer bypassing figure.savefig. To improof saving speed

classmethod bin_path()

Return the binary path to the commandline tool used by a specific subclass. This is a class method so that the tool can be looked for before making a particular MovieWriter subclass available.

cleanup()

Clean-up and collect the process used to write the movie file.

finish()

Finish any processing for writing the movie.

frame_size

A tuple `(width, height)` in pixels of a movie frame.

grab_frame(***savefig_kwargs*)

Grab the image information from the figure and save as a movie frame.

Doesn't use savefig to be faster: savefig_kwargs will be ignored.

classmethod isAvailable()

Check to see if a MovieWriter subclass is actually available.

saving(*fig*, *outfile*, *dpi*, **args*, ***kwargs*)

Context manager to facilitate writing the movie file.

`*args, **kw` are any parameters that should be passed to *setup*.

setup (*fig*, *outfile*, *dpi=None*)
Perform setup for writing the movie file.

Parameters

- **fig** (*~matplotlib.figure.Figure*) – The figure object that contains the information for frames
- **outfile** (*str*) – The filename of the resulting movie file
- **dpi** (*int, optional*) – The DPI (or resolution) for the file. This controls the size in pixels of the resulting movie file. Default is fig.dpi.

4.4 Fatigue Crack Growth Life Maximization

This fatigue crack growth life maximization designs a structure such that the most cycles are required for the crack to grow from a_0 (strating crack length) to a_{end} (final crack length). The theory behind the algorithm is explained in at *Fatigue Crack Growth Life Maximization* The crack path must be kown before running the optimization algorithms An example as how to use the optimization is shown in an example optimization example.py

- *Density Constraints*
- *Load Cases*
- *Finite Element Solvers*
- *Optimization Module*
- *Plotting Module*

4.4.1 Density Constraints

Constraints class used to specify the density constraints of the topology optimisation problem. It contains functions for minimum and maximum element density in the upcomming iteration and the magnitude of the volume constraint function itself of the current design. This version of the code is meant for the fatigue live maximization.

Bram Lagerweij Aerospace Structures and Materials Department TU Delft 2018

class src_FatigueLive.constraints.**DensityConstraint**(*nelx*, *nely*, *move*, *volume_frac*, *density_min=0.0*, *density_max=1.0*)
This object relates to the constraints used in this optimization. It can be used for the MMA updatescheme to derive what the limit is for all element densities at every itteration. The class itself is not changed by the itterations.

Parameters

- **nelx** (*int*) – Number of elements in x direction.
- **nely** (*int*) – Number of elements in y direction.
- **move** (*float*) – Maximum change in density of an element over 1 itteration.
- **volume_frac** (*float*) – Maximum volume that can be filled with material.
- **volume_derivative** (*2D array size(1, nelx*nely)*) – Sensityvity of the density constraint to the density in each element.

- **density_min** (*float (optional)*) – Minumum density, set at 0.0 if not specified.
- **density_max** (*float (optional)*) – Maximum density, set at 0.0 if not specified.

nelx
Number of elements in x direction.

Type int

nely
Number of elements in y direction.

Type int

move
Maximum change in density of an element over 1 itteration.

Type float

volume_frac
Maximum volume that can be filled with material.

Type float

volume_derivative
Sensityvity of the density constraint to the density in each element.

Type 2D array size(1, nelx*nely)

density_min
Minumum density, set at 0.0 if not specified.

Type float, optional

density_max
Maximum density, set at 0.0 if not specified.

Type float, optional

current_volconstrain(*x*)
Calculates the current magnitude of the volume constraint funcion:

$$V_{\text{constraint}} = \frac{\sum v_e X_e}{V_{\max}} - 1$$

Parameters **x** (*2D array size(nely, nelx)*) – Density distribution of this itteration.

Returns **curvol** – Curent value of the density constraint function.

Return type float

xmax(*x*)
This function calculates the maximum density value of all ellements of this itteration.

Parameters **x** (*2D array size(nely, nelx)*) – Density distribution of this itteration.

Returns **xmax** – Maximum density values of this itteration after updating.

Return type 2D array size(nely, nelx)

xmin(*x*)
This function calculates the minimum density value of all ellements of this itteration.

Parameters **x** (*2D array size(nely, nelx)*) – Density distribution of this itteration.

Returns **xmin** – Minimum density values of this itteration for the update scheme.

Return type 2D array size(nely, nelx)

4.4.2 Load Cases

This file contains the Load class that allows the generation of an object that contains geometric, mesh, loads and boundary conditions that belong to the load case. This version of the code is meant for the fatigue live maximization.

Bram Lagerweij Aerospace Structures and Materials Department TU Delft 2018

Parent Load Case

class src_FatigueLive.loads.**Load**(*nelx*, *nely*, *young*, *Emin*, *poisson*, *ext_stiff*, *hoe*)

Load parent class that contains the basic functions used in all load cases. This class and its children do contain information about the load case considered in the optimisation. The load case consists of the mesh, the loads, and the boundaries conditions. The class is constructed such that new load cases can be generated simply by adding a child and changing the function related to the geometry, loads and boundaries.

Parameters

- **nelx** (*int*) – Number of elements in x direction.
- **nely** (*int*) – Number of elements in y direction.
- **young** (*float*) – Young's modulus of the materials.
- **Emin** (*float*) – Artificial Young's modulus of the material to ensure a stable FEA. It is used in the SIMP based material model.
- **poisson** (*float*) – Poisson ration of the material.
- **ext_stiff** (*float*) – Extra stiffness to be added to global stiffness matrix. Due to interactions with mechanisms outside design domain.
- **hoe** (*dict*) – Dictionary with for every cracklength the x end y element locations that need to be enriched.

nelx

Number of elements in x direction.

Type int

nely

Number of elements in y direction.

Type int

dim

Amount of dimensions considered in the problem, set at 2.

Type int

edof

Dictionary containing list with all elements and their degree of freedom numbers for all crack_lengths, str(length) is the key.

Type dict

x_list

Dictionary with a 1D list that contains the x indices of all degrees of freedom for all crack lengths, str(length) is the key.

Type dict

y_list
Dictionary with a 1D list that contains the y indices of all degrees of freedom for all crack lengths, str(length) is the key.

Type dict

num_dofs
Amount of degrees of freedom.

Type int

young
Young's modulus of the materials.

Type float

Emin
Artificial Young's modulus of the material to ensure a stable FEA. It is used in the SIMP based material model.

Type float

poisson
Poisson ration of the material.

Type float

k_list
Dictionary containing a list for every crack length, these lists contain the element stiffness matrices of full density for all elements, str(length) is the key.

Type dict

kmin_list
Dictionary containing a list for every crack length, these lists contain the empty element stiffness matrices for all elements, str(length) is the key.

Type list len(nelx*nely)

ext_stiff
Extra stiffness to be added to global stiffness matrix. Due to interactions with mechanisms outside design domain.

Type float

alldofs()
Returns a list with all degrees of freedom.

Returns **all** – List with numbers from 0 to the maximum degree of freedom number.

Return type 1-D list

edofcalc(*hoe*)
Generates an array with the position of the nodes of each element in the global stiffness matrix. This takes the Higher Order Elements in account.

Parameters **hoe** (*list*) – A list containing the x and y location of the higher order elemens for this crack length.

Returns

- **edof** (*2-D list size(nelx*nely, # degrees of freedom per element)*) – The list with all elements and their degree of freedom numbers.

- **x_list** (*1-D array*) – The list with the x indices of all elements to be inserted into the global stiffness matrix.
- **y_list** (*1-D array*) – The list with the y indices of all elements to be inserted into the global stiffness matrix.
- **num_dofs** (*int*) – The amount of degrees of freedom.

fixdofs(*length_i*)

Returns a list with indices that are fixed by the boundary conditions.

Parameters **length_i** (*int*) – Length of the crack for the current mesh

Returns **fix** – List with all the numbers of fixed degrees of freedom. This list is empty in this parent class.

Return type 1-D list

force()

Returns an 1D array, the force vector of the loading condition.

Returns **f** – Empty force vector.

Return type 1-D column array length covering all degrees of freedom

freedofs(*length_i*)

Returns a list of arr indices that are not fixed

Parameters **length_i** (*int*) – Length of the crack for the current mesh

Returns **free** – List containing all elements of all dofs except those that appear in the freedos list.

Return type 1-D list

import_stiffness(*elementtype*, *E*, *nu*)

This function imports a matrix from a csv file that has variables to the material properties. The correct material properties are added.

Parameters

- **elementtype** (*str*) – Describes what .csv file should be used for the import.
- **E** (*float*) – Young's modulus of the material.
- **nu** (*float*) – Poisson's ratio of the material.

Returns **lk** – Element stiffness matrix

Return type array size(dofs, dofs)

kiloc()

The location of the stress intensity factor KI can be found at the second last index.

Returns **l** – Zeros except for the second last index.

Return type 1-D column array length covering all degrees of freedom

lk(*E*, *nu*, *hoe*)

Generates a list with all element stiffness matrices. It differentiates between the element types used.

Parameters

- **E** (*float*) – Young's modulus of the material.
- **nu** (*float*) – Poisson's ratio of the material.

Returns **k** – Returns a list with all local stiffness matrices.

Return type list len(nelx*nely)

node(*elx*, *ely*)
Calculates the topleft node number of the requested element. Does not toke Higher Order Elements in account.

Parameters

- **elx** (*int*) – X position of the considered element.
- **ely** (*int*) – Y position of the considered element.

Returns **topleft** – The node number of the top left node.

Return type int

nodes(*elx*, *ely*)
Calculates all node numbers of the requested element. Does not take Higher Order Elements in account.

Parameters

- **elx** (*int*) – X position of the considered element.
- **ely** (*int*) – Y position of the considered element.

Returns

- **n0** (*int*) – The node number of the bottom left node.
- **n1** (*int*) – The node number of the bottom right node.
- **n2** (*int*) – The node number of the top left node.
- **n3** (*int*) – The node number of the top right node.

passive()
Returns three lists containing the location and magnitude of fixed density values

Returns

- **elx** (*1-D list*) – X coordinates of all passive elements, empty for the parent class.
- **ely** (*1-D list*) – Y coordinates of all passive elements, empty for the parent class.
- **values** (*1-D list*) – Density values of all passive elements, empty for the parent class.
- **fix_ele** (*1-D list*) – List with all element numbers that are allowed to change.

reset_Kij()
Resets the global variable Kij. This is necessary as function import_stiffness will not clean up its local variables itself.

Child Load Cases

class src_FatigueLive.loads.**EdgeCrack**(*nelx*, *nely*, *crack_length*, *young*, *Emin*, *poisson*, *ext_stiff*)
Bases: *src_FatigueLive.loads.Load*

This child class of Load class represents the symmetric top half of an edge crack. The crack is positioned to the bottom left and propagates towards the right. Special elements are placed around the crack tip. The plate is subjected to a distributed tensile load ($\sigma = 1$) on the top.

For a perfectly flat plate analytical expressions for K_I are known.[2]

[2] Tada, H., Paris, P., & Irwin, G. (2000). "Part II 2.10-2.12 The Single Edge Notch Test Specimen", The stress analysis of cracks handbook (3rd ed.). New York: ASME Press, pp:52-54.

The stress intensity factors calculated can be be interperted in two ways:

1. Without scaling. This means that all elements have a size of 2 length units.
2. With scaling, comparison to reality should be based upon.

$$K^{\text{Real}} = K^{\text{FEA}}(\sigma = 1)\sigma^{\text{Real}}\sqrt{\frac{a^{\text{Real}}}{2a^{\text{FEA}}}}$$

where a^{FEA} is the cracklength in number of elements.

Parameters

- **nelx** (*int*) – Number of elements in x direction.
- **crack_length** (*array*) – An array containing all crack lengths considered.
- **young** (*float*) – Young's modulus of the materials.
- **Emin** (*float*) – Artificial Young's modulus of the material to ensure a stable FEA. It is used in the SIMP based material model.
- **poisson** (*float*) – Poisson ration of the material.
- **ext_stiff** (*float*) – Extra stiffness to be added to global stiffness matrix. Due to interactions with mechanisms outside design domain.

crack_length
Is the amount of elements that the crack is long.

Type int

hoe
List containing the x end y element locations that need to be enriched.

Type list len(2)

References

fixdofs (*length_i*)
The boundary conditions limit y-translation at the bottom of the design space (due to symetry) and x-translations at the top (due to the clamps)

Parameters **length_i** (*int*) – Length of the crack for the current mesh

Returns **fix** – List with all the numbers of fixed degrees of freedom.

Return type 1-D list

force (*length_i*)
The top of the design space is pulled upwards by 1MPa. This means that the nodal forces are 2 upwards, except for the top corners they have a load of 1 only.

Parameters **length_i** (*int*) – Length of the crack for the current mesh

Returns **f** – Force vector.

Return type 1-D column array length covering all degrees of freedom

passive ()
Returns three lists containing the location and magnitude of fixed density values. The elements around the crack tip are fixed at a density of one.

Returns

- **elx** (*1-D list*) – X coordinates of all passive elements, empty for the parent class.
- **ely** (*1-D list*) – Y coordinates of all passive elements, empty for the parent class.
- **values** (*1-D list*) – Density values of all passive elements, empty for the parent class.
- **fix_ele** (*1-D list*) – List with all element numbers that are allowed to change.

class src_FatigueLive.loads.**CompactTension**(*nelx*, *crack_length*, *young*, *Emin*, *poisson*, *ext_stiff*)

Bases: *src_FatigueLive.loads.Load*

This child class of Load class represents the symmetric top half of an compact tension specimen. The crack is positioned to the bottom left and propagates towards the right. Special elements are placed around the crack tip. The plate is subjected to upwards load of one. The design follows the ASTM standard.[3]

For a perfectly flat plate analytical expressions for K_I do exist.[4]

The stress intensity factors calculated can be be interpreted in two ways: 1. Without scaling. This means that all elements have a size of 2 length units. 2. With scaling, comparison to reality should be based upon.

$$K^{\text{Real}} = K^{\text{FEA}}(F=1)F^{\text{Real}}\sqrt{\frac{2W^{\text{FEA}}}{W^{\text{Real}}}}$$

where W^{FEA} is the width in number of elements.

Parameters

- **nelx** (*int*) – Number of elements in x direction.
- **crack_length** (*array*) – An array containing all crack lengths considered.
- **young** (*float*) – Young's modulus of the materials.
- **Emin** (*float*) – Artificial Young's modulus of the material to ensure a stable FEA. It is used in the SIMP based material model.
- **poisson** (*float*) – Poisson ration of the material.
- **ext_stiff** (*float*) – Extra stiffness to be added to global stiffness matrix. Due to interactions with mechanisms outside design domain.

nely
Number of y elements, this is now a function of nelx.

Type int

crack_length
Is for all cracks considered the crack_length.

Type array

References

fixdofs(*length_i*)
The bottom of the design space is fixed in y direction (due to symmetry around the x axis). While at the location that the load is introduced x translations are constraint.

[3] ASTM Standard E647-15e1, "Standard Test Method for Measurement of Fatigue Crack Growth Rates," ASTM Book of Standards, vol. 0.30.1, 2015.

[4] Tada, H., Paris, P., & Irwin, G. (2000). "Part II 2.19-2.21 The Compact Tension Test Specimen", The stress analysis of cracks handbook (3rd ed.). New York: ASME Press, pp:61-63.

Parameters **length_i** (*int*) – Length of the crack for the current mesh

Returns **fix** – List with all the numbers of fixed degrees of freedom.

Return type 1-D list

force(*length_i*)
The ASTM standard requires the force to be located approx. 1/5 of nelx and at 0.195 * nely from the top.

Parameters **length_i** (*int*) – Length of the crack for the current mesh

Returns **f** – Force vector

Return type 1-D column array length covering all degrees of freedom

passive()
Returns three lists containing the location and magnitude of fixed density values. The elements around the crack tip are fixed at a density of one.

Returns

- **elx** (*1-D list*) – X coordinates of all passive elements, empty for the parent class.
- **ely** (*1-D list*) – Y coordinates of all passive elements, empty for the parent class.
- **values** (*1-D list*) – Density values of all passive elements, empty for the parent class.
- **fix_ele** (*1-D list*) – List with all element numbers that are allowed to change.

4.4.3 Finite Element Solvers

Finite element solvers for the displacement from stiffness matrix, force and adjoint vector. This version of the code is meant for the fatigue crack growth maximization.

Bram Lagerweij Aerospace Structures and Materials Department TU Delft 2018

Parent Solver

class src_FatigueLive.fesolvers.**FESolver**(*verbose=False*)
This parent FEA class can only assemble the global stiffness matrix and exclude all fixed degrees of freedom from it. This function, gk_freedofs is used in all FEA solvers classes. The displace function is not implemented in this parrent class as it does not contain a solver for the linear problem.

Parameters **verbose** (*bool, optional*) – False if the FEA should not print updates

verbose
False if the FEA should not print updates.

Type bool

displace(*load*, *x*, *penal*, *length*)
FE solver based upon the sparse SciPy solver that uses umfpack.

Parameters

- **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.
- **x** (*2-D array size(nely, nelx)*) – Current density distribution.
- **penal** (*float*) – Material model penalisation (SIMP).
- **length** (*int*) – Length of the current crack conciderd.

Returns

- **u** (*1-D column array shape(max(edof), 1)*) – The displacement vector.
- **lamba** (*1-D column array shape(max(edof), 1)*) – Adjoint equation solution.

gk_freedofs(*load*, *x*, *penal*, *length*)

Generates the global stiffness matrix with deleted fixed degrees of freedom. It generates a list with stiffness values and their x and y indices in the global stiffness matrix. Some combination of x and y appear multiple times as the degree of freedom might apear in multiple elements of the FEA. The SciPy coo_matrix function adds them up at the background. At the location of the force introduction and displacement output an external stiffness is added due to stability reasons.

Parameters

- **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.
- **x** (*2-D array size(nely, nelx)*) – Current density distribution.
- **penal** (*float*) – Material model penalisation (SIMP).
- **length** (*int*) – Length of the current crack conciderd.

Returns **k** – Global stiffness matrix without fixed degrees of freedom.

Return type 2-D sparse csc matrix

Child Solvers

class src_FatigueLive.fesolvers.**CvxFEA**(*verbose=False*)

Bases: *src_FatigueLive.fesolvers.FESolver*

This parent FEA class can assemble the global stiffness matrix and solve the FE problem with a Supernodal Sparse Cholesky Factorization. It solves for both the equalibrium and adjoint problem.

verbose

False if the FEA should not print updates.

Type bool

displace(*load*, *x*, *penal*, *length*)

FE solver based upon a Supernodal Sparse Cholesky Factorization. It requires the instalation of the cvx module. It solves both the FEA equalibrium and adjoint problems.[1]

Parameters

- **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.
- **x** (*2-D array size(nely, nelx)*) – Current density distribution.
- **penal** (*float*) – Material model penalisation (SIMP).
- **length** (*int*) – Length of the current crack conciderd.

Returns

- **u** (*1-D column array shape(max(edof), 1)*) – The displacement vector.
- **lamba** (*1-D column array shape(max(edof), 1)*) – Adjoint equation solution.

[1] Y. Chen, T. A. Davis, W. W. Hager, S. Rajamanickam, "Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate", ACM Transactions on Mathematical Software, 35(3), 22:1-22:14, 2008.

References

class src_FatigueLive.fesolvers.**CGFEA**(*verbose=False*)
Bases: *src_FatigueLive.fesolvers.FESolver*

This parent FEA class can assemble the global stiffness matrix and solve the FE problem with a sparse solver based upon a preconditioned conjugate gradient solver. The preconditioning is based upon the inverse of the diagonal of the stiffness matrix.

Recomendations

- Make the tolerance change over the iterations, low accuracy is required for first itteration, more accuracy for the later ones.
- Add more advanced preconditioner.
- Add gpu accerelation.

verbose
False if the FEA should not print updates.

Type bool

ufree_old
Displacement field of previous iteration for every crack length, the keys are the related cracklengths.

Type dict

lambafree_old
Ajoint equation result of previos iteration for every crack length, the keys are the related cracklengths.

Type array len(freedofs)

displace(*load*, *x*, *penal*, *length*)
FE solver based upon the sparse SciPy solver that uses a preconditioned conjugate gradient solver, preconditioning is based upon the inverse of the diagonal of the stiffness matrix. Currently the relative tolerance is hardcoded as 1e-3.

Parameters

- **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.
- **x** (*2-D array size(nely, nelx)*) – Current density distribution.
- **penal** (*float*) – Material model penalisation (SIMP).
- **length** (*int*) – Length of the current crack conciderd.

Returns

- **u** (*1-D array len(max(edof)+1)*) – Displacement of all degrees of freedom
- **lamba** (*1-D column array shape(max(edof), 1)*) – Adjoint equation solution.

4.4.4 Optimization Module

Topology Optimization class that handles the itterations, objective functions, filters and update scheme. It requires to call upon a constraint, load case and FE solver classes. This version of the code is meant for the fatigue live maximization.

Bram Lagerweij Aerospace Structures and Materials Department TU Delft 2018

class src_FatigueLive.topopt.**Topopt**(*constraint*, *load*, *fesolver*, *weights*, *C*, *m*, *verbose=False*, *x0_loc=None*)

This is the optimisation object itself. It contains the initialisation of the density distribution.

Parameters

- **constraint** (*object of DensityConstraint class*) – The constraints for this optimization problem.
- **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.
- **fesolver** (*object, child of the CSCStiffnessMatrix class*) – The finite element solver.
- **weights** (*array length load cases*) – The weight given to each of the load cases.
- **C** (*float*) – Multiplication part of Paris-Erdogan law.
- **m** (*float*) – Power part of Paris-Erdogan law.
- **verbose** (*bool*) – Printing itteration results.
- **x0_loc** (*str*) – Set initial design with numpy '.npy' file location.

constraint
The constraints for this optimization problem.

Type object of DensityConstraint class

load
The loadcase(s) considerd for this optimisation problem.

Type object, child of the Loads class

fesolver
The finite element solver.

Type object, child of the CSCStiffnessMatrix class

verbose
Printing itteration results.

Type bool

itr
Number of iterations performed

Type int

weights
The weight given to each of the load cases.

Type array length load cases

C
Multiplication part of Paris-Erdogan law.

Type float

m
Power part of Paris-Erdogan law.

Type float

free_ele
All element nubers that ar allowed to change.

Type 1-D list

x
Array containing the current densities of every element.

Type 2-D array size(nely, nelx)

xold1
Flattend density distribution one iteration ago.

Type 1D array len(nelx*nely)

xold2
Flattend density distribution two iteration ago.

Type 1D array len(nelx*nely)

low
Column vector with the lower asymptotes, calculated and used in the MMA subproblem of the previous itteration.

Type 1D array len(nelx*nely)

upp
Column vector with the lower asymptotes, calculated and used in the MMA subproblem of the previous itteration.

Type 1D array len(nelx*nely)

densityfilt(*rmin*, *filt*)
Filters with a normalized convolution on the densities with a radius of rmin if:

```
>>> filt=='density'
```

The relusting geometry retains passive elements.

Parameters

- **rmin** (*float*) – Filter size.
- **filt** (*str*) – The filter type that is selected, either 'sensitivity' or 'density'.

Returns **xf** – Filterd density distribution.

Return type 2-D array size(nely, nelx)

iter(*penal*, *rmin*, *filt*)
This funcion performs one itteration of the topology optimisation problem. It

- loads the constraints,
- calculates the stiffness matrices,
- executes the density filter,
- executes the FEA solver,
- calls upon the displacment objective and its sensitivity calculation,
- executes the sensitivity filter,
- executes the MMA update scheme,
- and finaly updates density distribution (design).

Parameters

- **penal** (*float*) – Material model penalisation (SIMP).
- **rmin** (*float*) – Filter size.
- **filt** (*str*) – The filter type that is selected, either 'sensitivity' or 'density'.

Returns

- **change** (*float*) – Largest difference between the new and old density distribution.
- **volcon** (*float*) – Amount of volume of this itteration.
- **N** (*float*) – Fatigue live in cycles of the crack.
- **Obj** (*float*) – Objective in weigted cycles.

kicalc(*x*, *u*, *lamba*, *penal*, *length*)

This fuction calculates displacement of the objective node and its sensitivity to the densities.

Parameters

- **x** (*2-D array size(nely, nelx)*) – Possibly filterd density distribution.
- **u** (*1-D array size(max(edof), 1)*) – Displacement of all degrees of freedom.
- **lamba** (*2-D array size(max(edof), 1)*) –
- **ke** (*2-D array size(8, 8)*) – Element stiffness matrix with full density.
- **penal** (*float*) – Material model penalisation (SIMP).
- **length** (*int*) – Length of the crack conciderd.

Returns

- **ki** (*float*) – Displacement objective.
- **dki** (*2-D array size(nely, nelx)*) – Displacement objective sensitivity to density changes.

layout(*penal*, *rmin*, *delta*, *loopy*, *filt*, *history=False*)

Solves the topology optimisation problem by looping over the iter function.

Parameters

- **penal** (*float*) – Material model penalisation (SIMP).
- **rmin** (*float*) – Filter size.
- **delta** (*float*) – Convergence is roached when delta > change.
- **loopy** (*int*) – Amount of iteration allowed.
- **filt** (*str*) – The filter type that is selected, either 'sensitivity' or 'density'.
- **history** (*bool*) – Do the intermediate results need to be stored.

Returns

- **xf** (*array size(nely, nelx)*) – Density distribution resulting from the optimisation.
- **xf_history** (*list of arrays len(itterations size(nely, nelx))*) – List with the density distributions of all itterations, None when history != True.
- **ki** (*array*) – Stress intensity factor at each crack length increment.

mma(*m*, *n*, *itr*, *xval*, *xmin*, *xmax*, *xold1*, *xold2*, *f0val*, *df0dx*, *fval*, *dfdx*, *low*, *upp*, *a0*, *a*, *c*, *d*)

This function mmasub performs one MMA-iteration, aimed at solving the nonlinear programming problem:

$$\min f_0(x)$$

$$+a_0 z + \sum_{i=1}^{m} \left(c_i y_i + \frac{1}{2} d_i y_i^2 \right)$$

s.t.

$$f_i(x) - a_i z - y_i \leq 0 \qquad i \in \{1, 2, \ldots, m\}$$

$$x_{\min} \geq x_j \geq x_{\max} j \in \{1, 2, \ldots, n\}$$

$$y_i \leq 0 i \in \{1, 2, \ldots, m\}$$

$$z \geq 0$$

Parameters

- **m** (*int*) – The number of general constraints.
- **n** (*int*) – The number of variables x_j.
- **itr** (*int*) – Current iteration number (=1 the first time mmasub is called).
- **xval** (*1-D array len(n)*) – Vector with the current values of the variables x_j.
- **xmin** (*1-D array len(n)*) – Vector with the lower bounds for the variables x_j.
- **xmax** (*1-D array len(n)*) – Vector with the upper bounds for the variables x_j.
- **xold1** (*1-D array len (n)*) – xval, one iteration ago when iter>1, zero othewise.
- **xold2** (*1-D array len (n)*) – xval, two iteration ago when iter>2, zero othewise.
- **f0val** (*float*) – The value of the objective function f_0 at xval.
- **df0dx** (*1-D array len(n)*) – Vector with the derivatives of the objective function f_0 with respect to the variables x_j, calculated at xval.
- **fval** (*1-D array len(m)*) – Vector with the values of the constraint functions f_i, calculated at xval.
- **dfdx** (*2-D array size(m x n)*) – (m x n)-matrix with the derivatives of the constraint functions f_i. with respect to the variables x_j, calculated at xval.
- **low** (*1-D array len(n)*) – Vector with the lower asymptotes from the previous iteration (provided that iter>1).
- **upp** (*1-D array len(n)*) – Vector with the upper asymptotes from the previous iteration (provided that iter>1).
- **a0** (*float*) – The constants a_0 in the term $a_0 z$.
- **a** (*1-D array len(m)*) – Vector with the constants $a_i 1 in the terms : math :$.
- **c** (*1-D array len(m)*) – Vector with the constants c_i in the terms $c_i * y_i$.
- **d** (*1-D array len(m)*) – Vector with the constants d_i in the terms $0.5 d_i (y_i)^2$.

Returns

- **xmma** (*1-D array len(n)*) – Column vector with the optimal values of the variables x_j in the current MMA subproblem.
- **low** (*1-D array len(n)*) – Column vector with the lower asymptotes, calculated and used in the current MMA subproblem.
- **upp** (*1-D array len(n)*) – Column vector with the upper asymptotes, calculated and used in the current MMA subproblem.
- *Version September 2007 (and a small change August 2008)*
- *Krister Svanberg <krille@math.kth.se>*
- *Department of Mathematics KTH, SE-10044 Stockholm, Sweden.*
- *Translated to python 3 by A.J.J. Lagerweij TU Delft June 2018*

sensitivityfilt (*x*, *dOi*, *rmin*, *filt*)

Filters with a normalized convolution on the sensitivity with a radius of rmin if:

```
>>> filt=='sensitivity'
```

Parameters

- **x** (*2-D array size(nely, nelx)*) – Current density ditribution.
- **dOi** (*2-D array size(nely, nelx)*) – Objective sensitivity to density changes.
- **rmin** (*float*) – Filter size.
- **filt** (*str*) – The filter type that is selected, either 'sensitivity' or 'density'.

Returns **dOif** – Filterd sensitivity distribution.

Return type 2-D array size(nely, nelx)

solvemma (*m*, *n*, *epsimin*, *low*, *upp*, *alfa*, *beta*, *p0*, *q0*, *P*, *Q*, *a0*, *a*, *b*, *c*, *d*)

This function solves the MMA subproblem with a primal-dual Newton method:

$$\min \sum_{j-1}^{n} \left(\frac{p_{0j}^{(k)}}{U_j^{(k)} - x_j} + \frac{q_{0j}^{(k)}}{x_j - L_j^{(k)}} \right) + a_0 z + \sum_{i=1}^{m} \left(c_i y_i + \frac{1}{2} d_i y_i^2 \right)$$

s.t.

$$\sum_{j-1}^{n} \left(\frac{p_{ij}^{(k)}}{U_j^{(k)} - x_j} + \frac{q_{ij}^{(k)}}{x_j - L_j^{(k)}} \right) - a_i z - y_i \leq b_i$$

$$\alpha_j \geq x_j \geq \beta_j$$

$$z \geq 0$$

Returns **x** – Column vector with the optimal values of the variables x_j in the current MMA subproblem.

Return type 1-D array len(n)

4.4.5 Plotting Module

Plotting the simulated TopOpt geometry with boundery conditions and loads. This version of the code is meant for mixed element types problems for the fatigue live maximization.

Bram Lagerweij Aerospace Structures and Materials Department TU Delft 2018

class src_FatigueLive.plotting.**Plot**(*load*, *directory*, *title=None*)

This class contains functions that allows the visualisation of the TopOpt algorithem. It can print the density distribution, the boundary conditions and the forces.

Parameters

- **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.
- **directory** (*str*) – Relative directory that the results should be saved in.
- **title** (*str, optional*) – Title of the plot, optionaly.

nelx

Number of elements in x direction.

Type int

nely

Number of elements in y direction.

Type int

fig

An empty figure of size nelx/10 and nely/10 inch.

Type matplotlib.pyplot figure

ax

The axis system that belongs to fig.

Type matplotlib.pyplot axis

images

This list contains all density distributions that need to be plotted.

Type 1-D list with imshow objects

directory

Location where the results need to be saved.

Type str

add(*x*, *animated=False*)

Adding a plot of the density distribution to the figure.

Parameters

- **x** (*2-D array size(nely, nelx)*) – The density distribution.
- **animated** (*bool*) – An animated figure is genereted when history = True.

boundary(*load*)

Plotting the boundary conditions.

Parameters **load** (*object, child of the Loads class*) – The loadcase(s) considerd for this optimisation problem.

find(*dof*)
This function returns the location, x,y of any degree of freedom by corresponding it with the edof array.

Parameters **dof** (*int*) – Degree of freedom number of unknown location.

Returns

- **x** (*float*) – x location of the dof.
- **y** (*float*) – y location of the dof.

loading(*load*)
Plotting the loading conditions.

Parameters **load** (*object, child of the Loads class*) – The loadcase(s) considered for this optimisation problem.

save(*filename*, *fps=10*)
Saving an plot in svg or mp4 format, depending on the length of the images list. The FasterFFMpegWriter is used when videos are generated. These videos are encoded with a hardware accelerated h264 codec with the .mp4 file format. Other codecs and encoders can be set within the function itself.

Parameters

- **filename** (*str*) – Name of the file, excluding the file exstension.
- **fps** (*int*) – Amount of frames per second if the plots are animations.

saveXYZ(*x*, *x_size*, *thickness=1*)
This function allows the export of the density distribution as a point cloud. This can be used to create .stl files in the following steps:

1. Open meshlab and 'import mesh' on all .xyz files.
2. Use 'Per Vertex Normal Fnction' on all point clouds.
   - bot with [nx, ny, nz] = [0, 0,-1]
   - top with [nx, ny, nz] = [0, 0, 1]
   - x- with [nx, ny, nz] = [-1, 0, 0]
   - x+ with [nx, ny, nz] = [1, 0, 0]
   - y- with [nx, ny, nz] = [0,-1, 0]
   - y+ with [nx, ny, nz] = [0, 1, 0]
3. Apply the 'Screened Poisson Surface Reconstruction' filter with the option of 'Merge all visible layers' as True

Parameters

- **x** (*2-D array*) – Density array.
- **x_size** (*float*) – X dimension of the mesh.
- **thickness** (*foat*) – Thickness of the mesh.

show()
Showing the plot in a window.

class src_FatigueLive.plotting.**FasterFFMpegWriter**(***kwargs*)
Bases: matplotlib.animation.FFMpegWriter

FFMpeg-pipe writer bypassing figure.savefig. To improof saving speed

classmethod bin_path()
Return the binary path to the commandline tool used by a specific subclass. This is a class method so that the tool can be looked for before making a particular MovieWriter subclass available.

cleanup()
Clean-up and collect the process used to write the movie file.

finish()
Finish any processing for writing the movie.

frame_size
A tuple `(width, height)` in pixels of a movie frame.

grab_frame(***savefig_kwargs*)
Grab the image information from the figure and save as a movie frame.

Doesn't use savefig to be faster: savefig_kwargs will be ignored.

classmethod isAvailable()
Check to see if a MovieWriter subclass is actually available.

saving(*fig*, *outfile*, *dpi*, **args*, ***kwargs*)
Context manager to facilitate writing the movie file.

`*args, **kw` are any parameters that should be passed to *setup*.

setup(*fig*, *outfile*, *dpi=None*)
Perform setup for writing the movie file.

Parameters

- **fig** (*~matplotlib.figure.Figure*) – The figure object that contains the information for frames
- **outfile** (*str*) – The filename of the resulting movie file
- **dpi** (*int, optional*) – The DPI (or resolution) for the file. This controls the size in pixels of the resulting movie file. Default is fig.dpi.

CHAPTER 5

Indices and Tables

- genindex
- modindex
- search

Python Module Index

S

src_Actuator.constraints, 52
src_Actuator.fesolvers, 57
src_Actuator.loads, 54
src_Actuator.plotting, 65
src_Actuator.topopt, 60
src_Compliance.constraints, 37
src_Compliance.fesolvers, 43
src_Compliance.loads, 39
src_Compliance.plotting, 50
src_Compliance.topopt, 46
src_FatigueLive.constraints, 86
src_FatigueLive.fesolvers, 94
src_FatigueLive.loads, 88
src_FatigueLive.plotting, 102
src_FatigueLive.topopt, 96
src_StressIntensity.constraints, 67
src_StressIntensity.fesolvers, 76
src_StressIntensity.loads, 68
src_StressIntensity.plotting, 83
src_StressIntensity.topopt, 79

Index

A

add() (*src_Actuator.plotting.Plot method*), 65
add() (*src_Compliance.plotting.Plot method*), 51
add() (*src_FatigueLive.plotting.Plot method*), 102
add() (*src_StressIntensity.plotting.Plot method*), 84
alldofs() (*src_Actuator.loads.Load method*), 55
alldofs() (*src_Compliance.loads.Load method*), 40
alldofs() (*src_FatigueLive.loads.Load method*), 89
alldofs() (*src_StressIntensity.loads.Load method*), 70
ax (*src_Actuator.plotting.Plot attribute*), 65
ax (*src_Compliance.plotting.Plot attribute*), 51
ax (*src_FatigueLive.plotting.Plot attribute*), 102
ax (*src_StressIntensity.plotting.Plot attribute*), 84

B

Beam (*class in src_Compliance.loads*), 42
BiAxial (*class in src_Compliance.loads*), 43
bin_path() (*src_Actuator.plotting.FasterFFMpegWriter class method*), 66
bin_path() (*src_Compliance.plotting.FasterFFMpegWriter class method*), 51
bin_path() (*src_FatigueLive.plotting.FasterFFMpegWriter class method*), 103
bin_path() (*src_StressIntensity.plotting.FasterFFMpegWriter class method*), 85
boundary() (*src_Actuator.plotting.Plot method*), 65
boundary() (*src_Compliance.plotting.Plot method*), 51
boundary() (*src_FatigueLive.plotting.Plot method*), 102
boundary() (*src_StressIntensity.plotting.Plot method*), 84

C

C (*src_FatigueLive.topopt.Topopt attribute*), 97
Canti (*class in src_Compliance.loads*), 42
CGFEA (*class in src_Actuator.fesolvers*), 59
CGFEA (*class in src_Compliance.fesolvers*), 45
CGFEA (*class in src_FatigueLive.fesolvers*), 96
CGFEA (*class in src_StressIntensity.fesolvers*), 78
cleanup() (*src_Actuator.plotting.FasterFFMpegWriter method*), 66
cleanup() (*src_Compliance.plotting.FasterFFMpegWriter method*), 51
cleanup() (*src_FatigueLive.plotting.FasterFFMpegWriter method*), 104
cleanup() (*src_StressIntensity.plotting.FasterFFMpegWriter method*), 85
comp() (*src_Compliance.topopt.Topopt method*), 47
CompactTension (*class in src_FatigueLive.loads*), 93
CompactTension (*class in src_StressIntensity.loads*), 74
constraint (*src_Actuator.topopt.Topopt attribute*), 60
constraint (*src_Compliance.topopt.Topopt attribute*), 46
constraint (*src_FatigueLive.topopt.Topopt attribute*), 97
constraint (*src_StressIntensity.topopt.Topopt attribute*), 79
crack_length (*src_FatigueLive.loads.CompactTension attribute*), 93
crack_length (*src_FatigueLive.loads.EdgeCrack attribute*), 92
crack_length (*src_StressIntensity.loads.CompactTension attribute*), 75
crack_length (*src_StressIntensity.loads.DoubleEdgeCrack attribute*), 74
crack_length (*src_StressIntensity.loads.EdgeCrack attribute*), 72
current_volconstrain() (*src_Actuator.constraints.DensityConstraint method*), 53
current_volconstrain() (*src_Compliance.constraints.DensityConstraint method*), 38
current_volconstrain() (*src_FatigueLive.constraints.DensityConstraint method*), 87

current_volconstrain() (*src_StressIntensity.constraints.DensityConstraint method*), 68
CvxFEA (*class in src_Actuator.fesolvers*), 58
CvxFEA (*class in src_Compliance.fesolvers*), 44
CvxFEA (*class in src_FatigueLive.fesolvers*), 95
CvxFEA (*class in src_StressIntensity.fesolvers*), 77

D

density_max (*src_Actuator.constraints.DensityConstraint attribute*), 53
density_max (*src_Compliance.constraints.DensityConstraint attribute*), 38
density_max (*src_FatigueLive.constraints.DensityConstraint attribute*), 87
density_max (*src_StressIntensity.constraints.DensityConstraint attribute*), 68
density_min (*src_Actuator.constraints.DensityConstraint attribute*), 53
density_min (*src_Compliance.constraints.DensityConstraint attribute*), 38
density_min (*src_FatigueLive.constraints.DensityConstraint attribute*), 87
density_min (*src_StressIntensity.constraints.DensityConstraint attribute*), 68
DensityConstraint (*class in src_Actuator.constraints*), 53
DensityConstraint (*class in src_Compliance.constraints*), 37
DensityConstraint (*class in src_FatigueLive.constraints*), 86
DensityConstraint (*class in src_StressIntensity.constraints*), 67
densityfilt() (*src_Actuator.topopt.Topopt method*), 61
densityfilt() (*src_Compliance.topopt.Topopt method*), 47
densityfilt() (*src_FatigueLive.topopt.Topopt method*), 98
densityfilt() (*src_StressIntensity.topopt.Topopt method*), 80
dim (*src_Actuator.loads.Load attribute*), 55
dim (*src_Compliance.loads.Load attribute*), 40
dim (*src_FatigueLive.loads.Load attribute*), 88
dim (*src_StressIntensity.loads.Load attribute*), 69
directory (*src_FatigueLive.plotting.Plot attribute*), 102
directory (*src_StressIntensity.plotting.Plot attribute*), 84
disp() (*src_Actuator.topopt.Topopt method*), 61
displace() (*src_Actuator.fesolvers.CGFEA method*), 59
displace() (*src_Actuator.fesolvers.CvxFEA method*), 58
displace() (*src_Actuator.fesolvers.FESolver method*), 57
displace() (*src_Compliance.fesolvers.CGFEA method*), 45
displace() (*src_Compliance.fesolvers.CvxFEA method*), 44
displace() (*src_Compliance.fesolvers.FESolver method*), 44
displace() (*src_FatigueLive.fesolvers.CGFEA method*), 96
displace() (*src_FatigueLive.fesolvers.CvxFEA method*), 95
displace() (*src_FatigueLive.fesolvers.FESolver method*), 94
displace() (*src_StressIntensity.fesolvers.CGFEA method*), 78
displace() (*src_StressIntensity.fesolvers.CvxFEA method*), 77
displace() (*src_StressIntensity.fesolvers.FESolver method*), 76
displaceloc() (*src_Actuator.loads.Inverter method*), 57
displaceloc() (*src_Actuator.loads.Load method*), 55
DoubleEdgeCrack (*class in src_StressIntensity.loads*), 73

E

EdgeCrack (*class in src_FatigueLive.loads*), 91
EdgeCrack (*class in src_StressIntensity.loads*), 72
edof (*src_FatigueLive.loads.Load attribute*), 88
edof (*src_StressIntensity.loads.Load attribute*), 69
edof() (*src_Actuator.loads.Load method*), 55
edof() (*src_Compliance.loads.Load method*), 40
edofcalc() (*src_FatigueLive.loads.Load method*), 89
edofcalc() (*src_StressIntensity.loads.Load method*), 70
Emin (*src_Actuator.loads.Load attribute*), 55
Emin (*src_Compliance.loads.Load attribute*), 39
Emin (*src_FatigueLive.loads.Load attribute*), 89
Emin (*src_StressIntensity.loads.Load attribute*), 69
ext_stiff (*src_Actuator.loads.Load attribute*), 55
ext_stiff (*src_FatigueLive.loads.Load attribute*), 89
ext_stiff (*src_StressIntensity.loads.Load attribute*), 70

F

FasterFFMpegWriter (*class in src_Actuator.plotting*), 66
FasterFFMpegWriter (*class in src_Compliance.plotting*), 51
FasterFFMpegWriter (*class in src_FatigueLive.plotting*), 103

FasterFFMpegWriter (*class in src_StressIntensity.plotting*), 85
FESolver (*class in src_Actuator.fesolvers*), 57
FESolver (*class in src_Compliance.fesolvers*), 43
FESolver (*class in src_FatigueLive.fesolvers*), 94
FESolver (*class in src_StressIntensity.fesolvers*), 76
fesolver (*src_Actuator.topopt.Topopt attribute*), 60
fesolver (*src_Compliance.topopt.Topopt attribute*), 46
fesolver (*src_FatigueLive.topopt.Topopt attribute*), 97
fesolver (*src_StressIntensity.topopt.Topopt attribute*), 79
fig (*src_Actuator.plotting.Plot attribute*), 65
fig (*src_Compliance.plotting.Plot attribute*), 51
fig (*src_FatigueLive.plotting.Plot attribute*), 102
fig (*src_StressIntensity.plotting.Plot attribute*), 84
find() (*src_FatigueLive.plotting.Plot method*), 102
find() (*src_StressIntensity.plotting.Plot method*), 84
finish() (*src_Actuator.plotting.FasterFFMpegWriter method*), 66
finish() (*src_Compliance.plotting.FasterFFMpegWriter method*), 52
finish() (*src_FatigueLive.plotting.FasterFFMpegWriter method*), 104
finish() (*src_StressIntensity.plotting.FasterFFMpegWriter method*), 85
fixdofs() (*src_Actuator.loads.Inverter method*), 57
fixdofs() (*src_Actuator.loads.Load method*), 55
fixdofs() (*src_Compliance.loads.Beam method*), 42
fixdofs() (*src_Compliance.loads.BiAxial method*), 43
fixdofs() (*src_Compliance.loads.Canti method*), 42
fixdofs() (*src_Compliance.loads.HalfBeam method*), 41
fixdofs() (*src_Compliance.loads.Load method*), 40
fixdofs() (*src_Compliance.loads.Michell method*), 42
fixdofs() (*src_FatigueLive.loads.CompactTension method*), 93
fixdofs() (*src_FatigueLive.loads.EdgeCrack method*), 92
fixdofs() (*src_FatigueLive.loads.Load method*), 90
fixdofs() (*src_StressIntensity.loads.CompactTension method*), 75
fixdofs() (*src_StressIntensity.loads.DoubleEdgeCrack method*), 74
fixdofs() (*src_StressIntensity.loads.EdgeCrack method*), 73
fixdofs() (*src_StressIntensity.loads.Load method*), 70
force() (*src_Actuator.loads.Inverter method*), 57
force() (*src_Actuator.loads.Load method*), 55
force() (*src_Compliance.loads.Beam method*), 42
force() (*src_Compliance.loads.BiAxial method*), 43
force() (*src_Compliance.loads.Canti method*), 42
force() (*src_Compliance.loads.HalfBeam method*), 41
force() (*src_Compliance.loads.Load method*), 40
force() (*src_Compliance.loads.Michell method*), 43
force() (*src_FatigueLive.loads.CompactTension method*), 94
force() (*src_FatigueLive.loads.EdgeCrack method*), 92
force() (*src_FatigueLive.loads.Load method*), 90
force() (*src_StressIntensity.loads.CompactTension method*), 76
force() (*src_StressIntensity.loads.DoubleEdgeCrack method*), 74
force() (*src_StressIntensity.loads.EdgeCrack method*), 73
force() (*src_StressIntensity.loads.Load method*), 70
frame_size (*src_Actuator.plotting.FasterFFMpegWriter attribute*), 66
frame_size (*src_Compliance.plotting.FasterFFMpegWriter attribute*), 52
frame_size (*src_FatigueLive.plotting.FasterFFMpegWriter attribute*), 104
frame_size (*src_StressIntensity.plotting.FasterFFMpegWriter attribute*), 85
free_ele (*src_FatigueLive.topopt.Topopt attribute*), 97
free_ele (*src_StressIntensity.topopt.Topopt attribute*), 79
freedofs() (*src_Actuator.loads.Load method*), 56
freedofs() (*src_Compliance.loads.Load method*), 40
freedofs() (*src_FatigueLive.loads.Load method*), 90
freedofs() (*src_StressIntensity.loads.Load method*), 70

G

gk_freedofs() (*src_Actuator.fesolvers.FESolver method*), 58
gk_freedofs() (*src_Compliance.fesolvers.FESolver method*), 44
gk_freedofs() (*src_FatigueLive.fesolvers.FESolver method*), 95
gk_freedofs() (*src_StressIntensity.fesolvers.FESolver method*), 77
grab_frame() (*src_Actuator.plotting.FasterFFMpegWriter method*), 66
grab_frame() (*src_Compliance.plotting.FasterFFMpegWriter method*), 52
grab_frame() (*src_FatigueLive.plotting.FasterFFMpegWriter method*), 104
grab_frame() (*src_StressIntensity.plotting.FasterFFMpegWriter method*), 85

H

HalfBeam (*class in src_Compliance.loads*), 41
hoe (*src_FatigueLive.loads.EdgeCrack attribute*), 92

hoe (*src_StressIntensity.loads.CompactTension attribute*), 75
hoe_type (*src_StressIntensity.loads.CompactTension attribute*), 75
hoe_type (*src_StressIntensity.loads.DoubleEdgeCrack attribute*), 74
hoe_type (*src_StressIntensity.loads.EdgeCrack attribute*), 73

I

images (*src_Actuator.plotting.Plot attribute*), 65
images (*src_Compliance.plotting.Plot attribute*), 51
images (*src_FatigueLive.plotting.Plot attribute*), 102
images (*src_StressIntensity.plotting.Plot attribute*), 84
import_stiffness() (*src_FatigueLive.loads.Load method*), 90
import_stiffness() (*src_StressIntensity.loads.Load method*), 70
Inverter (*class in src_Actuator.loads*), 57
isAvailable() (*src_Actuator.plotting.FasterFFMpegWriter class method*), 66
isAvailable() (*src_Compliance.plotting.FasterFFMpegWriter class method*), 52
isAvailable() (*src_FatigueLive.plotting.FasterFFMpegWriter class method*), 104
isAvailable() (*src_StressIntensity.plotting.FasterFFMpegWriter class method*), 85
iter() (*src_Actuator.topopt.Topopt method*), 61
iter() (*src_Compliance.topopt.Topopt method*), 47
iter() (*src_FatigueLive.topopt.Topopt method*), 98
iter() (*src_StressIntensity.topopt.Topopt method*), 80
itr (*src_Actuator.topopt.Topopt attribute*), 60
itr (*src_Compliance.topopt.Topopt attribute*), 46
itr (*src_FatigueLive.topopt.Topopt attribute*), 97
itr (*src_StressIntensity.topopt.Topopt attribute*), 79

K

k_list (*src_FatigueLive.loads.Load attribute*), 89
k_list (*src_StressIntensity.loads.Load attribute*), 69
ki() (*src_StressIntensity.topopt.Topopt method*), 81
kicalc() (*src_FatigueLive.topopt.Topopt method*), 99
kiloc() (*src_FatigueLive.loads.Load method*), 90
kiloc() (*src_StressIntensity.loads.Load method*), 71
kmin_list (*src_FatigueLive.loads.Load attribute*), 89
kmin_list (*src_StressIntensity.loads.Load attribute*), 70

L

lambafree_old (*src_Actuator.fesolvers.CGFEA attribute*), 59
lambafree_old (*src_FatigueLive.fesolvers.CGFEA attribute*), 96
lambafree_old (*src_StressIntensity.fesolvers.CGFEA attribute*), 78
layout() (*src_Actuator.topopt.Topopt method*), 62
layout() (*src_Compliance.topopt.Topopt method*), 48
layout() (*src_FatigueLive.topopt.Topopt method*), 99
layout() (*src_StressIntensity.topopt.Topopt method*), 81
lk() (*src_Actuator.loads.Load method*), 56
lk() (*src_Compliance.loads.Load method*), 40
lk() (*src_FatigueLive.loads.Load method*), 90
lk() (*src_StressIntensity.loads.Load method*), 71
Load (*class in src_Actuator.loads*), 54
Load (*class in src_Compliance.loads*), 39
Load (*class in src_FatigueLive.loads*), 88
Load (*class in src_StressIntensity.loads*), 68
load (*src_Actuator.topopt.Topopt attribute*), 60
load (*src_Compliance.topopt.Topopt attribute*), 46
load (*src_FatigueLive.topopt.Topopt attribute*), 97
load (*src_StressIntensity.topopt.Topopt attribute*), 79
loading() (*src_Actuator.plotting.Plot method*), 65
loading() (*src_Compliance.plotting.Plot method*), 51
loading() (*src_FatigueLive.plotting.Plot method*), 103
loading() (*src_StressIntensity.plotting.Plot method*), 84
low (*src_Actuator.topopt.Topopt attribute*), 61
low (*src_Compliance.topopt.Topopt attribute*), 47
low (*src_FatigueLive.topopt.Topopt attribute*), 98
low (*src_StressIntensity.topopt.Topopt attribute*), 80

M

m (*src_FatigueLive.topopt.Topopt attribute*), 97
Michell (*class in src_Compliance.loads*), 42
mma() (*src_Actuator.topopt.Topopt method*), 62
mma() (*src_Compliance.topopt.Topopt method*), 48
mma() (*src_FatigueLive.topopt.Topopt method*), 99
mma() (*src_StressIntensity.topopt.Topopt method*), 81
move (*src_Actuator.constraints.DensityConstraint attribute*), 53
move (*src_Compliance.constraints.DensityConstraint attribute*), 38
move (*src_FatigueLive.constraints.DensityConstraint attribute*), 87
move (*src_StressIntensity.constraints.DensityConstraint attribute*), 67

N

nelx (*src_Actuator.constraints.DensityConstraint attribute*), 53
nelx (*src_Actuator.loads.Load attribute*), 54
nelx (*src_Actuator.plotting.Plot attribute*), 65
nelx (*src_Compliance.constraints.DensityConstraint attribute*), 38
nelx (*src_Compliance.loads.Load attribute*), 39

nelx (*src_Compliance.plotting.Plot attribute*), 50
nelx (*src_FatigueLive.constraints.DensityConstraint attribute*), 87
nelx (*src_FatigueLive.loads.Load attribute*), 88
nelx (*src_FatigueLive.plotting.Plot attribute*), 102
nelx (*src_StressIntensity.constraints.DensityConstraint attribute*), 67
nelx (*src_StressIntensity.loads.Load attribute*), 69
nelx (*src_StressIntensity.plotting.Plot attribute*), 83
nely (*src_Actuator.constraints.DensityConstraint attribute*), 53
nely (*src_Actuator.loads.Load attribute*), 54
nely (*src_Actuator.plotting.Plot attribute*), 65
nely (*src_Compliance.constraints.DensityConstraint attribute*), 38
nely (*src_Compliance.loads.Load attribute*), 39
nely (*src_Compliance.plotting.Plot attribute*), 50
nely (*src_FatigueLive.constraints.DensityConstraint attribute*), 87
nely (*src_FatigueLive.loads.CompactTension attribute*), 93
nely (*src_FatigueLive.loads.Load attribute*), 88
nely (*src_FatigueLive.plotting.Plot attribute*), 102
nely (*src_StressIntensity.constraints.DensityConstraint attribute*), 67
nely (*src_StressIntensity.loads.CompactTension attribute*), 75
nely (*src_StressIntensity.loads.DoubleEdgeCrack attribute*), 74
nely (*src_StressIntensity.loads.Load attribute*), 69
nely (*src_StressIntensity.plotting.Plot attribute*), 84
node() (*src_Actuator.loads.Load method*), 56
node() (*src_Compliance.loads.Load method*), 40
node() (*src_FatigueLive.loads.Load method*), 91
node() (*src_StressIntensity.loads.Load method*), 71
nodes() (*src_Actuator.loads.Load method*), 56
nodes() (*src_Compliance.loads.Load method*), 41
nodes() (*src_FatigueLive.loads.Load method*), 91
nodes() (*src_StressIntensity.loads.Load method*), 71
num_dofs (*src_FatigueLive.loads.Load attribute*), 89
num_dofs (*src_StressIntensity.loads.Load attribute*), 69

P

passive() (*src_Actuator.loads.Load method*), 56
passive() (*src_Compliance.loads.BiAxial method*), 43
passive() (*src_Compliance.loads.Load method*), 41
passive() (*src_FatigueLive.loads.CompactTension method*), 94
passive() (*src_FatigueLive.loads.EdgeCrack method*), 92
passive() (*src_FatigueLive.loads.Load method*), 91
passive() (*src_StressIntensity.loads.CompactTension method*), 76
passive() (*src_StressIntensity.loads.DoubleEdgeCrack method*), 74
passive() (*src_StressIntensity.loads.EdgeCrack method*), 73
passive() (*src_StressIntensity.loads.Load method*), 71
Plot (*class in src_Actuator.plotting*), 65
Plot (*class in src_Compliance.plotting*), 50
Plot (*class in src_FatigueLive.plotting*), 102
Plot (*class in src_StressIntensity.plotting*), 83
poisson (*src_Actuator.loads.Load attribute*), 55
poisson (*src_Compliance.loads.Load attribute*), 39
poisson (*src_FatigueLive.loads.Load attribute*), 89
poisson (*src_StressIntensity.loads.Load attribute*), 69

R

reset_Kij() (*src_FatigueLive.loads.Load method*), 91
reset_Kij() (*src_StressIntensity.loads.Load method*), 72

S

save() (*src_Actuator.plotting.Plot method*), 65
save() (*src_Compliance.plotting.Plot method*), 51
save() (*src_FatigueLive.plotting.Plot method*), 103
save() (*src_StressIntensity.plotting.Plot method*), 84
saveXYZ() (*src_FatigueLive.plotting.Plot method*), 103
saveXYZ() (*src_StressIntensity.plotting.Plot method*), 85
saving() (*src_Actuator.plotting.FasterFFMpegWriter method*), 66
saving() (*src_Compliance.plotting.FasterFFMpegWriter method*), 52
saving() (*src_FatigueLive.plotting.FasterFFMpegWriter method*), 104
saving() (*src_StressIntensity.plotting.FasterFFMpegWriter method*), 85
sensitivityfilt() (*src_Actuator.topopt.Topopt method*), 64
sensitivityfilt() (*src_Compliance.topopt.Topopt method*), 49
sensitivityfilt() (*src_FatigueLive.topopt.Topopt method*), 101
sensitivityfilt() (*src_StressIntensity.topopt.Topopt method*), 82
setup() (*src_Actuator.plotting.FasterFFMpegWriter method*), 66
setup() (*src_Compliance.plotting.FasterFFMpegWriter method*), 52
setup() (*src_FatigueLive.plotting.FasterFFMpegWriter method*), 104

setup() (*src_StressIntensity.plotting.FasterFFMpegWriter method*), 85
show() (*src_Actuator.plotting.Plot method*), 66
show() (*src_Compliance.plotting.Plot method*), 51
show() (*src_FatigueLive.plotting.Plot method*), 103
show() (*src_StressIntensity.plotting.Plot method*), 85
solvemma() (*src_Actuator.topopt.Topopt method*), 64
solvemma() (*src_Compliance.topopt.Topopt method*), 50
solvemma() (*src_FatigueLive.topopt.Topopt method*), 101
solvemma() (*src_StressIntensity.topopt.Topopt method*), 83
src_Actuator.constraints (*module*), 52
src_Actuator.fesolvers (*module*), 57
src_Actuator.loads (*module*), 54
src_Actuator.plotting (*module*), 65
src_Actuator.topopt (*module*), 60
src_Compliance.constraints (*module*), 37
src_Compliance.fesolvers (*module*), 43
src_Compliance.loads (*module*), 39
src_Compliance.plotting (*module*), 50
src_Compliance.topopt (*module*), 46
src_FatigueLive.constraints (*module*), 86
src_FatigueLive.fesolvers (*module*), 94
src_FatigueLive.loads (*module*), 88
src_FatigueLive.plotting (*module*), 102
src_FatigueLive.topopt (*module*), 96
src_StressIntensity.constraints (*module*), 67
src_StressIntensity.fesolvers (*module*), 76
src_StressIntensity.loads (*module*), 68
src_StressIntensity.plotting (*module*), 83
src_StressIntensity.topopt (*module*), 79

T

Topopt (*class in src_Actuator.topopt*), 60
Topopt (*class in src_Compliance.topopt*), 46
Topopt (*class in src_FatigueLive.topopt*), 96
Topopt (*class in src_StressIntensity.topopt*), 79

U

ufree_old (*src_Actuator.fesolvers.CGFEA attribute*), 59
ufree_old (*src_Compliance.fesolvers.CGFEA attribute*), 45
ufree_old (*src_FatigueLive.fesolvers.CGFEA attribute*), 96
ufree_old (*src_StressIntensity.fesolvers.CGFEA attribute*), 78
upp (*src_Actuator.topopt.Topopt attribute*), 61
upp (*src_Compliance.topopt.Topopt attribute*), 47
upp (*src_FatigueLive.topopt.Topopt attribute*), 98
upp (*src_StressIntensity.topopt.Topopt attribute*), 80

V

verbose (*src_Actuator.fesolvers.CGFEA attribute*), 59
verbose (*src_Actuator.fesolvers.CvxFEA attribute*), 58
verbose (*src_Actuator.fesolvers.FESolver attribute*), 57
verbose (*src_Actuator.topopt.Topopt attribute*), 60
verbose (*src_Compliance.fesolvers.CGFEA attribute*), 45
verbose (*src_Compliance.fesolvers.CvxFEA attribute*), 44
verbose (*src_Compliance.fesolvers.FESolver attribute*), 44
verbose (*src_Compliance.topopt.Topopt attribute*), 46
verbose (*src_FatigueLive.fesolvers.CGFEA attribute*), 96
verbose (*src_FatigueLive.fesolvers.CvxFEA attribute*), 95
verbose (*src_FatigueLive.fesolvers.FESolver attribute*), 94
verbose (*src_FatigueLive.topopt.Topopt attribute*), 97
verbose (*src_StressIntensity.fesolvers.CGFEA attribute*), 78
verbose (*src_StressIntensity.fesolvers.CvxFEA attribute*), 77
verbose (*src_StressIntensity.fesolvers.FESolver attribute*), 76
verbose (*src_StressIntensity.topopt.Topopt attribute*), 79
volume_derivative (*src_Actuator.constraints.DensityConstraint attribute*), 53
volume_derivative (*src_Compliance.constraints.DensityConstraint attribute*), 38
volume_derivative (*src_FatigueLive.constraints.DensityConstraint attribute*), 87
volume_derivative (*src_StressIntensity.constraints.DensityConstraint attribute*), 67
volume_frac (*src_Actuator.constraints.DensityConstraint attribute*), 53
volume_frac (*src_Compliance.constraints.DensityConstraint attribute*), 38
volume_frac (*src_FatigueLive.constraints.DensityConstraint attribute*), 87
volume_frac (*src_StressIntensity.constraints.DensityConstraint attribute*), 67

W

weights (*src_FatigueLive.topopt.Topopt attribute*), 97

X

x (*src_Actuator.topopt.Topopt attribute*), 60

x (*src_Compliance.topopt.Topopt attribute*), 46
x (*src_FatigueLive.topopt.Topopt attribute*), 98
x (*src_StressIntensity.topopt.Topopt attribute*), 79
x_list (*src_FatigueLive.loads.Load attribute*), 88
x_list (*src_StressIntensity.loads.Load attribute*), 69
xmax() (*src_Actuator.constraints.DensityConstraint method*), 54
xmax() (*src_Compliance.constraints.DensityConstraint method*), 38
xmax() (*src_FatigueLive.constraints.DensityConstraint method*), 87
xmax() (*src_StressIntensity.constraints.DensityConstraint method*), 68
xmin() (*src_Actuator.constraints.DensityConstraint method*), 54
xmin() (*src_Compliance.constraints.DensityConstraint method*), 39
xmin() (*src_FatigueLive.constraints.DensityConstraint method*), 87
xmin() (*src_StressIntensity.constraints.DensityConstraint method*), 68
xold1 (*src_Actuator.topopt.Topopt attribute*), 60
xold1 (*src_Compliance.topopt.Topopt attribute*), 46
xold1 (*src_FatigueLive.topopt.Topopt attribute*), 98
xold1 (*src_StressIntensity.topopt.Topopt attribute*), 79
xold2 (*src_Actuator.topopt.Topopt attribute*), 61
xold2 (*src_Compliance.topopt.Topopt attribute*), 47
xold2 (*src_FatigueLive.topopt.Topopt attribute*), 98
xold2 (*src_StressIntensity.topopt.Topopt attribute*), 80

Y

y_list (*src_FatigueLive.loads.Load attribute*), 88
y_list (*src_StressIntensity.loads.Load attribute*), 69
young (*src_Actuator.loads.Load attribute*), 54
young (*src_Compliance.loads.Load attribute*), 39
young (*src_FatigueLive.loads.Load attribute*), 89
young (*src_StressIntensity.loads.Load attribute*), 69