TopOpt in Python Documentation

Release 0.0.9

A.J.J. Lagerweij

May 06, 2019

About

CHAPTER 1

Introduction

CHAPTER 2

About

2.1 About this Project

2.2 Background in Topology Optimization

2.3 Different Objectives

2.4 MIT License

CHAPTER 3

Introduction and Examples

CHAPTER 4

Docstrings

4.1 Compliance Minimization

intro 1

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.

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)

4.1.2 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

Parrent 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

Parrent 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.

displace(*load*, *x*, *ke*, *kmin*, *penal*)
FE solver based upon a Supernodal Sparse Cholesky Factorization. It requires the instalation of the cvx module.

See: 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 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))

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.

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.
- **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

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 thnp.array([1,2])at 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 in the terms a_i*z.
- **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.**FasterFFMpegWriter**(***kwargs*)
Bases: matplotlib.animation.FFMpegWriter

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

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.

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

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*) – 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.

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, optional*) – Amount of frames per second if the plots are animations.

show()
Showing the plot in a window.

4.2 Maximum Compliance Actuators

4.3 Stress Intensity Factor Minimization

4.4 Fatigue Crack Growth Life Maximization

CHAPTER 5

Indices and Tables

- genindex
- modindex
- search

Python Module Index

S

Index