

Department of Precision and Microsystems Engineering

USING TOPOLOGY OPTIMIZATION FOR ACTUATOR PLACEMENT WITHIN MOTION SYSTEMS

Stefan Broxterman

Report no : EM 2017.38
Coach : dr.ir. M. Langelaar
Professor : dr.ir. M. Langelaar
Specialisation : Engineering Mechanics
Type of report : MSc thesis
Date : August 30, 2017

Using Topology Optimization for Actuator Placement within Motion Systems

MASTER OF SCIENCE THESIS

For the degree of Master of Science in Mechanical Engineering at Delft
University of Technology

Stefan Broxterman

August 30, 2017

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of
Technology



Copyright © 2017 by Stefan Broxterman

All rights reserved.

*PME - the Ultimate in
Mechanical Engineering*

“A person who never made a mistake never tried anything new.”

— *A. Einstein*

Abstract

Topology optimization is a strong approach for generating optimal designs which cannot be obtained using conventional optimization methods. Improving structural characteristics by changing the internal topology of a design domain has been fascinating scientists and engineers for years. Topology optimization can be described as a distribution of a given amount of material in a specified design domain, which is subjected to certain loading and boundary conditions. This domain can then be optimized to minimize specified objectives, for example compliance. For static problems, topology optimization is extensively used. The distribution of material, void and solid regions, can be used to solve several problems within the mechanical domain. However, this method of optimization is also used to optimize structures with respect to their resonant dynamics.

Using topology optimization preliminaries, the research first focuses on the design of supports. By taking a bridge as example, it is explained why design of supports can be helpful. When supports are not prescribed, the process of design of supports can be used to determine where these supports should be placed. The combination of topology optimization and design of supports can also be very helpful in compliant mechanisms.

Design of supports is then exploited to design of actuator placement. This new approach of optimizing focuses on design problems, where the placement of force is not prescribed. For a given material in a static domain, the optimal actuator lay-out is determined. This optimal placement of actuators can contribute to a better objective. A minimal force constraint is implemented, to avoid trivial solutions.

Topology optimization is included and combined with design of actuator placement. The simultaneous optimization process of topology and load placement is shown and explained. It is shown that topology and load placement are influencing each other which leads to even better objective results, while respecting given constraints.

Finally, a wafer stage is considered as case study. By implementing a harmonic force, dynamics are introduced. Some basic phenomena of the dynamics are introduced and explained. Then, design of actuator placement is used to ensure that certain mode shapes are not excited whereas other are. It is shown that a larger actuator design domain typically results in better dynamic performance.

After the process of design of actuators, topology optimization is included here. Topology optimization will be used to determine the most ideal placement of the actuators to comply with the requested (minimal) frequency response and lead to a better objective.

This placement of actuators can be used within certain motion systems, especially where the placement of actuators is not pre-defined by manufacturing. However, the results of the theoretical model could trigger users to reconsider their current manufacturing, in order to apply the improved actuator placements to improve their current dynamic performance.

Preface

This thesis is the result of my Master of Science graduation project. The opportunity was given by the department of Precision and Microsystems Engineering at the Delft University of Technology.

During this thesis I have been supervised by Gijs van der Veen and Matthijs Langelaar. I would like to thank my supervisors for their advice, feedback and support during my research.

Unfortunately Gijs left the University during my thesis research. Matthijs replaced him very well. Both are very supportive and their enthusiasm and knowledge of both gentlemen inspired me a lot.

I also would like to thank my friends, family and my fellow students for their support, love and patience during this project.

Stefan Broxterman
August 2017

Table of Contents

Abstract	i
Preface	iii
Nomenclature	xvii
1 Introduction	1
1.1 Background	1
1.2 Research goals	3
1.3 Approach	3
1.4 Outline	4
I Topology Optimization Preliminaries	5
2 Topology Optimization	7
2.1 Topology optimization formulation	7
2.1.1 Compliance example	8
2.1.2 Mesh-refinement	8
2.1.3 Volume fraction	10
2.2 Solution methods	10
2.2.1 SIMP method	11
2.2.2 BESO method	12
2.2.3 Sensitivity analysis	13
2.2.4 Filtering	15
2.3 Applications	16
2.3.1 Statics	16
2.3.2 Dynamics	17

2.3.3	Other domains	17
2.4	Design of supports	17
2.4.1	Optimizing supports	18
2.5	Conclusions	19
3	Topology Optimization for Engineers	21
3.1	Solution method: MMA	21
3.1.1	OC versus MMA	22
3.2	Advanced applications	23
3.2.1	Restrictive regions	23
3.2.2	Multiple load cases	24
3.2.3	Self-weight implementation	25
3.2.4	Continuation method	26
3.2.5	Different filter techniques	27
3.3	Turning 2D into 3D	29
3.3.1	Gray-scale filter	29
3.4	Compliant mechanisms	31
3.4.1	Inverter and amplifier	31
3.4.2	Micro-gripper	33
3.5	Conclusions	34
II	Topology Optimization Extensions: Design of Supports and Loads	35
4	Design of Supports	37
4.1	Support design formulation	37
4.2	The bridge	40
4.2.1	The optimal bridge	40
4.3	Advanced bridge designs	42
4.3.1	Hanging bridge	43
4.3.2	Train tunnel	44
4.3.3	Integration of layout design in supports	45
4.4	Design of compliant mechanisms	46
4.4.1	The optimal amplifier	46
4.4.2	The optimal micro-gripper	47
4.5	Application of support design	47
4.5.1	Actuator locations	48
4.6	Conclusions	49

5	Design of Actuator Placement	51
5.1	Actuator design formulation	51
5.1.1	Sensitivity selection	52
5.1.2	Arching continuation approach	53
5.1.3	Finite difference method	54
5.2	Simple cantilever beam	54
5.2.1	Minimal displacement	55
5.3	Advanced applications	56
5.3.1	Maximal displacement	56
5.3.2	Triple fixed beam	57
5.3.3	Minimal area displacement	57
5.4	Topology optimization for actuator placement	58
5.4.1	Displacement consideration	60
5.4.2	Compliance constraint	60
5.4.3	Objective refinement	62
5.5	Application of actuator placement	64
5.6	Conclusions	64
III	Dynamic Topology Optimization	65
6	Case Study: Wafer Stage	67
6.1	Case introduction	67
6.2	Dynamics	68
6.2.1	Single force actuator	69
6.2.2	Eigenmodes	70
6.2.3	Frequency response	71
6.2.4	Dynamic mode dependency	72
6.2.5	Double actuator	73
6.2.6	Distributed actuators	74
6.3	Design of actuators	75
6.3.1	Design of negative forces	77
6.3.2	Design of force at multiple sides	77
6.4	Topology optimization for dynamic performance	79
6.4.1	Topology optimization for fixed force	79
6.4.2	Topology optimization for double actuator	80
6.4.3	Side force and topology optimization	81
6.4.4	Negative forces and topology optimization	82
6.5	Topology optimization for actuator placement	84
6.5.1	Improving gray regions	85
6.5.2	Changing conditions	85
6.6	3D extrusion	87
6.7	Conclusions	87

IV	Closure	89
7	Conclusions and Recommendations	91
7.1	Conclusions	91
7.2	Recommendations	92
A	Appendix	95
A.1	Computational setup	96
A.2	Numerical results	96
A.2.1	Chapter 2 results	96
A.2.2	Chapter 3 results	98
A.2.3	Chapter 4 results	101
A.2.4	Chapter 5 results	103
A.2.5	Chapter 6 results	104
A.3	Convergence graph	107
A.3.1	Chapter 3 graphs	109
A.3.2	Chapter 4 graphs	111
A.4	Computational graph	112
A.5	Arching continuation	113
A.6	Deformed geometry	114
A.6.1	Deformed triple fixed beam	114
A.6.2	Deformed cantilever beam	115
A.6.3	Deformed cantilever beam with density dependency	116
A.6.4	Deformed cantilever beam topology	117
A.6.5	Deformed cantilever beam topology with density dependency	118
A.7	Mode contribution	119
A.7.1	Mode contribution tables	119
A.7.2	Mode contribution graphics	125
A.7.3	Mode contribution progress plots	127
A.8	Additional stage examples	128
A.8.1	Optimizing at eigenfrequency	128
A.8.2	Overfitting design of actuators	129
A.8.3	Overfitting design of actuators with topology optimization	131
A.8.4	Changing conditions representations	132
A.8.5	3D Extrusion	133
A.9	Flowcharts	134

B Matlab Codes	135
B.1 Basic.m	136
B.2 ADVANCED.m	142
B.3 BASIC.m	148
B.4 BASIC 3D.m	156
B.5 ADVANCED 3D.m	169
B.6 BASIC COMPLIANT MECHANISMS.m	176
B.7 Design of Supports.m	186
B.8 ADVANCED DOS.m	201
B.9 Design of Actuator Placement.m	208
B.10 Design of Actuator Placement Including Topology Optimization.m	223
C Add-in Codes	241
C.1 Basic MMA Add-in.m	242
C.2 Basic Restrictions Add-in.m	244
C.3 Basic Load Cases Add-in.m	246
C.4 Basic Self-weight Add-in.m	247
C.5 Basic Continuity Add-in.m	248
C.6 Basic Filters Add-in.m	249
C.7 3D Add-in.m	251
C.8 Complaint Mechanisms Add-in.m	262
C.9 Design of Supports Add-in.m	266
C.10 Design of Actuator Placement Add-in.m	276
C.11 Topology Add-in.m	288
D Supplementary Codes	295
D.1 Mmasub.m	296
D.2 Subsolv.m	299
D.3 Arrowz.m	304
Bibliography	307

List of Figures

1-1	Three categories of structural optimization	2
2-1	Compliance example	9
2-2	Mesh example	10
2-3	Volume example	11
2-4	Penalty example	12
2-5	BESO compliance example	13
2-6	Filter example	15
2-7	Design of lightweight city bus	16
2-8	Bridge example	18
3-1	Compare solution methods	23
3-2	Passive example	24
3-3	Active example	25
3-4	Multiple load cases	26
3-5	Self-weight example	27
3-6	Different filters example	28
3-7	3D example	29
3-8	3D lateral example	30
3-9	Micro-gripper hand tool	31
3-10	Compliant mechanism problem	32
3-11	Inverter and amplifier examples	32
3-12	Micro-gripper design problem	33
3-13	Micro-gripper example 1	33
3-14	Micro-gripper example 2	34

4-1	Bridge example	38
4-2	Support springs	39
4-3	Simple bridge example	40
4-4	The optimal bridge	41
4-5	The optimal bridge example 1	42
4-6	The optimal bridge example 2	43
4-7	Hanging bridge	44
4-8	Train tunnel	45
4-9	Micro-gripper design problem	46
4-10	Optimal amplifier example	47
4-11	Optimal micro-gripper example	48
5-1	Actuator placement example	55
5-2	Simple cantilever minimal displacement	56
5-3	Advanced applications actuator placement	57
5-4	Triple fixed beam	58
5-5	Minimal displacement including topology	60
5-6	Optimal cantilever beam with topology optimization	61
5-7	Minimal displacement including topology	62
5-8	Optimal cantilever beam with topology optimization including density dependency	63
6-1	Wafer stage	68
6-2	Design domain single force case	69
6-3	Mode shape solid body	70
6-4	Mode shape solid body	71
6-5	Bode plot	72
6-6	Design domain two forces case	74
6-7	Design domain distributed force case	74
6-8	Design of actuators case 1	75
6-9	Design of actuators case 2	75
6-10	Design of actuators case 3	77
6-11	Design of actuators case 4	78
6-12	Design of actuators case 5	79
6-13	Topology optimization for static force	80
6-14	Topology optimization and actuator placement for two force design	81
6-15	Optimal actuator and topology case 2	82
6-16	Optimal actuator and topology case 3	83
6-17	Optimal actuator and topology case 5	84
6-18	Optimal actuator and topology case 4	85
6-19	Improving gray regions	86

6-20 3D extrusion	87
A-1 Convergence standard compliance example	107
A-2 Convergence mesh refinement example	107
A-3 Convergence volume example	108
A-4 Convergence penalty example	108
A-5 Convergence filter example	109
A-6 Convergence OC vs MMA	109
A-7 Different filters example	110
A-8 3D mesh refinement example	110
A-9 Optimal bridge example	111
A-10 Hanging bridge example	111
A-11 Computational comparison	112
A-12 Arching continuation	113
A-13 Triple fixed beam deformed geometry	114
A-14 Triple fixed beam deformed geometry for area	114
A-15 Cantilever beam deformed geometry	115
A-16 Cantilever beam deformed geometry with density dependency	116
A-17 Cantilever beam deformed topology geometry	117
A-18 Cantilever beam deformed topology geometry	118
A-19 Mode contribution single force case	125
A-20 Mode contribution two forces case	126
A-21 Mode contribution distributed force case	126
A-22 Mode contribution progress plot	127
A-23 Design of eigenmode case	129
A-24 Overfitting design case	130
A-25 Overfitting design case with topology	131
A-26 Different volume restrictions	132
A-27 Different actuating frequencies	132
A-28 3D extrusion	133
A-29 Flowchart of optimization methods	134

List of Tables

6-1	Mode contribution of single force case	73
6-2	Results overview	88
A-1	Computer resources	96
A-2	Standard compliance example	96
A-3	Mesh refinement example	97
A-4	Volume fraction example	97
A-5	Penalty example	97
A-6	Filter example	97
A-7	OC vs MMA	98
A-8	Passive and active examples	98
A-9	Multiple load cases	98
A-10	Self-weight example	99
A-11	Different filters example	99
A-12	3D mesh refinement example	99
A-13	Complaint mechanism example	100
A-14	Optimal bridge	101
A-15	Optimal bridge example 1	101
A-16	Optimal bridge example 2	101
A-17	Hanging bridge	102
A-18	Train tunnel example	102

A-19 Optimal compliant mechanisms	102
A-20 Minimal compliance beam	103
A-21 Simple cantilever beam	103
A-22 Triple fixed beam	103
A-23 Cantilever beam with topology optimization	103
A-24 Dynamic solid beam	104
A-25 Design of dynamic actuator placement	104
A-26 Design of dynamic actuator placement	104
A-27 Dynamic actuator placement and topology	105
A-28 Dynamic actuator placement and topology	105
A-29 Dynamic actuator placement and topology	105
A-30 Dynamic actuator additional cases	106
A-31 Dynamic actuator additional cases	106
A-32 Dynamic actuator overfitting cases	106
A-33 Mode contribution of single force case	119
A-34 Mode contribution of two forces case	120
A-35 Mode contribution distributed force case	120
A-36 Mode contribution of case 1	121
A-37 Mode contribution of case 2	121
A-38 Mode contribution of case 3	122
A-39 Mode contribution of case 4	122
A-40 Mode contribution of case 5	123
A-41 Mode contribution of overfitting case	123
A-42 Mode contribution of eigenmode case	124

Nomenclature

General meaning of often used symbols, unless mentioned otherwise in the context.

α	Penalty slope	h	Perturbation value
γ	Support factor	i	Node number
β	Heavi-side filter parameter	\mathbf{K}	Stiffness matrix
η_i	Mode influence	\mathbf{K}_e	Element stiffness matrix
λ	Lagrange multiplier	\mathbf{K}_s	Spring stiffness matrix
ρ	Density	\mathbf{K}_x	Stiffness density displacement
$\tilde{\rho}_e$	Filtered element density	$K_{s,0}$	Maximum stiffness
$\bar{\rho}_e$	Projected element density	\mathbf{L}	Selection vector
ρ_0	Self-weight density	m	Mass
ρ_n	Node density array	\mathbf{M}	Global mass matrix
ϕ_i	Eigenvector	\mathbf{M}_e	Elemental mass matrix
ω	Actuating frequency	N	Number of elements
ω_i	Eigenfrequency	N_i	Number of nodes
A	Support area	p	Penalty
c	Compliance	q	Spring penalty
d_{in}	Input displacement	r	Filter radius
d_{out}	Output displacement	\mathbf{u}	General displacement array
E	Young's modulus	$\ddot{\mathbf{u}}$	Acceleration array
E_p	Penalized Young's modulus	$\tilde{\mathbf{u}}$	Adjoint displacement array
e	Element number	u_a	Displacement of selected area
f	Function value	\mathbf{u}_x	Node density displacement
f'	Differentiated function value	V	Total volume
\mathbf{f}_{sw}	Self-weight force array	v	Volume
\mathbf{f}	Force array	W	Weight factor
\mathbf{f}_p	Penalized force array	z	Support design variable
G_d	Displacement gain		

Chapter 1

Introduction

This report is a representation of my Master of Science thesis project. The aim of the research is to investigate the use of topology optimization for the optimal placement of actuators, to use within motion systems. At first, a background for this thesis is given in section 1.1, followed by the main research goals in section 1.2. The methodology is depicted next in section 1.3. This chapter is concluded by a quick outline of this thesis project in section 1.4.

1.1 Background

Nowadays, engineers are faced with structures of increasing complexity. These structures are getting smaller, lighter and more detailed. This tendency should not conflict the objective of the structure. A car, for example, would benefit from less weight for fuel cost reduction. The chassis however, should remain stiff enough to counteract deformations and provide safety for the driver. In the high-tech industry, and the equipment used there, like a wafer stage or robots, complexity is increasing. Also, the design space is getting smaller, especially in the semiconductor industry. The structure should, however, be stiff enough to not conflict its reliability. A very promising approach for these type of problems is the use of topology optimization.

Topology optimization is the process, which determines the optimal material placement within a certain design domain, in order to obtain the best possible structural performance. The process is widely used within the engineering domain, since the use of a homogenization method in topology optimization (Bendsoe and Kikuchi, 1988). Topology optimization gives the connectivity, shape and topology of elements in the design domain. The topology of elements can be described as a distribution of void and solid regions within that design domain.

Topology optimization is the newest technique in the field of structural optimization. Structural optimization is divided in three main categories, the choice of optimization is mainly based on the design variables. Three examples of this categories are depicted in Figure 1-1. Structural optimization can be used in discrete and continuum structures, depending on the

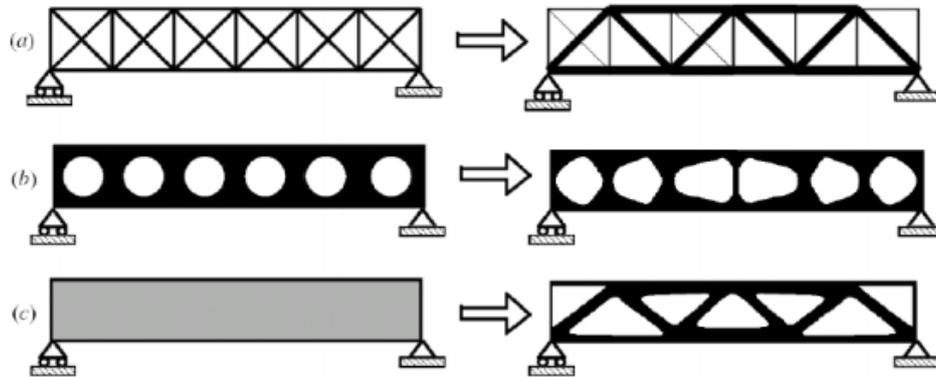


Figure 1-1: Three categories of structural optimization. a) Sizing optimization of a truss structure, b) shape optimization and c) topology optimization. The initial problems are shown at the left hand and the optimal solutions are shown at the right. (Bendsoe and Sigmund, 2003).

design properties and domain.

As can be seen in Figure 1-1a, sizing optimization is here used for a truss structure. The optimization objective is to maximize the vertical stiffness by changing the cross-sectional area of each truss element. This cross-sectional area can thus be considered as a design variable. The case depicted in Figure 1-1b is a shape optimization. Changing the geometry of the holes can provide a higher stiffness. The area and number of holes remains fixed, which is called a constraint. However, the shapes of the holes which are the design variables can be changed. In most cases, structural optimization problems are not fixed at only sizing or only shape problems. A mixture of the categories is needed, in order to achieve the most optimal result. As can be seen in Figure 1-1c a continuum structure is optimized to achieve maximum stiffness for a given amount of material. This is a typical topology optimization problem. The term topology is derived from the Greek word *topos* (τόπος), which is landscape or place. The 2D-landscape is changed, so the topology of the material is changed (Sigmund, 2000).

Current topology optimization is focusing on structural design, but there are other aspects designers have to make decisions for, like boundary conditions and load placement. These type of design problems emerge for example in the field of high-precision positioning systems, like a wafer stage. All these aspects are important in this case. In current research, there is a lot missing in this particular field. There is no research available on actuator placement, nor a combined with dynamics.

1.2 Research goals

The previous section depicts plenty of opportunity for research. The need for smaller, lighter and more complex structures can be labeled as the main reason for this research project. The first goal of this research project is to investigate the way to include the placement of supports within static topology optimization. The next step is to investigate the usage of design of supports for a variety of example problems.

The second goal of this thesis research project is to investigate the principles of actuator placement and find a way to include the best placement of these actuators. If this actuator placement is correct, topology optimization can also be included, to achieve even further improvements.

All previous investigations were in a static setting. A next step is to extend the work to a dynamic setting. This can be formulated as the third research goal of this thesis. By using a harmonic excitation the optimal actuator placement can be found. If this actuator placement can be combined with dynamic excitations, topology optimization should be implemented also in here. An interesting research goal is to optimize the actuation of a wafer stage by using topology optimization and actuator placement.

1.3 Approach

This thesis will make use of the background of topology optimization. This basics are used to achieve the research goals. In order to get familiar with topology optimization, an investigation on the process called topology optimization is done. The possibilities of this process are investigated and a user-friendly code using Matlab¹ is made, for further usage of my own and other research. The implementation process in Matlab of several features discussed in this thesis, can be found at the back of this thesis, by means of the used Matlab codes. These codes are made user-friendly to make further research more accessible.

This research focuses mostly on two-dimensional examples, where discretization sizes are held the same along the chapters, as much as possible. For consistency, the produced output pictures are shown in the same manner along this report.

The main approach of this thesis can be reflected by the partition of three different parts. First, general topology optimization preliminaries are explained. Using this gained knowledge, extensions are made in the field of boundary conditions and load placement. At last, dynamics are implemented. With these fundamentals established, a case study is used to combine all this gained knowledge.

¹Matrix Laboratory. A numerical computing environment, using a proprietary programming language.

1.4 Outline

In this section an outline of this research project can be found. This project is divided in four parts.

Before starting this thesis an introduction to topology optimization is given in chapter 2. For readers familiar with topology optimization this chapter is optional. Next, the level of topology optimization is increased to investigate several options of topology optimization in chapter 3.

The second part of this thesis consists multiple topology optimization extensions. Design of supports, including topology optimization can be found in chapter 4. Next, in chapter 5 we translate the design of supports in the design of actuator placement.

In chapter 6, dynamics are introduced. This gives a new aspect to the method. Therefore, a case study is described in the form of a wafer stage. By the implementation of dynamics, this chapter can be seen as a complete summation and practical application of the gained knowledge. The thesis project is ended by a conclusion and recommendations for future work in chapter 7.

Appendix A contains detailed specifications of the hardware that is used. Since time is important within the topology optimization process, all the calculation results can also be found in this chapter. In Appendix B the used Matlab codes can be found. Also, for each implementation a simple and user-friendly add-in can be found in Appendix C. Using this add-in codes everyone can simply upgrade the basic code up to a desired code. Supplementary codes can be found in Appendix D, followed by a list of references.

Part I

Topology Optimization Preliminaries

Topology Optimization

This chapter is dedicated to obtaining general knowledge and options using topology optimization. First it is explained what the formulation of topology optimization looks like in section 2.1. Next, it is discussed how this formulation can be solved using solution methods in 2.2. Practical applications of topology optimization can be found in section 2.3. Design of supports is slightly touched in section 2.4. Section 2.5 contains an overview of topology optimization and concludes this chapter.

2.1 Topology optimization formulation

Starting at a certain configuration the topology optimization process will optimize the structure to an objective, by varying the design variables. The structure is then optimized by creating several void and solid regions within the design domain.

A typical optimization problem is to set up a minimum compliance design. This design aims to optimize a simple mechanical structure to have a maximum stiffness, or minimum compliance ($c = k^{-1}$). Of course, the maximum stiffness will be achieved when the structure is thus a solid structure. However, for several reasons, it could be interesting to reduce the weight of the structure, while preserving its high stiffness properties. Reducing weight could reduce the material costs, save fuel costs (for example in aerodynamics), and could change intended dynamical behavior (for example in machinery).

To set up such an optimization problem, the intended volume is defined as a boundary condition. The mechanical equations should hold during this optimization problem, which can also be labeled as a boundary condition.

So now let's set up a basic topology optimization. Here we want the structure to be as stiff as possible, while it is subjected to (s.t.) a certain reduced weight value.

$$\begin{aligned} \max \quad & \text{Stiffness} \\ \text{s.t.} \quad & m \leq m_{max} \end{aligned} \tag{2-1}$$

Now assume linear elasticity, and replace stiffness by compliance to give a standard topology compliance optimization problem (Langelaar, 2012).

$$\begin{aligned}
 &\text{Equilibrium: } \mathbf{K}\mathbf{u} = \mathbf{f} \\
 &\text{Compliance: } c = \mathbf{f}^T \mathbf{u} \\
 &\min_{\text{design}} \quad \mathbf{f}^T \mathbf{u} \\
 &\text{s.t.} \quad \mathbf{K}\mathbf{u} = \mathbf{f} \\
 &\quad \quad m \leq m_{max}
 \end{aligned} \tag{2-2}$$

In this equation (2-2) the objective is to minimize the compliance. This objective can be achieved by varying specified design parameters, in this case there are no parameters specified, so the design parameters are free to choose. However, in most cases this does not apply. Due to external circumstances or internal properties in most cases it is necessary to specify the design variables. In many cases of topology optimization, the topology can be seen as a free variable, so the density is a design variable.

2.1.1 Compliance example

In this section a very simple compliance problem will be used, just to show how the topology optimization process is actually working. *A picture is worth a thousand words* give rise to this section. The process which leads to the optimum results will be explained further in this chapter, but for now let's focus just on the evolution of the topology optimization solution. In this particular example, the main objective is to maximize stiffness (minimize compliance), while the maximum mass of the structure is enforced. Using the formulation as depicted in (2-2), this topology optimization problem can be defined. The design parameter is the material's density.

$$\begin{aligned}
 &\min_{\rho} \quad \mathbf{f}^T \mathbf{u} \\
 &\text{s.t.} \quad \mathbf{K}\mathbf{u} = \mathbf{f} \\
 &\quad \quad m \leq m_{max}
 \end{aligned} \tag{2-3}$$

The maximum allowable weight is restricted to 50% of the solid weight. So the equation $m_{max} = \frac{m_{solid}}{2}$ holds. The structure in this case is a simple cantilever beam, where the left side is clamped, while the right side is free. The height-to-width ratio is $\frac{1}{3}$, in order to give a more clear representation of iterations. A vertical point load is attached to the right lower node of the beam, as can be seen in Figure 2-1.

As can be seen, there is some evolutionary behavior showing up. The overall pattern is somewhat the same, but the details are evolving during the iterations steps. How this iteration scheme exactly looks like is depending on the solution, as well on the solution method that is being used. In the upcoming chapters this will become more clear.

2.1.2 Mesh-refinement

To actually perform a topology optimization, the optimization solver uses this discretization of elements. In order to achieve the optimum solution, the solver determines whether an

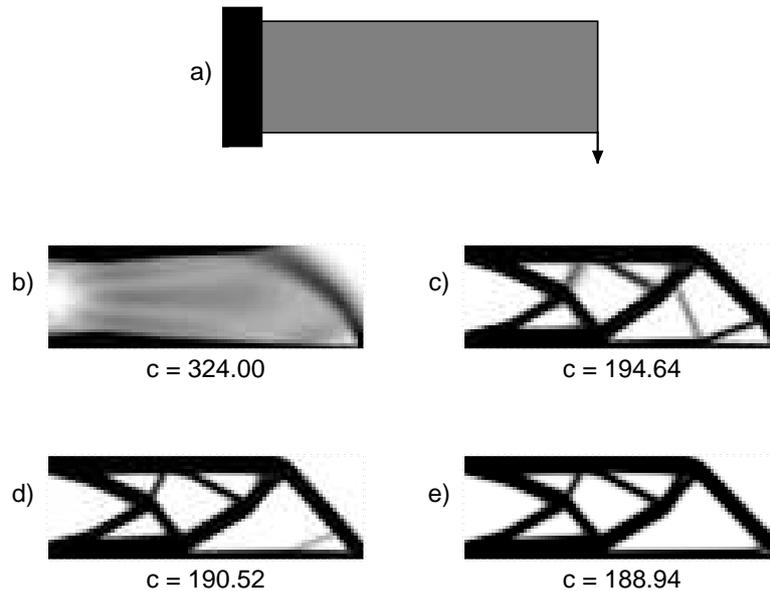


Figure 2-1: Evolution of iterations using the SIMP method. The beam is discretized by 90×30 elements. a) design problem, b) 5% of total iteration steps, c) 25% of total iteration steps, d) 50% of total iteration steps, e) final solution. The associated compliance values are shown below each figure, which represents a ratio of strain to stress (Appendix A-2).

element is a void (0) or solid (1) region. A collection of all these discretized elements then forms the topology of the complete structure.

Every topology optimization problems deals with the problem: *How should the mesh be refined?*, which refinement is fine enough to approach the reality? Of course, every continuum object needs to be discretized into a number of elements. This is basically a mesh-refinement. An increasing amount of mesh elements results in a longer computational time, but if the mesh-refinement is too rough, the solution does not represent the reality enough. In order to achieve the most optimal mesh-field, the trade-off between precision and computational time should be solved. It is very interesting to have a look at the manufacturing part of topology optimization. For example the resolution of the additive manufacturing device can be seen as the maximum amount of discretization elements, a finer mesh-refinement will from this point not lead automatically to a finer end product.

An example of the influence of different mesh-refinements is made, to show the importance of choosing a good mesh. The same configuration as defined in Figure 2-1 is used, this means the height-to-width ratio remains constant, while the mesh-refinement is changed. As can be seen in Figure 2-2 there are some notable changes in the optimization configurations. The number of elements thus influences the structural optimization result.

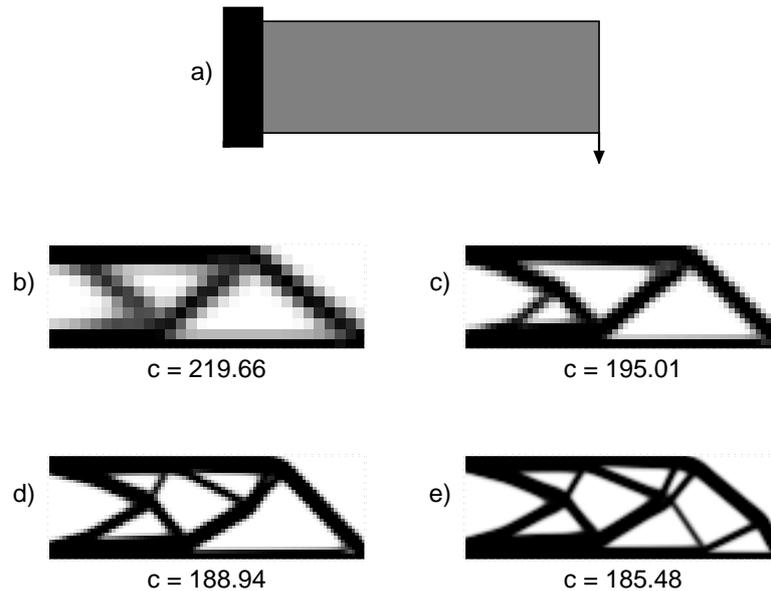


Figure 2-2: Dependency of topology on the mesh refinement using the SIMP method. The beam is discretized by b) 30×10 , c) 60×20 , d) 90×30 , e) 120×400 elements (Appendix A-3).

2.1.3 Volume fraction

As already stated in (2.1.1) the volume fraction is restricted to 50% of the solid weight, up to here. In this section the volume fraction is noticed. This volume fraction is derived from the maximum allowable weight of the structure. As already considered before, a big advantage of topology optimization is weight reduction. Although this volume fraction is most of the time seen as the biggest restriction of the optimization, in Figure 2-3 an example of different volume fraction is shown. This picture can be used to show the differences between certain volume fraction levels. As can be seen, the structure is largely dependent on the volume fraction, the layout is heavily changed when the volume fraction increases. The associated compliances are increasing also. This is pretty clear, since more volume fraction means more available material, which leads to increasing stiffness. The compliances in this example are thus not that suitable for comparison.

2.2 Solution methods

As already mentioned before, the used method to produce the optimization is the SIMP method. This method is just a way to transfer the original structure to the optimum structure in topological perspective. A Finite Element Analysis is used for calculating the problem. This method is very useful to calculate stiffness values. Since each element is described by four nodes, shear locking can occur. Methods to overcome this problem are not within the scope of this research, however. Although in section (2.1) the method is just used, without

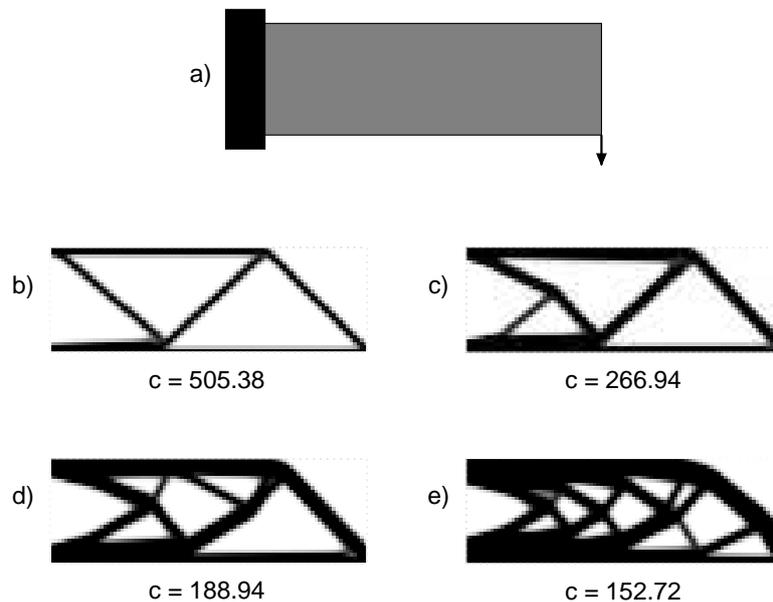


Figure 2-3: Different topologies for different volume fraction. The volume fraction is given as b) 20%, c) 35%, d) 50%, e) 65% (Appendix A-4).

any explanation; the method however, will get some more attention in this section. There are some more optimization methods around, this literature survey will only focus on the main two solution methods that are around and being used these days. Due to an overview and an example of comparison, the best method will be chosen for further usage in the literature survey.

2.2.1 SIMP method

As already can be deduced from Figure 2-2, an increasing amount of discretization elements results in a more complicated, porous, structure. But at the other hand we want many discretized elements, in order to mimic the reality. Now let's have a closer look at the Figure 2-2, it can be seen that Figure 2-2b consists of several *gray* regions, which is not desirable. Topology optimization should preferably result in a solution with only void (0) or solid (1) regions. These regions represent no or full material, respectively. Several regions in Figure 2-2b however, represent a gray region, which could be physically defined as a material with only a part of the element's density. By stating this, it can be concluded that gray regions are undesirable. To work around with this problem the *Solid Isotropic Microstructure with Penalization* (SIMP) can be used to avoid this (Rozvany et al., 1992). Reminder: only isotropic materials are considered. The SIMP approach is used to make intermediate densities unattractive, we are looking either for no (void regions) or full (solid regions) densities. Now recap (2-2), where the stiffness matrix K is mentioned. Suppose the discretization model to hold. For each j th element in the structure, a maximum stiffness K_0 can be derived, which corresponds to a fully solid element. The stiffness of the optimized element K_j is derived

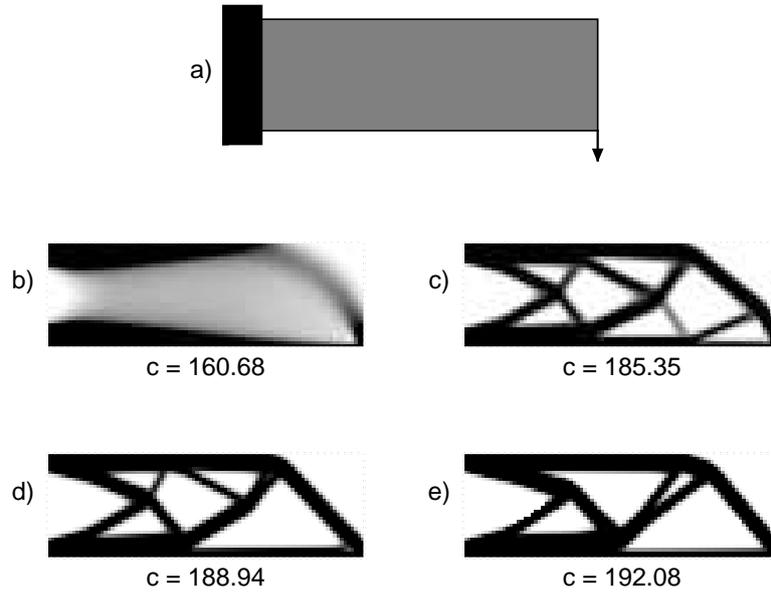


Figure 2-4: Different topologies for different penalization power. The penalty is defined by b) $p = 1$, c) $p = 2$, d) $p = 3$, e) $p = 5$ (Appendix A-5).

using the maximum stiffness and the density of this j th element. The following equation is a representation of the penalty-function of the SIMP method.

$$K_j = (\rho_j)^p K_0 \quad (2-4)$$

As can be seen, this penalty p , also named penalization power is an exponential function of the density. An example of the actual influence can be seen in Figure 2-4.

Usually, a penalty term of $p \geq 3$ should result in a void-solid division, which is a target in topology optimization. So a greater penalty results in a better result. However, the computational time is increasing also. So again for this parameter a trade-off should be made between precision and time.

In (A.9) the complete optimization scheme is shown. Each part of this scheme will be discussed later on.

2.2.2 BESO method

Besides the SIMP Method, the BESO method sure needs some attention. Using the *Evolutionary Structural Optimization* (ESO), an upgraded version of this method was found. The *Bi-directional Evolutionary Structural Optimization* (BESO) can be used within structural topology optimization (Querin et al., 2000), including compliance mechanisms (Huang and Xie, 2007).

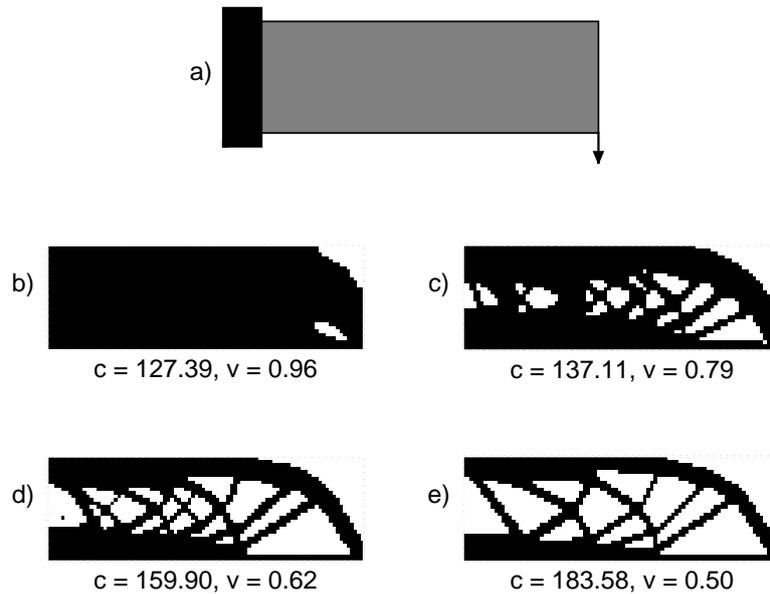


Figure 2-5: Evolution of iterations using the BESO method. The beam is discretized by 90×30 elements. a) design problem, b) 5% of total iteration steps, c) 25% of total iteration steps, d) 50% of total iteration steps, e) final solution.

The ESO concept can be seen as a process, whereby slowly inefficient material is removed, in order to achieve an optimal result. Here inefficient means not efficient toward the objective function. The BESO method uses this removal process, but also include, at the same time, an addition of material step. This explains the *bi-directional*-term. Within each iteration step, a *Constrain*-step is built in, to check whether or not the predefined constrained volume is conflicted or satisfied. A simplified flowchart of the BESO method can be found in (A.9). As already shown in (2.1.1), for this BESO method also, an evolutionary scheme can be made, just to visualize the whole method. The BESO optimization can be seen in Figure 2-5.

In contrast to the SIMP method, the BESO method starts from a solid structure. Therefore, the final solution can differ from another optimization method. This is mainly caused by the difference in the convergence approach. However, both solutions show the same kind of topology. And of course, both optimization methods can be fine-tuned, in order to achieve equal solutions. But for this time, only the standard parameters are considered.

2.2.3 Sensitivity analysis

As already depicted in the flowcharts in (A.9), the Sensitivity Analysis plays an important role in optimization processes. In this section the sensitivities of the compliance example will be explained. As can be seen in (A.9), the loop of the topology optimization starts with a sensitivity analysis. In this analysis the derivatives of the objective function are calculated, with respect to the design variables. In case of the compliance example as defined in (2-2),

the sensitivity can be seen as the derivative of the compliance over density.

In topology optimization is mainly worked with a moderate number of constraints, the *adjoint* method is used. In this method, the derivatives are not explicitly calculated, but a back-substitution is needed for each response and design variable. In order to get some more insight in the actual analysis, let's recap the formulation of (2-2), and combine it with (2-4). The minimum compliance example can now be formulated as:

$$\begin{aligned}
\min_{\rho_e} \quad & \mathbf{f}^T \mathbf{u} \\
\text{s.t.} \quad & \left(\sum_{e=1}^n \rho_e^p \mathbf{K}_e \right) \mathbf{u} = \mathbf{f} \\
& \sum_{e=1}^n \nu_e \rho_e \leq V \\
& 0 \leq \rho_e \leq 1 \\
& e = 1, \dots, N
\end{aligned} \tag{2-5}$$

In this formulation a couple of tweaks are made, regarding the previous formulations. Substitution of (2-4) in (2-3) results in (2-5). The stiffness of the total structure is discretized by a number of elements (1 to N), as showed in Figure 2-2. The summation of these elements e results in the total stiffness. The summation of all these element volumes results in the total volume V , while the density of each element should be within the range 0 to 1.

The objective function is minimize compliance, by varying the density. To compute this minimum, the derivative of the objective function should be computed, with respect to the design variables. Using the equilibrium equation $\mathbf{K}\mathbf{u} = \mathbf{f}$, the derivative of the original objective function $c(\rho)$ can be computed:

$$\frac{\partial c}{\partial \rho_e} = \mathbf{f}^T \frac{\partial \mathbf{u}}{\partial \rho_e} \tag{2-6}$$

Keep in mind, the stiffness matrix \mathbf{K} is typically very large. The computation needs to be done over each element e , which results in a very large computational time. In order to work around with this problem, an effective method is to define a zero function, also adjoint function, which will be added to the original compliance problem. Here, the adjoint vector $\tilde{\mathbf{u}}$ represents a fixed, real vector and satisfies the following adjoint equation.

$$\mathbf{f}^T - \tilde{\mathbf{u}}^T \mathbf{K} = 0 \tag{2-7}$$

Now adding this (2-7) to the original compliance example results in (2-8). This formulation is valid for any choice of $\tilde{\mathbf{u}}$, so we can basically take each expression we want. As long as this vector is fixed and real.

$$c(\rho) = \mathbf{f}^T \mathbf{u} - \tilde{\mathbf{u}}^T (\mathbf{K}\mathbf{u} - \mathbf{f}) \tag{2-8}$$

Now computing the derivative of (2-8) in a similar way of (2-6) results in

$$\frac{\partial c}{\partial \rho_e} = (\mathbf{f}^T - \tilde{\mathbf{u}}^T \mathbf{K}) \frac{\partial \mathbf{u}}{\partial \rho_e} - \tilde{\mathbf{u}}^T \frac{\partial \mathbf{K}}{\partial \rho_e} \mathbf{u} \tag{2-9}$$

Using the property of (2-7) result in the short, low-cost equation

$$\frac{\partial c}{\partial \rho_e} = -\tilde{\mathbf{u}}^T \frac{\partial \mathbf{K}}{\partial \rho_e} \mathbf{u} \tag{2-10}$$

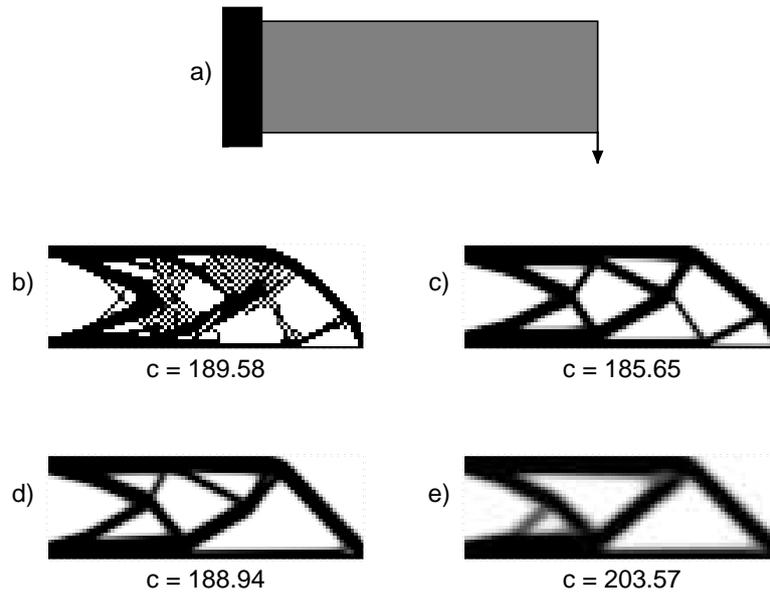


Figure 2-6: The filter radius r is here changed. The filter radius is given by b) $r=1.0$, c) $r = 1.25$, d) $r = 1.5$, e) $r = 3$ (Appendix A-6).

Now replacing the regular stiffness matrix \mathbf{K} with the penalty-termed stiffness as derived in (2-5), results in:

$$\frac{\partial c}{\partial \rho_e} = -p(\rho_e^{p-1})\mathbf{u}^T \mathbf{K}_e \mathbf{u} \quad (2-11)$$

Which can be seen as the sensitivity of the optimization problem. Please keep in mind; the derivatives depicted in (2-6) and the subsequently derived derivatives assuming that \mathbf{f} is not dependent on the element's densities ρ_e , which is not very common.

2.2.4 Filtering

The next step in optimization, as can be seen in (A.9) is a filtering technique. The calculated sensitivities are filtered, in order to prevent so-called checkerboard patterns (Sigmund and Petersson, 1998). An increased number of elements will not automatically lead to a solution that can actually be additive manufactured. The additive manufacturing has its own resolution, in other words, the minimum thickness it can produce. To work around with this problem, a filter radius can be used in the topology optimization scheme. By modifying the element sensitivities of the compliance, using a filter radius, a weighted average of the element itself and its eight surrounded elements can be made.

Using this weighted average, the iteration scheme determines a solution, which fulfill the filter radius specification. To get some more insight of this working principle, an example is made and can be seen in Figure 2-6.

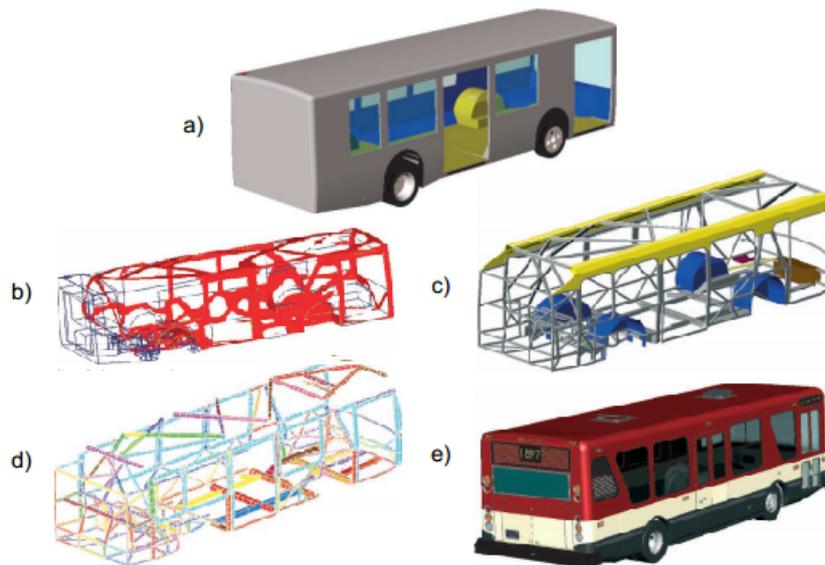


Figure 2-7: Design of a lightweight city bus. a) Initial design, b) topology optimization, c) CAD representation of the topology optimization, d) sizing optimization, e) final design (Thomas et al., 2002).

As can be concluded, a filter radius too low results in a checkerboard problem, which may be not manufacturable. By varying the filter radii this problem can be overcome. However, picking a filter radius too high can result in a non-optimal solution, since this will lead to thicker material trusses.

2.3 Applications

Topology optimization can be seen as a very effective way of creating optimum structures. As already explained in (2.1), it can be used to maximize stiffness for a lighter structure. Now let's have a look at the actual practical examples of the topology optimization. And after, the main focus of the literature survey will be explained.

2.3.1 Statics

A very interesting example is an optimization of a city bus. The main objective here is to reduce the weight of the bus, by doing this the gasoline and thus fuel costs can be reduced. Using different optimization programs the final bus design is modeled. The shape of the windows was decided by the results of the structural topology optimization. A framework of this process can be seen in Figure 2-7

Another example of the need of topology optimization is found in the MEMS industry, for example micro-scale compliant mechanisms. A common challenge in MEMS is to produce very little prescribed displacements. Using topology optimization can be very useful to fulfill this need. So in this case, the topology optimization is not mainly used to reduce weight for

example, but it is used to actually achieve a certain goal. By varying the associated objective functions and constraints, a lot of possibilities can be defined in topology optimization.

2.3.2 Dynamics

The benefits of topology optimization in statics is straightforward. However, the optimization can also be of need in the dynamics.

Up to now, only statically loaded structures are considered. However, periodically loaded structures can also be optimized using structural topology optimization. Dependency of the optimum topology is shown for a structure with respect to different excitation frequencies (Ma et al., 1995).

But one can also think of the need of topology optimization to achieve a certain target in the dynamical domain. For example a structural topology optimization of vibrating structures, with specified eigenfrequencies and eigenmodes (Maeda et al., 2006). Here topology optimization is used to achieve a high eigenfrequency for example. Here this eigenfrequency can be seen as an objective function which should be maximized.

2.3.3 Other domains

Upcoming research is done in fluid design. For example the optimum structure of a channel to achieve a certain velocity and Reynolds number. Or in the (micro)fluidics, for example in micro mixers. Here topology optimization is used to optimize the mixing process of certain fluids (Andreasen et al., 2009).

Work is done in multiphysics, although there is only one physics, this term is widely used in engineering. Within this multiphysics multiple domains are coupled together to achieve a realistic behavior. While designing a micro-actuator in MEMS, thermal and electrical behavior interfere. The coupling of these domain results in a multiphysics actuator. Topology optimization can be used for both domains, and both domains can be coupled together, in order to achieve the overall optimal actuator (Sigmund, 2001b).

2.4 Design of supports

While designing compliant mechanisms we have considered a structure, with boundary conditions and objective functions. Although the boundary conditions for the support are not defined in (2-2), the compliant example does include a clamped end on the left hand, as can be seen in Figure 2-1a. However, the main objective is to maximize stiffness, minimize compliance. The position of the support can maybe changed in this example, while aiming at minimizing compliance. If this support can be varied, the support should be placed right under the load case. This results in a zero displacement and consequently infinite stiffness. Different supports will lead to different optimum structures, which is pretty straightforward. In this section, the design of supports will be discussed, which is also the main target in the upcoming literature survey and sequential thesis project.

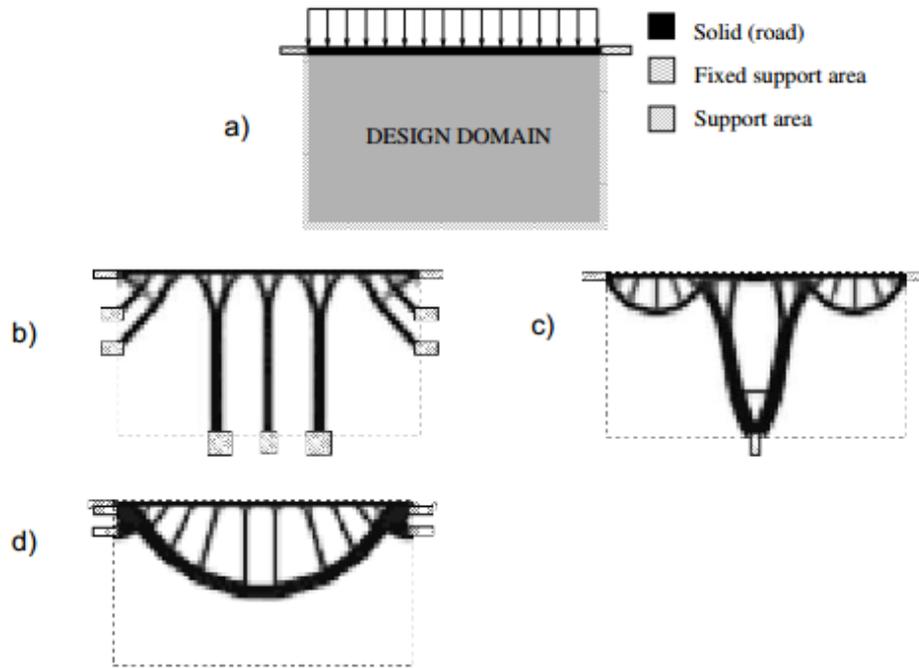


Figure 2-8: Example of design of supports with different support cost functions r_c a) Initial design, b) $r_c = 1$, c) $r_c = 10$, d) $r_c = 20$ (Buhl, 2002).

2.4.1 Optimizing supports

When optimizing the design of support, the prescribed support locations, as seen in Figure 2-1a are not longer prescribed, but interpreted as a design variable. A well-known bridge example is shown (Buhl, 2002). Here a bridge is designed and optimized to make a road in a deep canyon. In this Bridge example three cases are considered, as depicted in Figure 2-8.

In this example a pavement is modeled as a solid, clamped, side at the top of the design domain. The road experiences a distributed force as a representation of continuing traffic over the bridge. The bridge is fixed to the upper left and upper right edges. The sides and the bottom of the design domain are considered as possible support areas. A volume constraint of 20% is applied, the number of support constraint should yield maximum 20% of the total number of supports.

In Figure 2-8b, the cost of support is equally distributed for the sides and bottom. This ratio of cost $r_c = 1$, so without any other constraints, this optimization should be the perfect bridge structure with this constraints, and will result in the minimal compliance.

The pillars however, could be very expensive, or hard to place under this bridge. The second optimization Figure 2-8c is award a ratio of cost $r_c = 10$, this means a linear support cost function from 1 to 10 (top edge to bottom edge). Therefore, material at the bottom is undesirable, as can be seen only one support remains. In the third case Figure 2-8d a cost function of $r_c = 20$ is applied. By doing this, the support material at the bottom is very unlikely, and no pillar exists anymore. An application example could be a very deep canyon, where pillars are unwanted, but a maximum stiffness is wanted.

Design of supports is a promising optimization technique and can be used in a wide range of applications. This literature survey will continue in the next chapter onto this optimization

domain. A concrete working direction will be defined and further investigation will continue on this subject.

2.5 Conclusions

Structural topology optimization is a very promising way of achieving several benefits. These benefits can vary from active money saving, using less structural material (2.1), to passive money saving, the bus example (2.3.1), where removal of material results in a lighter bus and less fuel costs. Topology optimization can also be used to achieve specific targets, for example in the MEMS industry (2.3.1) and within the dynamics domain. Vibrating structures can be optimized (2.3.2) to achieve desired eigenfrequencies or eigenmodes.

We consider three different categories of optimization (1.1), namely: sizing, shape and topology optimization. Sizing optimization only changes for example cross-sections of a truss structure. Shape optimization changes the shape of the material, without removing or adding material. Topology optimization defines an optimal topology solution for a given problem, this is the main target of this literature survey.

Topology optimization can be done with several solution methods (2.2). In this survey two main methods are considered. The SIMP method (2.2.1) uses a penalization method, to prevent so-called grey regions in the optimal solution, since the material should be void or solid, and not partially present. The SIMP method optimizes with respect to the constraint, the best objective function. The BESO method (2.2.2) combines addition and removal of material until it reaches the volume constraint. An overview of both optimization processes can be found in (A.9).

There are however some considerations with topology optimization. A sensitivity analysis (2.2.3) is made, followed by associated filtering, to prevent checker-boarding (2.2.4) patterns. To remove this non-realistic solution, in terms of manufacturing, the filter radius can be tuned. Setting the radius too low results in checker-boarding, but setting the radius too high can skip other optimal solutions by not allowing fine features to emerge.

The design of supports will play a big role in the upcoming literature survey and sequential thesis project. Varying support locations results in other optimization results (2.4). This design of supports can also be used to actively achieve an optimal solution regarding the actual number and placement of supports. The bridge example (2.4.1) showed a way to improve a structure, with respect to external factors. A deep canyon could be very unlikely to support using pillars, although this will result in a stiffer construction. Using a ratio of support cost can give a mathematical insight in the relation between cost of supports and stiffness. Especially in compliance problems the exact support location may not be fixed, and some relaxation of this support location can lead to actual really good optimization results. In the upcoming chapter some deeper investigation will be done regarding this subject. Although not explicitly documented, there are some numerical results available of all the executed optimizations can be found in (A.1). These values can be used for upcoming study, in order to make a decision regarding the choice of optimization parameters.

Topology Optimization for Engineers

In chapter 2 some simple cases and basic properties of topology optimization are described. In this chapter the philosophy and possibilities of topology optimization is taken a step further. Topology optimization for engineers can be used to solve a variety of mechanical problems. The simple solution method, the Optimality Criteria, can be used for simple compliance problems. When dealing with more complex problems, there is a need for a different solution method. The Method of Moving Asymptotes, as described in section 3.1 can be used to overcome this.

When solving mechanical problems, some advanced applications can be helpful, to reflect the actual design problem. In section 3.2 a number of applications are implemented and described. The main advantage of topology optimization is within the additive manufacturing domain, an example to 3D cases is given in section 3.3. In section 3.4 the use of topology optimization within compliant mechanisms is explained. Section 3.5 concludes this chapter.

Using the attached MATLAB codes (B.3 up to B.6 and C.1 up to C.8) the problems in this chapter can be solved.

3.1 Solution method: MMA

Besides the described solution method in (2.2) and up-following (2.2.3), there are some other solution methods around. For simple compliance problems, like the cantilever problems, the Optimality Criteria method from (2.2) can be used. This method is easy, fast and very cost-efficient. The Optimality Criteria method is very useful for compliance problems, since this method always wants to add material, in order to achieve a high stiffness. However, in more complex problems, this method is insufficient.

The Method of Moving Asymptotes (Svanberg, 1987), also known as MMA, is a mathematical programming algorithm which is very suitable for topology optimization. This method can be used to restrict the optimization problem to multiple constraints, and multiple design variables. In the upcoming chapter a number of applications, with the use of MMA will be

shown. The MMA program solves the following optimization problem:

$$\begin{aligned}
\min_{x,y,z} \quad & f_0(x) + a_0 \cdot z + \sum_{i=1}^m (c_i y_i + \frac{1}{2} d_i y_i^2) \\
\text{s.t.} \quad & f_i(x) - a_i \cdot z - y_i \leq 0, & i = 1, \dots, m \\
& x_j^{\min} \leq x_j \leq x_j^{\max}, & j = 1, \dots, n \\
& y_i \geq 0, & i = 1, \dots, m \\
& z \geq 0
\end{aligned} \tag{3-1}$$

In this formulation, f_0 is the objective function, while f_i represents the constraint functions, defined by the number of constraints m . A vector of design variables x will be updated, using y and z as positive optimization variables. This vector should be in-line with the number n of defined constraints. The programming parameters a_0 , a_i , c_i , d are so-called *magic numbers of MMA* and can be used to determine the type of optimization problem.

In order to use this method for a compliance problem, the author of (Svanberg, 1987) suggested some MMA constants: $a_0 = 1$, $a_i = 0$, $c_i = 1000$, $d = 0$. Using these constants and at the same time writing the function in terms of a compliance problem, results in:

$$\begin{aligned}
\min_x \quad & \mathbf{c}(\mathbf{x}) \\
\text{s.t.} \quad & f_i(\mathbf{x}) \leq 0, \quad i = 1, \dots, m \\
& 0 \leq x_j \leq 1, \quad j = 1, \dots, n
\end{aligned} \tag{3-2}$$

Here, the optimization variables y and z should be zero at the optimum. The vector of design variables \mathbf{x} is in this example just the density of each element.

This routine is implemented using the available MMA-code, which can be found in (D.1) and (D.2)

3.1.1 OC versus MMA

In this section a comparison between the Optimality Criteria (with density filter) and the MMA routines is made. As already stated before, the MMA-routine is very useful, dealing with multiple constraints, while the OC-routine is not handy for these types of problems. In order to make a good comparison, the simple compliance problem from (2.1.1) will be optimized using these two routines. A comparison will be made regarding the final compliance, as well as the number of iterations and total optimization time. An evolutionary scheme, related to Figure 2-1 is produced. In this problem, the Optimality Criteria is applied, using sensitivity filtering. Although this method is usable in practice, it is mathematically inconsistent. Density filtering is a solution to overcome this. As can be seen in Figure 3-1, both methods will produce a somewhat same result, but there are some differences notable. When looking at the process, it seems like Figure 3-1a goes slightly faster towards its final state, while the MMA (Figure 3-1c) needs some more time to get a slightly better result, in contrast to Figure 3-1a. The number of iterations displays some interesting results: the Optimality Criteria needed 40% more iterations to get the final result, with respect to the MMA. The computational time however, is in favor of the Optimality Criteria. The OC method is approximately four times faster than the MMA.

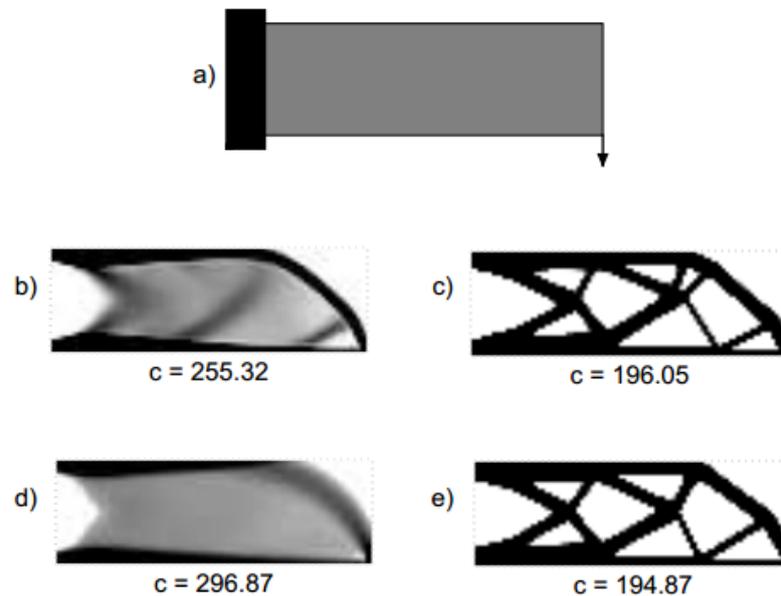


Figure 3-1: Evolution of iterations using the SIMP method. The beam is discretized by 90×30 elements. a) design problem, b) 5% of total iteration steps of Optimality Criteria, c) final solution of Optimality Criteria, d) 5% of total iteration steps of MMA approach, e) final solution of MMA approach (Appendix A-7).

So for this particular example, the MMA results in a better, stiffer result with less iterations with respect to the OC. However, the calculation time for each iteration step is a lot longer with respect to the OC. As a reference, all the calculated data can be found in (Appendix A-7).

3.2 Advanced applications

Using the now defined code, a lot of tweaks and application can be made. In this section a small amount of useful applications will be depicted. First, some words will be stated about restrictive regions, i.e. active or passive area's in the design domain. Second, an example of multiple load cases will be discussed. The next subsection results in some thoughts about self-weight of a structure. An example of a compliant mechanism synthesis will be made. And at last, but not at least, an example of a 3D problem will be displayed, just to give some more insight in topology optimization.

3.2.1 Restrictive regions

Topology optimization in general, can be used for a variety of open problems, in some cases, however, some restrictions should be implemented in the optimization problem. A particular example is to implement a so-called passive region. In this region, there should be zero

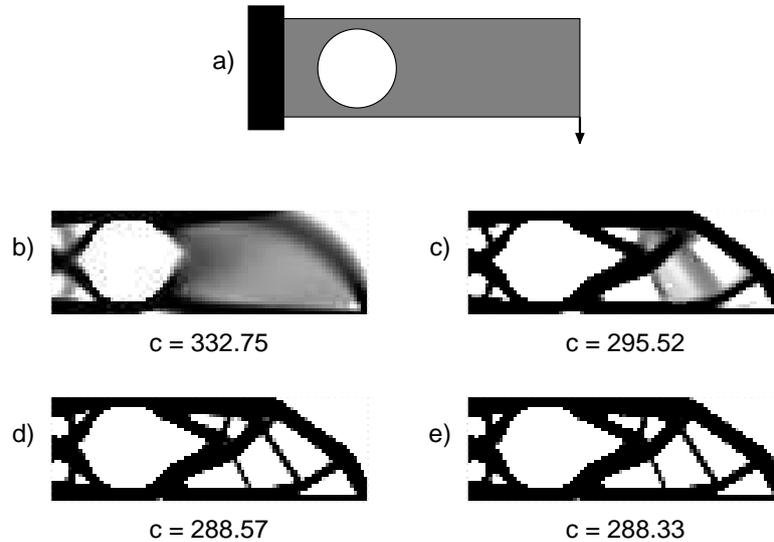


Figure 3-2: Evolution of iterations using the SIMP method and the application of a passive region. The beam is discretized by 90×30 elements. a) design problem, b) 5% of total iteration steps, c) 25% of total iteration steps, d) 50% of total iteration steps, e) final solution (Appendix A-8).

material, for example because a certain space needs to be free from material to apply a screw. As can be seen in Figure 3-2, a passive region is implemented by a circular area, which always should remain free from any material. The evolution of iteration steps give a nice view in this process.

The same procedure can be applied for active regions. In certain cases it could be very helpful to pinch material on certain places, for example for adhesive purposes. In this case, depicted in Figure 3-3 the evolutionary scheme gives a very clear view on the process.

3.2.2 Multiple load cases

Some problems can occur when defining multiple loads. A choice can be made whether to choose one or multiple load cases. Each choice will result in a different solution. When applying simultaneous load cases, the optimization solution will act as if it is an optimization of the equilibrium of the two loads. When using separated load cases, the structure will be more resistant to buckling and much stiffer when one of the loads is removed, with respect to the single load case. Of course, this will result in a longer computational time.

A small example of this load case dilemma can be found in Figure 3-4. Clearly, Figure 3-4 only make sense when both loads act simultaneously.

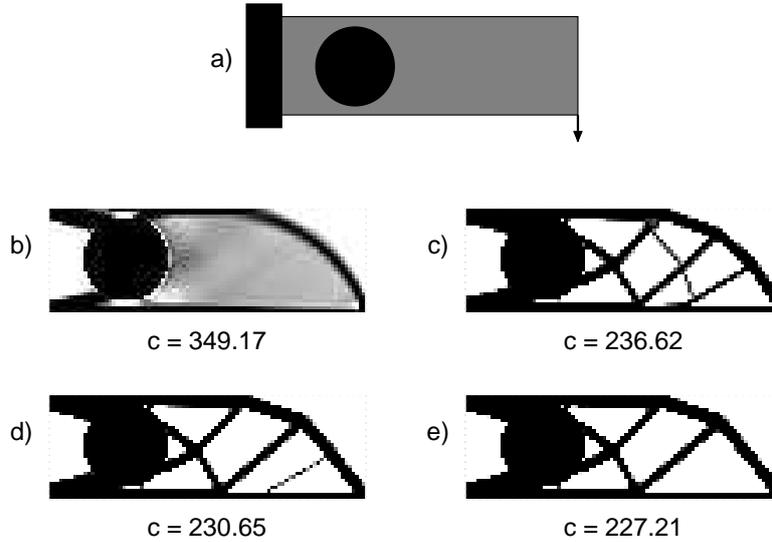


Figure 3-3: Evolution of iterations using the SIMP method and the application of an active region. The beam is discretized by 90×30 elements. a) design problem, b) 5% of total iteration steps, c) 25% of total iteration steps, d) 50% of total iteration steps, e) final solution (Appendix A-8).

3.2.3 Self-weight implementation

Up to here, the influence of gravity is not taken into account. However, when optimizing towards an optimal solution in reality, gravitational force should be implemented. By implementing this self-weight, some density ρ_0 should be used, to give each element a natural density, when the element is a complete solid region. This ρ_0 can be used to reflect the material properties of the optimization material.

When this ρ_0 is set to zero, the influence of self-weight is completely removed. The total resultant self-weight can be calculated by a summation of all the weights of the elements, as a combination of gravitational force g , the optimization density, and the material density ρ_0 . This total resultant self-weight force can be compared to the external force. A weight factor W is introduced, as a ratio of the resultant gravitational force to external force. When this ratio is zero, no self-weight is taken into account. When this ratio is high, the optimization routine tends to neglect the external force, as this becomes only a fraction of the total force. In each iteration, a calculation of the current self-weight is made. Each element, combined with an element density, is divided to its four nodes. These four nodes then experiences a gravitational force of one fourth of the element density times the material density ρ_0 .

This extra term of force needs an adjustment on the sensitivity analysis, as derived in (2-11). The self-weight acts like an external force, the derivative of the compliance of this force needs to be added to the original sensitivity analysis, to account for this. The updated sensitivity can be seen in (3-3).

$$\frac{\partial c}{\partial \rho_e} = 2\mathbf{u}^T \frac{\partial \mathbf{f}_{sw}}{\partial \rho_e} - p(\rho_e^{p-1})\mathbf{u}^T \mathbf{K}_e \mathbf{u} \quad (3-3)$$

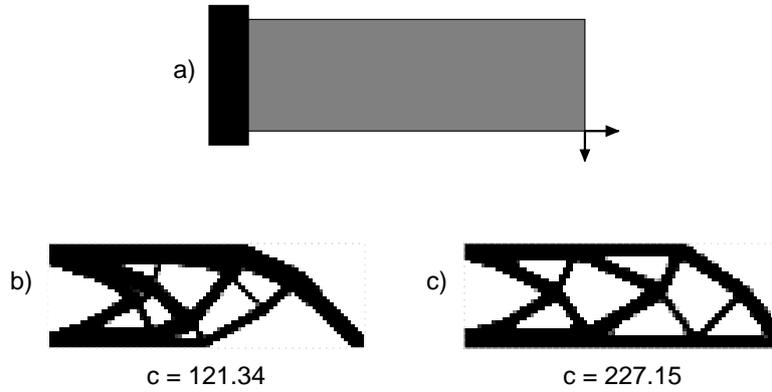


Figure 3-4: Solution regarding different load cases. a) One load case: vertical, horizontal, b) Two separated load cases (Appendix A-9).

Using the same SIMP method as before, a problem comes up. When optimizing in the lower density area, the ratio of the first and the second term in (3-3) becomes crucial and tends to prevent a complete solid/void pattern for the solution. An alternative interpolation scheme could overcome this problem. A linear profile is selected, under a certain pseudo-density ρ_c . Above this pseudo-density, a penalized E_p is calculated, just like before (Bruyneel and Duysinx, 2005). The interpolation scheme can be seen in (3-4). An example of self-weight implementation can be seen in Figure 3-5. Here, a penalty factor of $p = 5$ is used, in order to force a black-white solution.

$$E_p = \begin{cases} \rho^p E_0 & \rho_c < \rho \leq 1 \\ \rho(\rho_c^{p-1})E_0 & 0 < \rho \leq \rho_c \end{cases} \quad (3-4)$$

3.2.4 Continuation method

With the introduction of the self-weight, as explained in (3.2.3) some serious problem regarding the optimal solution comes up. Since the introduction of additional (self-weight) forces, the chance of getting close to the global optimum has decreased. One way to overcome this problem is by implementing a so-called continuation strategy (Groenwold and Etman, 2010). While using this continuation method, an unpenalized material distribution is used, for the first number of computational cycles. After a certain number of iterations, the penalty is increased with each iteration, up to a predefined maximum penalty. When this maximum penalty is achieved, this penalty is used along the iteration scheme; up until the convergence criterion is met.

$$p^i = \begin{cases} 1 & i \leq 20 \\ \min \{p^{max}, 1.02 \cdot p^{i-1}\} & i > 20 \end{cases} \quad (3-5)$$

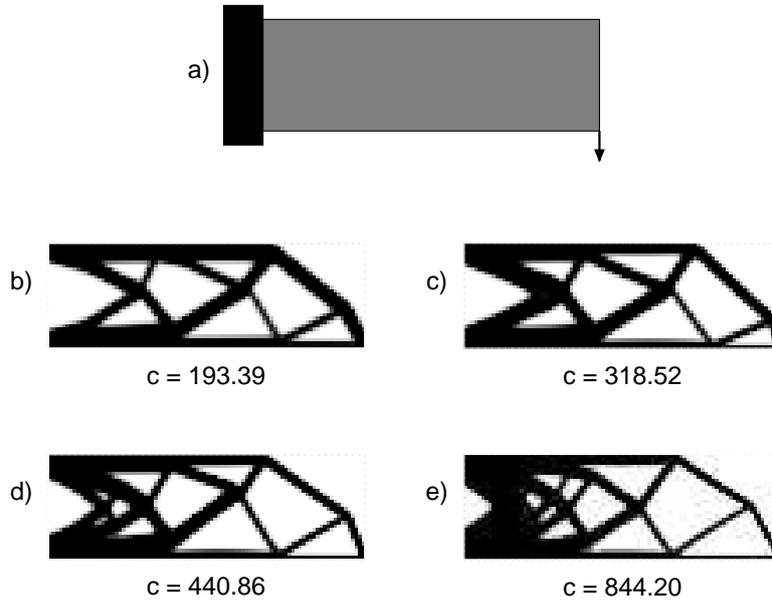


Figure 3-5: Influence of self-weight of the structure. A weight factor W can be defined as the ratio of the resultant gravitational force to external force. a) design problem, b) no self-weight ($W = 0$), c) $W = 1$, d) $W = 2$, e) $W = 5$ (Appendix A-10).

In (3-5) the increasing factor of 1.02 can be varied in the code, this value, however, seems to be a decent value for the compliance problems. The threshold value of the penalization (20) can also be varied.

3.2.5 Different filter techniques

Up until now, sensitivity filtering is used. Although this method is usable in practice, it is mathematically inconsistent. Density filtering is a solution to overcome this (Bourdin, 2001). The density filter transforms the original densities ρ_e to filtered densities $\tilde{\rho}_e$.

$$\tilde{\rho}_e = \frac{1}{\sum_{i \in N_e} H_{ei}} \sum_{i \in N_e} H_{ei} \cdot x_i \quad (3-6)$$

In this equation (3-6) the filtered density is computed by taking a weight factor H_{ei} over the set of elements N_e . This weight factor H_{ei} is zero outside the filter radius, while the operator $\Delta(e, i)$ is defined as the distance between the center of element e and the center of element i . The weight factor H_{ei} is defined as:

$$H_{ei} = \max \{0, r - \Delta(e, i)\} \quad (3-7)$$

The sensitivities with respect to the design variables ρ_e can be calculated accordingly:

$$\frac{\partial c}{\partial \rho_e} = \sum_{i \in N_e} \frac{\partial f}{\partial \tilde{\rho}_e} \frac{\partial \tilde{\rho}_e}{\partial x_i} \quad (3-8)$$

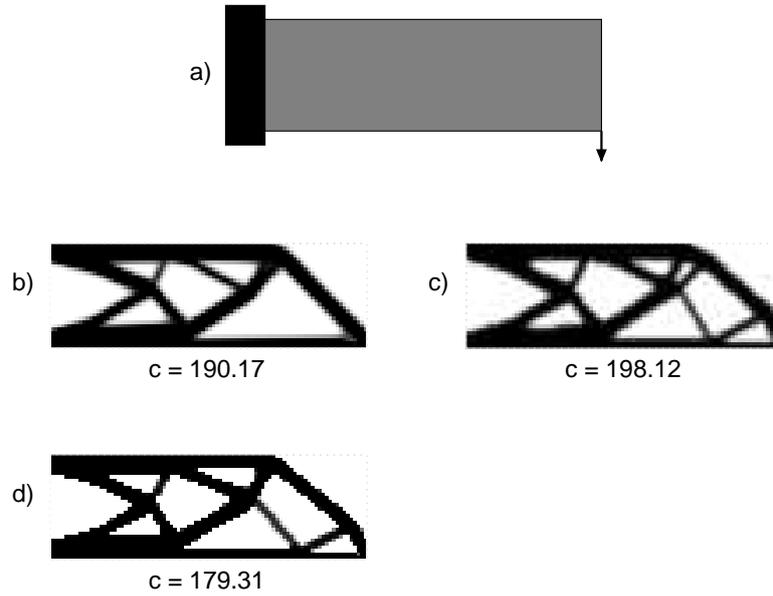


Figure 3-6: Influence of different filter techniques, using a OC solution method and continuation method. a) design problem, b) sensitivity filter, c) density filter, d) heaviside projection filter (Appendix A-11).

Another problem that could come up, when using self-weight, is the existence of gray patterns. As already discussed before, we prefer to produce a black-to-white pattern, to be able to actually produce the optimum result using additive manufacturing.

One way to obtain a black-and-white solution is using the Heaviside projection filter (Guest et al., 2004). The Heaviside filter can be seen as an upgrade of the density filter. This step function projects the filtered density $\tilde{\rho}_e$ to a projected filtered density $\bar{\rho}_e$. This $\bar{\rho}_e$ is defined as:

$$\bar{\rho}_e = \begin{cases} 1 & \text{if } \tilde{\rho}_e > 0 \\ 0 & \text{if } \tilde{\rho}_e = 0 \end{cases} \quad (3-9)$$

Since a gradient-based optimization is used, a smooth formulation for this Heaviside projection can be defined

$$\bar{\rho}_e = 1 - e^{-\beta \tilde{\rho}_e} + \tilde{\rho}_e e^{-\beta} \quad (3-10)$$

In this equation (3-10) the parameter β can be used to make a smooth approximation. This β is gradually increased from 1 to 512 by multiplying this value by 2 at every 50 iterations. Of course, this can be varied, but literature study suggests this approach. Also, this method should adjust the sensitivities of the function $f(\bar{\rho}_e)$, with respect to the filtered densities $\tilde{\rho}_e$ accordingly, as can be seen in (3-11).

$$\frac{\partial f}{\partial \tilde{\rho}_e} = \frac{\partial f}{\partial \bar{\rho}_e} \frac{\partial \bar{\rho}_e}{\partial \tilde{\rho}_e} \quad (3-11)$$

A comparison of the three used filter methods can be seen in Figure 3-6.

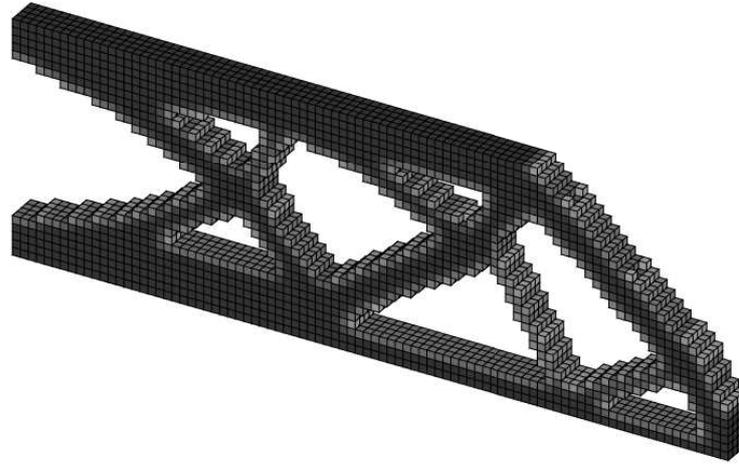


Figure 3-7: Compliance example of 3D. The design problem: clamped on all degree of freedoms on the left hand side of the beam, one downward load, attached at the right-hand side, at the bottom node in the y-direction, at the middle node in the added z-direction.

3.3 Turning 2D into 3D

While most topology optimization problems are displayed as 2D-results, the main advantage of topology optimization is found in relation to additive manufacturing. Having a 3D implementation is thus crucial. In order to work around with this additional dimension, a tweaked code is produced, which is able to calculate a 3D optimization problem.

Another dimension will add also an additional computational load. For now, only one example is depicted, just to show a working code. As can be seen in Figure 3-7, the same loads and constraints are applied. However, the third added dimension z , is now also implemented in this problem. The clamped left-hand side is fixed for all its degree of freedom, including the z -direction. A simple point load is applied to the right, in the middle of the z -direction. Because of the small number of elements in this z -direction, no difference can be found in the distribution of the elements in this z -direction. However, when discretizing in more elements, an expected discrepancy can be seen. While the example shown in Figure 3-7 is pretty clear and easy, the code actually has some more options. All the previous advanced applications are now available. An additional restrictive region is implemented, in the sense of a sphere, which can be either active or passive. The solution method can be varied, as well as the filter method.

3.3.1 Gray-scale filter

A new filter is introduced, namely a gray-scale filter. This gray-scale filter is a very powerful filter overlay to enable white-black regions (Groenwold and Etman, 2009). Because of its easy implementation and proven effectiveness for 3D applications (Liu and Tovar, 2014), this new filter is introduced. Gray-scale filter is used to further achieve black-white regions, by

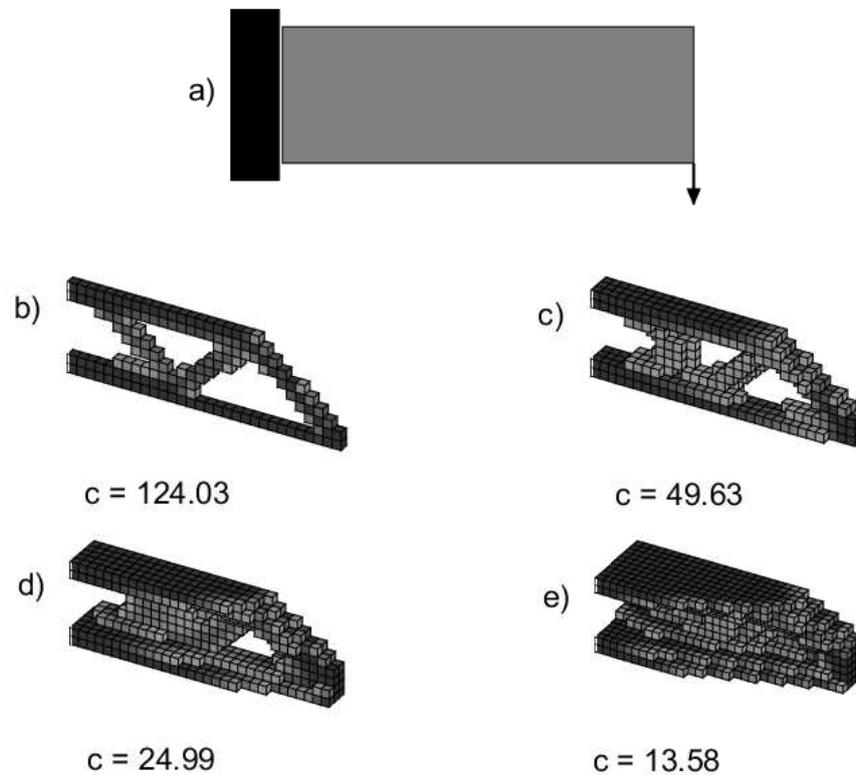


Figure 3-8: Influence of lateral elements of the structure. The number of lateral elements in the z-direction are varied. A continuation method is used, as well as a sensitivity filter with gray-scale filtering. The beam is discretized by b) $30 \times 10 \times 1$, c) $30 \times 10 \times 3$, d) $30 \times 10 \times 5$, e) $30 \times 10 \times 10$ elements (Appendix A-12).

introducing an exponent q . The working principle of gray-scale filtering can be seen in (3-12). The standard Optimality Criteria is a special case of gray-scale filtering with $q = 1$.

$$x_i^{new} = \begin{cases} \max(0, x_i - m) & x_i B_i^n \leq \max(0, x_i - m) \\ \min(1, x_i + m) & x_i B_i^n \geq \min(1, x_i - m) \\ (x_i B_i^n)^q & \text{otherwise} \end{cases} \quad (3-12)$$

The main advantage of the three dimensional optimization is of course the third dimension. This number of lateral elements can be varied, to see some interesting results. In Figure 3-8 this variation of lateral elements can be found. The force is pointed downwards, just like the simple 2D cases. This force however, is kept at the same spot each variation, in order to actually see some really nice results.

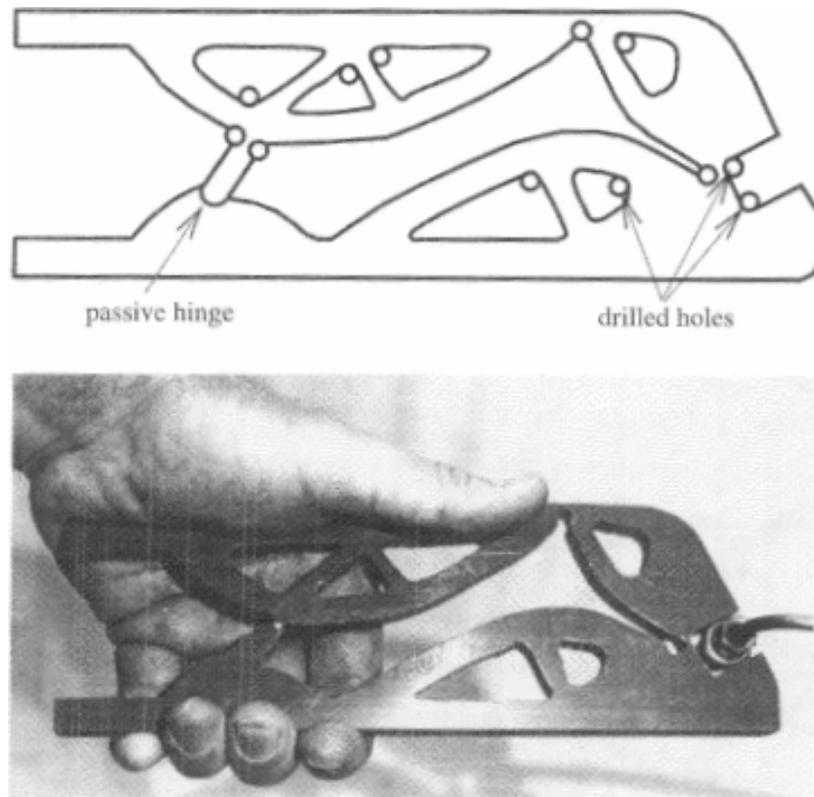


Figure 3-9: Interpretation and realization of a hand tool (Sigmund, 1997)

3.4 Compliant mechanisms

Compliant mechanisms are very popular nowadays. Besides to the compliance examples, which mainly rely on their stiffnesses; compliant mechanisms are used for their mobility. This mobility comes mainly from the flexibility of the mechanisms. These mechanisms can be manufactured easily with 3D printing, so the need for topology optimization is quite big. Especially within the MEMS-domain, these compliant mechanisms can be helpful. Using topology optimization, the optimal structure of very small compliant mechanisms can be designed. A typical example of the use of compliant mechanism design can be seen in Figure 3-9.

3.4.1 Inverter and amplifier

As can be seen in Figure 3-9 an inverter can be useful within the domain of small compliant mechanisms. This inverter can be used to invert an input displacement to a reversed output displacement, while maintaining almost the same amount of movement.

While designing these compliant mechanisms, it is very common to have an in- and output displacement, instead of an in- and output force load. Therefore, a small spring is introduced to the in- and output nodes. These springs will convert the force into a direct displacement. By varying these spring stiffness, one can achieve different inputs and consequentially designs. In order to solve some compliance problems, a design problem is formulated in Figure 3-10. As

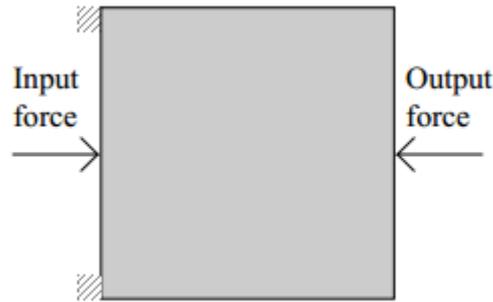
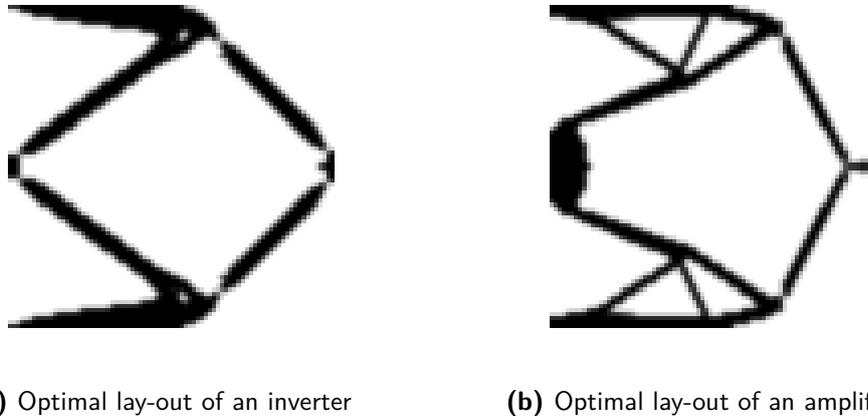


Figure 3-10: Compliant mechanism design problem



(a) Optimal lay-out of an inverter

(b) Optimal lay-out of an amplifier

Figure 3-11: Two compliant mechanisms. The inverter inverts a input of $d_{in} = 891$ into an output of $d_{out} = -899$, which results in a displacement gain of $G_d = -1.01$. The amplifier converts an input of $d_{in} = 18.42$ into an output of $d_{out} = -40.89$, which results in a mechanical displacement gain of $G_d = -2.22$ (Appendix A-13).

can be seen in Figure 3-10, these input and output forces are attached to predefined springs, in order to describe a displacement field. Using the boundary conditions at the upper and lower left corners, and using the prescribed input displacement; the optimal topology can be determined for maximizing the output displacement. In Figure 3-11a the optimal topology can be seen for an inverter mechanism. Here, the main objective is to convert a positive input into a negative output, while maintaining the same absolute displacement. In Figure 3-11b an example of an amplifier can be found. This mechanism also converts a positive input into a negative input, but also doubles the amount of displacement at the output side. Keep in mind that these compliant mechanisms are flexible and only can be used for very small displacements. The displacement patterns of both the inverter and amplifier are correct, however, not included in this report. Due to the geometry of the structures, as well as using small displacements, the deformed shape of the structure looks almost the same as the optimal topology lay-out. In this section there is no need to display the deformed shapes. In the next section however, some displacements are plotted for a gripper problem.

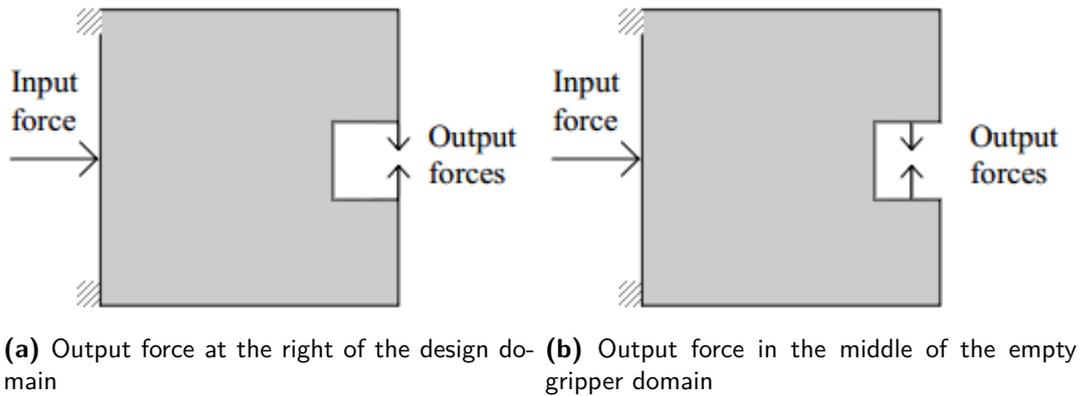


Figure 3-12: Design problem for micro-gripper

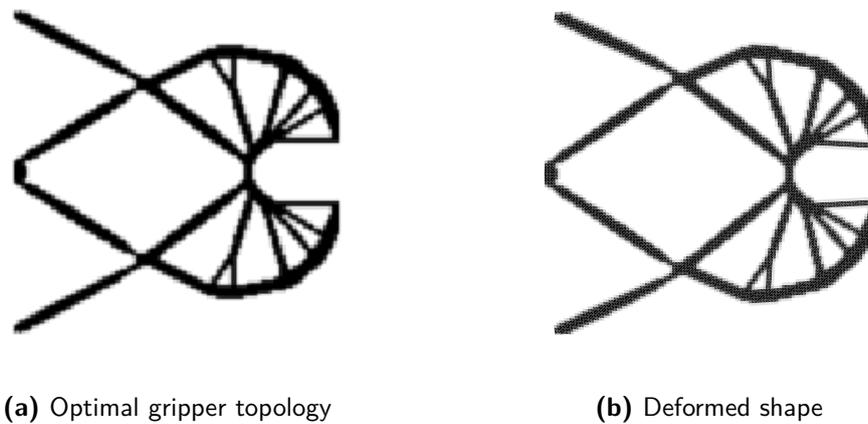


Figure 3-13: Optimal topology and displacement pattern for design problem Figure 3-12a, a horizontal input on the left, a vertical output on the outer right. The gripper inverts a input of $d_{in} = 76.48$ into an output of $d_{out} = 28.06$, which results in a displacement gain of $G_d = 0.73$.

3.4.2 Micro-gripper

In (3.4.1) only horizontal displacements are taken into account. However, when taking a look at Figure 3-9 there is also a conversion step needed to translate horizontal action into vertical output. Two simple design cases are depicted in Figure 3-12. For this particular design problem, the white area can be seen as a restricted area, where no material is allowed, a void region. In Figure 3-12a an output force at the right is requested, in example for gripping a small sphere. In Figure 3-12b the same void region is considered. However, an output force is requested in the middle of the area, for example grabbing a small cube.

The optimal topology for design problem Figure 3-12a can be seen in Figure 3-13a. Here, a maximum volume of 20% is allowed, while respecting the fixtures as described in the problem statement. A displacement field is plotted in Figure 3-13b, where a small displacement input is given. As can be seen, the jaws are slightly pulled together, just enough to grab a sphere. The same solutions, but now for design problem Figure 3-12b, can be found in Figure 3-14a and the subsequent displacement field in Figure 3-14b.

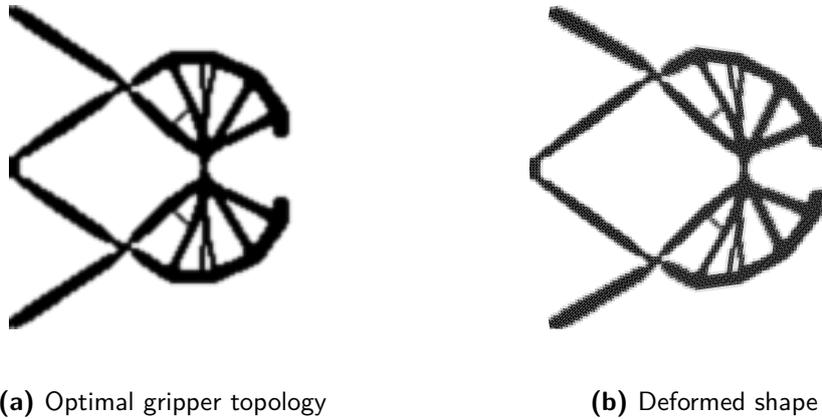


Figure 3-14: Optimal topology and displacement pattern for design problem Figure 3-12b, a horizontal input on the left, a vertical output on the middle of the void region. The gripper inverts a input of $d_{in} = 75.21$ into an output of $d_{out} = 30.48$, which results in a displacement gain of $G_d = 0.81$.

3.5 Conclusions

In chapter 3, a variety of engineering problems are described. The Method of Moving Asymptotes (3.1) seems to be a very effective solution pattern. The computational time is usually somewhat longer, but the final result is more accurate and more importantly, this method is able to solve a wider range of problems, as the OC method is restricted to simple compliance problems. When dealing with predefined regions within the design domain, an option to implement restrictive region (3.2.1) can be very helpful. When solving a physical design problem, the existence of gravity needs to be taken into account, a self-weight implementation (3.2.3) can be used to deal with this.

On the computational side, the continuation method (3.2.4) can be used to gain some speed and accuracy. Varying with different filter techniques (3.2.5) can be helpful to force the optimal solution into a strict black-and-white solution.

Adding a third dimension (3.3) can be used to mimic actual design problems, the computational time however will increase exponentially.

Small compliant mechanisms can be optimized for different objectives. Using different in- and output requirements results in different optimal topology designs. For small displacements only, the displacement field for this elastic material can be plotted on the go, in order to check whether or not the optimal result is working.

All of these described features need to be used for creating an optimal bridge (2.4), which will be the main focus for the next chapter and further research.

Part II

Topology Optimization Extensions: Design of Supports and Loads

Design of Supports

Design of supports has already been touched in subsection 2.4.1, which will be recalled in Figure 4-1. Support design can be used in a variety of domains, especially when the placement of the support is not prescribed. Also, when a part of the support should be fixed, topology optimization can be used to create additional supports within the design domain, in order to optimize towards the prescribed objective.

In section 4.1 the basic fundamentals of support design are described. The bridge example as shown in Figure 4-1 is solved in section 4.2, including a variety of adaptations and possibilities. The integration of supports in existing layouts is described in subsection 4.3.3.

Compliant mechanisms as described before are also in big interest for design of supports, as discussed in section 4.4. Practical applications of support design are shown in section 4.5. The topic design of supports is concluded in section 4.6.

All the described problems in this chapter can be solved using the attached MATLAB codes (B.7, B.8 and C.9).

4.1 Support design formulation

In order to work with this method of support design, a new set of design variables can be introduced. For all the possible support area, springs are attached on the four nodes of the elements within that area, in vertical and horizontal direction. So each element is now supported by eight springs as depicted in Figure 4-2. This new set of support design variables \mathbf{z} can now be used to calculate a new stiffness. Just like the SIMP-method (2.2.1), the spring stiffness matrix K_s can be deduced from the maximum stiffness $K_{s,0}$:

$$K_s = (z_j)^q K_{s,0} \quad (4-1)$$

Where q can be seen as a penalization factor of the new design variables, corresponding to the

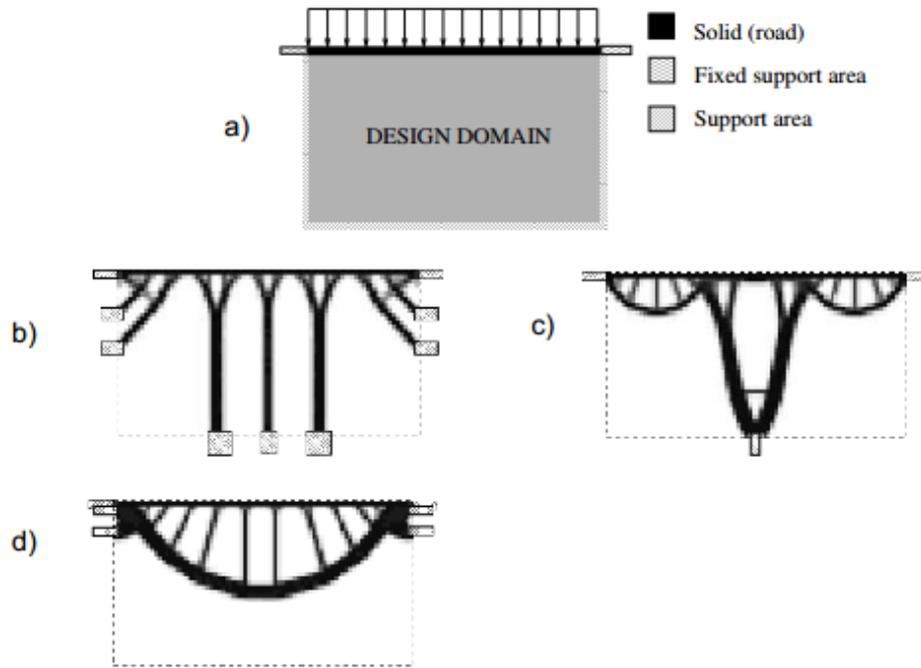


Figure 4-1: Example of design of supports with different support cost functions γ a) Initial design, b) $\gamma = 1$, c) $\gamma = 10$, d) $\gamma = 20$ (Buhl, 2002).

penalty factor p , from the SIMP method. To actually get the total stiffness, the mechanical stiffness and the spring stiffness should be added, which results in a global stiffness matrix \mathbf{K} .

$$\mathbf{K} = \sum_{e=1}^n \rho_e^p \mathbf{K}_e + \sum_{e=1}^n z_e^q \mathbf{K}_{s,e} \quad (4-2)$$

While dealing with support design, a large risk of obtaining local optima can be labeled as a significant issue. A lower bound of the spring design variable \mathbf{z} can be used to overcome this problem. In order to prevent extreme structures, for example creating supports only in one direction, it might be helpful to combine each spring of the element to each other, and thus creating one spring design variable for one element. To work with the support cost function, as depicted in Figure 4-1 a support factor γ can be imposed to the spring design variables, attaching a certain cost to the support of an element. The total amount of weighted support area, should be attached to a certain constraint A . This can be seen and formulated as the material distribution. Just like in (2-5) a simple compliance minimization problem can be

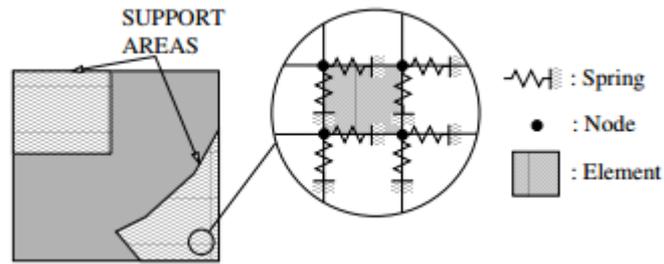


Figure 4-2: Example of support springs, each elements is supported by eight springs (Buhl, 2002).

formulated.

$$\begin{aligned}
 & \min_{\rho_e} \quad \mathbf{f}^T \mathbf{u} \\
 & \text{s.t.} \quad \mathbf{K} \mathbf{u} = \mathbf{f} \\
 & \quad \mathbf{K} = \sum_{e=1}^n \rho_e^p \mathbf{K}_e + \sum_{e=1}^n z_e^q \mathbf{K}_{s,e} \\
 & \quad \sum_{e=1}^n \nu_e \rho_e \leq V \\
 & \quad \sum_{e=1}^n \gamma_e z_e \leq A \\
 & \quad 0 \leq \rho_e \leq 1 \\
 & \quad e = 1, \dots, N
 \end{aligned} \tag{4-3}$$

The associated sensitivity, with respect to the spring design variables can be calculated accordingly:

$$\frac{\partial c}{\partial z_e} = -q(z_e^{q-1}) \mathbf{u}^T \mathbf{K}_{s,e} \mathbf{u} \tag{4-4}$$

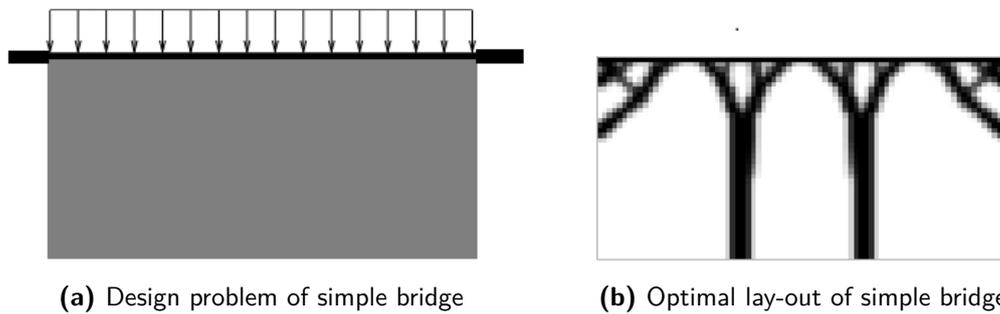


Figure 4-3: A simple bridge example. The bridge is discretized by 80×40 elements. a) design problem, b) Optimal solution.

4.2 The bridge

Consider a simple bridge example, as depicted in Figure 4-3a. For now, let's ignore the design of supports. So in this simple example a discretized bridge of 80 by 40 elements is optimized. A distributed force is exerted on the top of the bridge, which can be seen as a total self-weight of the upper road of the bridge. For this optimization, the upper elements are described as a restrictive region, being fully solid elements. The bridge is fixed at the upper left and upper right node. A volume fraction of 20% is given as constraint. The objective is to minimize the total compliance. The design problem with the associated optimal solution can be found in Figure 4-3b. Since this case represents a practical problem, some notes on the load case should be made. Since the road is dominant among the external loads, the distributed load can be designed as 1 loadcase. In this section, and the following chapters this consideration is implemented. For certain simple cases, there is no need for design of supports. Especially in straightforward theoretical cases this result will be sufficient.

However, when imposing restrictions or variations of costs within the support design domain, it could be a good idea to implement support design. A practical example of the need of support design is described in 4.2.1.

4.2.1 The optimal bridge

A simple bridge design is already described in Figure 4-3. Let's consider a practical example. The bridge in this example is used to make a pavement road across a deep valley. The valley can be seen as two parallel vertical walls, which can be used to create supports. Big and long pillars will be very expensive, especially when the valley becomes deeper. This is a very good example where design of support can be very useful. A volume fraction of 20% of the design domain is a constraint. Only 20% of the support design area (ie. the wall and floor of the valley) can be used to create support. In Figure 4-4b these constraints are built in. As can be seen, the supports are created at the sides and at the bottom; almost the same result as the simple bridge example in Figure 4-3. Now a cost distribution is imposed on the same design. A linear cost distribution is created along the y-axis, this variation is varying linearly from 1 at the top edge, to a certain γ at the bottom edge. In Figure 4-4 some variations of this γ is made. As can be seen, when increasing the upper limit of the support cost γ , the structure tends to support itself towards the pavement road. This is pretty straightforward, since the

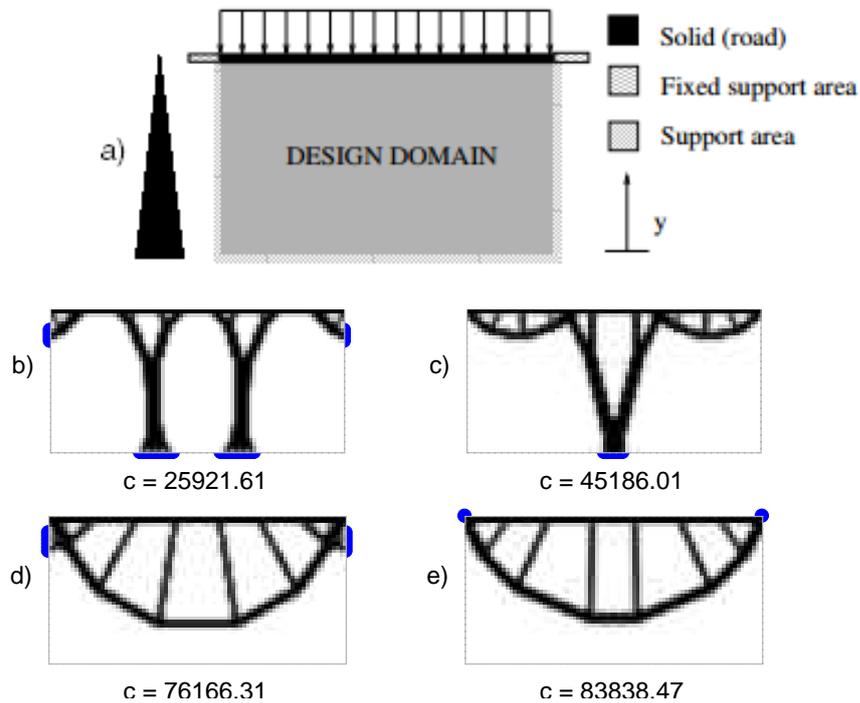


Figure 4-4: The bridge including design of supports with a varying cost of support design. The cost is linearly varying in the vertical direction from 1 (top edge) to γ (bottom edge). The bridge is discretized by 80×40 elements. a) design problem, b) $\gamma = 1$, c) $\gamma = 5$, d) $\gamma = 10$, e) $\gamma = 50$ (Appendix A-14).

cost of placing supports is increasing at the bottom of the design domain. In Figure 4-4e the cost of placing supports became too high to even place supports. The optimal structure in this case is of course less stiff than the original one. As can be seen in Figure 4-4, the result is in line with the result as depicted in Figure 4-1.

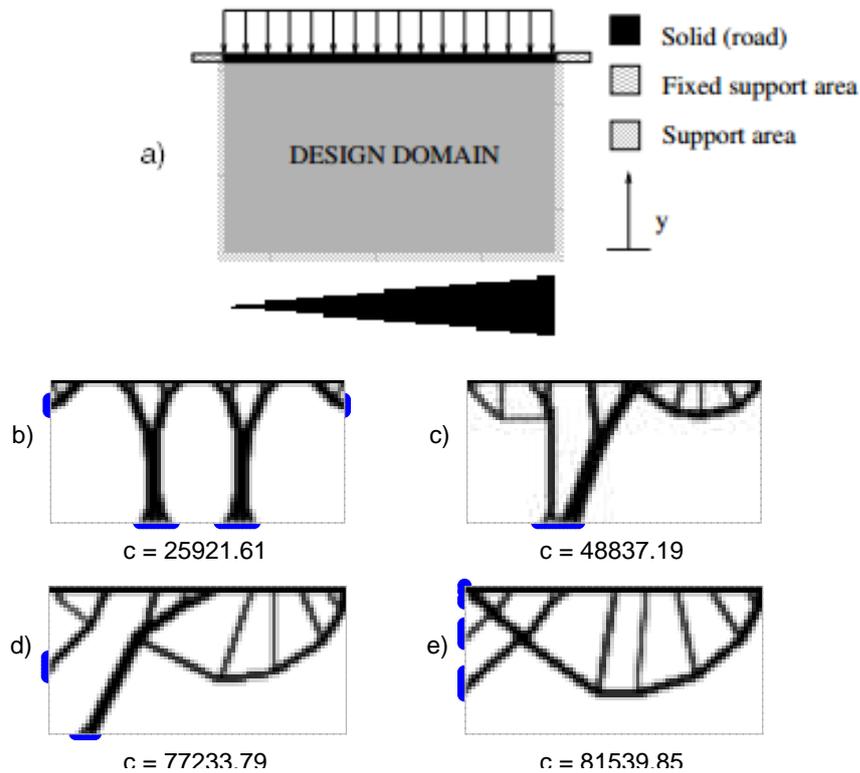


Figure 4-5: The bridge including design of supports with a varying cost of support design. The cost is linearly varying in the horizontal direction from 1 (left) to γ (right), from . The bridge is discretized by 80×40 elements. a) design problem, b) $\gamma = 1$, c) $\gamma = 5$, d) $\gamma = 10$, e) $\gamma = 20$ (Appendix A-15).

4.3 Advanced bridge designs

A variation of cost distribution in the vertical support domain is described in 4.2.1. However, there are some more variations possible. Think about the same bridge example as before. Now, a lake exists in the bottom right edge of the design domain. Creating pillars within the lake is expensive and unwanted. To overcome this design problem, a horizontal cost distribution is imposed on the bridge. Designing a support on the left hand side will cost 1, while the right hand side costs γ . This ratio of costs is varying linearly and results in Figure 4-5. The result is kind of similar to the results of Figure 4-4, which means in this case that the supports are forced to the left side of the design domain. As depicted in Figure 4-6, let's move this lake problem into the middle of the bottom of the valley. By doing this, a cost variation can be introduced varying from 1 to γ to 1, which corresponds from left to middle to right. As can be seen in Figure 4-6, a higher value of γ results in a greater tendency of the structure to move the support to the outer regions. Which in this case means a greater tendency to avoid placing supports within the lake.

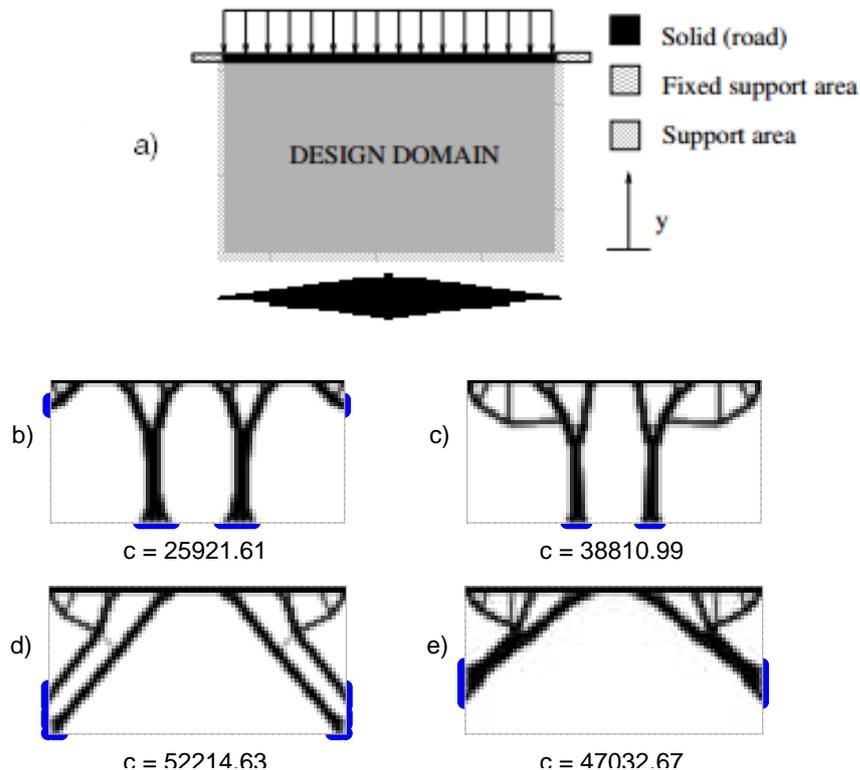


Figure 4-6: The bridge including design of supports with a varying cost of support design. The cost is varying in the horizontal direction from 1 (left) to γ (center) to 1 (right), from . The bridge is discretized by 80×40 elements. a) design problem, b) $\gamma = 1$, c) $\gamma = 3$, d) $\gamma = 5$, e) $\gamma = 10$ (Appendix A-16).

4.3.1 Hanging bridge

In this the design domain of Figure 4-4 is doubled. The force application and the fixed support locations remains the same. By the expansion of the design domain however, the support area is expanded also. As can be seen in Figure 4-7a, the sides can be used for support location, as well as the ground. Since the design domain is doubled, the volume constraint is scaled twice also. In order to compare this result with Figure 4-4, the volume constraint is divided by two. The upper limit of the total weight of the hanging bridge is now the same as the optimal (normal) bridge. The support area is almost doubled as well, compared to the optimal bridge, the support constraint is, however, kept at 20% of the total support area.

As can be seen in Figure 4-7, the support cost ratio γ is varying linearly from the road edge to the bottom edge. The cost from the top edge to the road edge remains 1 in each case. The support cost of the upper half of the design domain is thus very cheap, while the support cost of the lower half can be varied, which can be helpful in case of deep valleys, as already explained in 4.2.1.

An increasing support cost function γ results in a tendency to lift the bridge up. In the same time, the supports at the bottom of the design domain are reduced. As can be seen in Figure 4-7d, the two pillars of Figure 4-7b are replaced with only one pillar. In Figure 4-7e the pillars are completely vanished. The "upper" supports are increased at the same time, in

order to remain a low compliance.

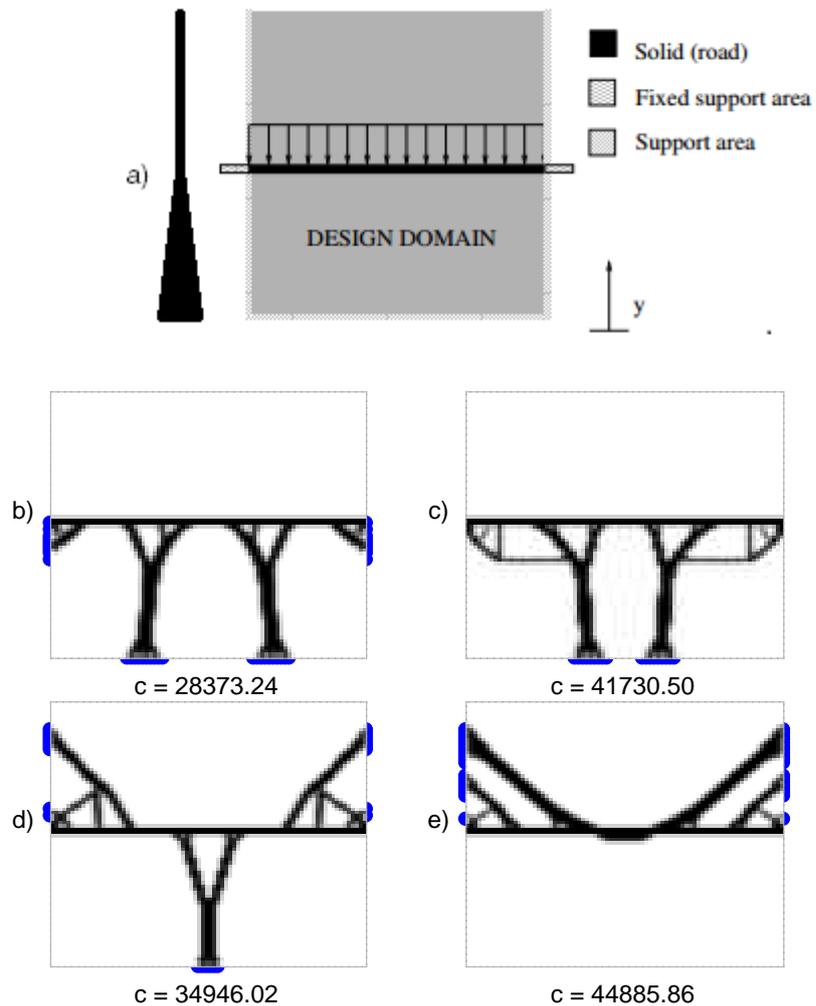


Figure 4-7: The hanging bridge including design of supports with a varying cost of support design. The cost is varying in the vertical direction from 1 (top road edge) to γ (bottom edge). The cost from the top edge to road edge remains 1 in each case. The bridge is discretized by 80×80 elements. a) design problem, b) $\gamma = 1$, c) $\gamma = 2$, d) $\gamma = 4$, e) $\gamma = 6$ (Appendix A-17).

4.3.2 Train tunnel

In 4.2.1 an example of a lake in the middle of the design domain is already explained and used to demonstrate the need of implementing ratio of support cost design. In this case the lake is exchanged by a train tunnel, since trains are also a starting point of bridge design. The main difference between the lake and the train is the possibility of placing supports within that particular area. The train tunnel is designed in Figure 4-8a. In this figure two cases are considered. In case 1 the space for the train can be seen as an open space. This space needs

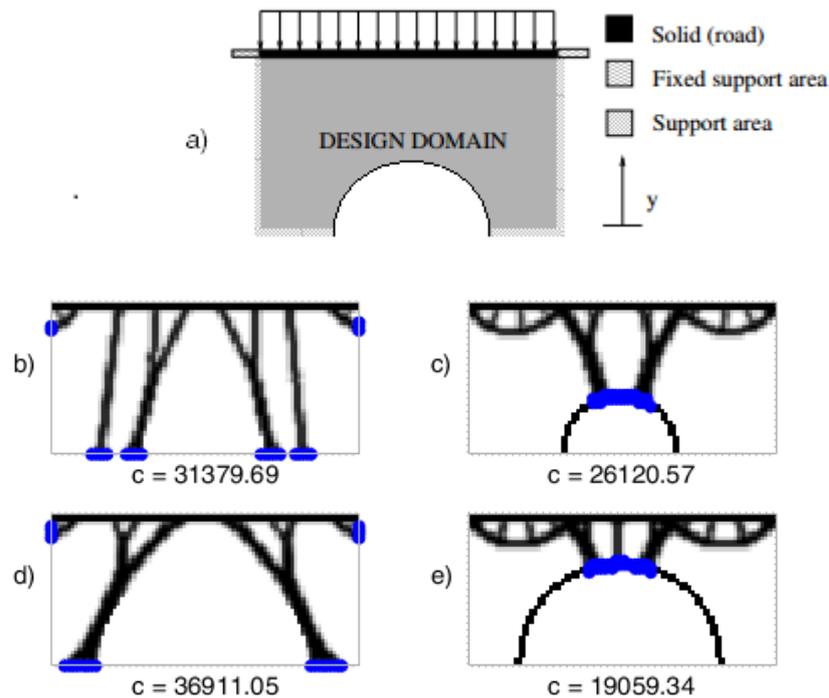
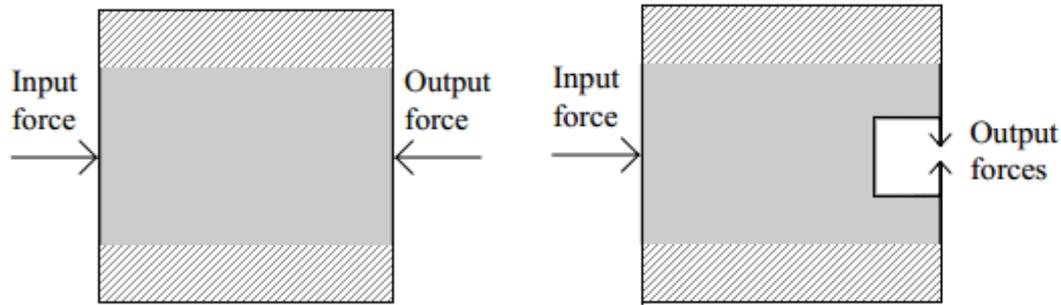


Figure 4-8: The train tunnel shows four different examples of train tunnel designs. The cost is varying in the vertical direction from 1 (top road edge) to γ (bottom edge). The cost from the top edge to road edge remains 1 in each case. The bridge is discretized by 80×40 elements. a) design problem, b) open space, small gap, c) tunnel, small gap, d) open space, big gap, e) tunnel, big gap (Appendix A-18).

to be avoided by the bridge design (Figure 4-8b and Figure 4-8d). In the second case the train space is covered by a train tunnel. Space within the tunnel needs to be void, the outside of the tunnel is solid region and can be used to place supports (Figure 4-8c and Figure 4-8e). Also, the radius of the train tunnel/space is varied. Figure 4-8b and Figure 4-8c shows an example of a small train crossing, while Figure 4-8d and Figure 4-8e demonstrate the optimization process of a big train crossing. As can be seen, the open air train bridge design is slightly weaker compared to the tunnel train bridge design, which is expected before. A bigger open air train space results in a higher compliance, since the topology is forced to the outwards of the design domain. However, a bigger tunnel results in a lower compliance, compared to the small tunnel, since this tunnel on itself provide a lot of additional stiffness to the bridge.

4.3.3 Integration of layout design in supports

In 4.2 the optimal layout of a variety of bridge designs is depicted. The supports are designed by using the expressions as shown in 4.1. These supports can be seen as topology, which is attached to the support area. A practical support however, should be a variation of the topology, since the shape and size of the support is very important, to use the support location of the topology as actual real support. Some research is already done within this field (Zhu



(a) Compliant mechanism amplifier design problem (b) Output force at the right of the design domain

Figure 4-9: Design problem for micro-gripper

and Zhang, 2010), where complex shapes of supports are implemented within the topology optimization problem. In this report this subject is only slightly touched in this section, since the practical implementation of the supports is outside the scope of this research project.

4.4 Design of compliant mechanisms

In 3.4 compliant mechanisms are already described. The topology of the compliant mechanisms is optimized in order to maximize outputs. In Figure 3-11b an example of topology optimization for an amplifier is shown. In Figure 3-13a the optimal topology of a micro-gripper example is shown. For these two compliant mechanisms the support locations are fixed, while the topology is the free variable. In this chapter, design of supports is explained. In this section an example of the use of support design within compliant mechanisms is made and compared to the previous results.

As a recap, the amplifier and micro-gripper design problems are depicted in Figure 4-9, the main difference with the previous design problem is the implementation of the support design area. The support design domain is modeled as an upper and lower band with a total area of one third of the whole design domain. This support design domain can be used to place supports of the structure, with an upper limit of 20 % of the total support design domain.

4.4.1 The optimal amplifier

In Figure 4-9a a positive amplifier is topologically optimized. The result of this optimization is already shown and explained in 3.4.1. The result can be seen in Figure 4-10a. As already explained before, design of support is now included in the design domain, to achieve an even better amplifier, by means of a higher amplification gain (G_d). The design of support formulation as described in 4.1 is used here and implemented in the compliant mechanism code. The result can be found in Figure 4-10b. As can be seen, the design is slightly different, the supports are design within the support design domain. The supports are represented by the blue dots. The input displacement is within the same range, while the output displacement of Figure 4-10b is two times higher, which eventually results in a amplification gain of almost

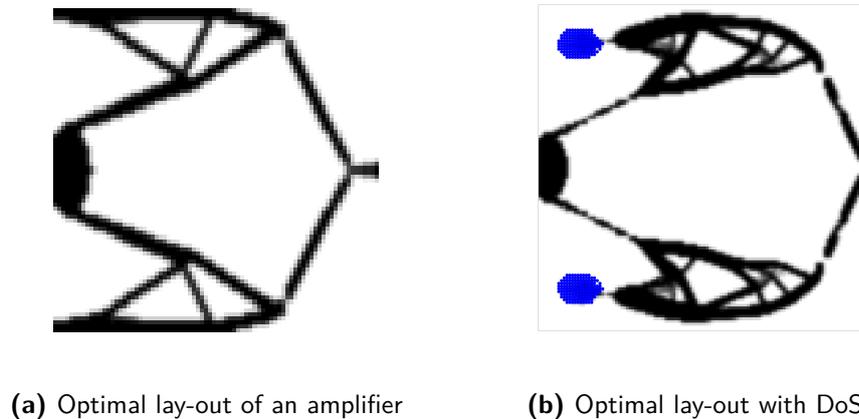


Figure 4-10: Optimal topology and displacement pattern for design problem Figure 4-9a, a horizontal input on the left is amplified to the right. a) the original amplifier converts an input of $d_{in} = 18.42$ into an output of $d_{out} = -40.89$, which results in a mechanical displacement gain of $G_d = -2.22$. b) the amplifier including design of supports converts an input of $d_{in} = 14.15$ into an output of $d_{out} = -83.40$, which results in a mechanical displacement gain of $G_d = -5.89$ (Appendix A-19).

six times the input displacement. From this can be concluded, that implementing design of supports within the design domain results in a higher amplification factor and thus a better result.

4.4.2 The optimal micro-gripper

In 3.4.2 two different micro-grippers are shown. In this section the first one is chosen to optimize with the use of design of supports. The design problem can be found in Figure 4-9b. Again, the upper and lower band represents support area, which can be used for support placement, with an upper limit of 20 %. In Figure 4-11a the optimal gripper is depicted, with fixed supports. In Figure 4-11b this gripper is optimized with the use of design of supports, and can be seen as the solution of the design problem Figure 4-9b. As can be seen, the optimal topology is still a small pliers. The supports however, are pushed from the outer edges from the design domain. This topology results in a total displacement gain of $G_d = 1.97$, which is almost three times higher as the original topology result.

4.5 Application of support design

As can be seen, application of support design can be used for a variety of application, big structures like bridges 4.2, as well as very small structures like micro-grippers 4.4. In this section some words about the application of support design will be said.

Design of supports can be easily used to connect components of a multi-component structure to each other (Chickermane and Gea, 1997). Also, when using the connection locations as joints, the optimal support lay-out can be used to achieve the optimal result (Qian and Ananthasuresh, 2004). Within the assembly of aircraft structures, not only the structure

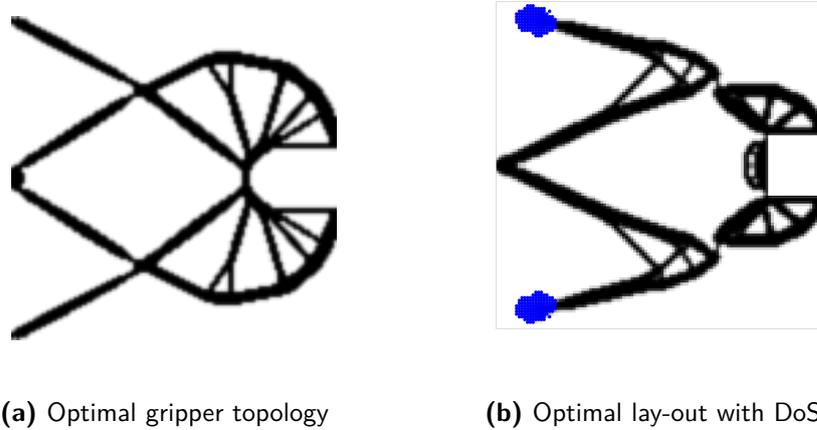


Figure 4-11: Optimal topology and displacement pattern for design problem Figure 4-9b, a horizontal input on the left, a vertical output on the outer right. a) the original gripper inverts a input of $d_{in} = 76.48$ into an output of $d_{out} = 28.06$, which results in a displacement gain of $G_d = 0.73$. b) the gripper including design of supports inverts input of $d_{in} = 52.23$ into an output of $d_{out} = 51.40$, which results in a displacement gain of $G_d = 1.97$ (Appendix A-19).

is optimized for minimizing compliance, but also the shear loads on the fastener joints are controlled. By using design of supports the optimal locations of the fastener joints can be found to achieve maximum overall strength (Zhu et al., 2014).

In the domain of dynamics, maximization of natural frequency can be used to maximize the bandwidth of the structure. Design of supports can be used to achieve the optimal support lay-out to tune dynamic behavior (Jihong and Weihong, 2006). This same technique can be used to achieve certain dynamic behavior, like changing eigenmodes and eigenfrequencies, or maximizing transmission. Another promising development lays within the thermodynamic domain. In this field, heating can be seen as force application, while isolation prevents heating. This prevention is a counteract of the applied force, so a support location. So design of supports can be seen as design of isolation location.

4.5.1 Actuator locations

As already mentioned in 4.5, design of supports can be used in dynamics to tune and tweak frequency response. Support locations are fixed, with no external loads. However, it could be interesting to apply a force on these support locations, which will make the supports act as actuators. By applying this, the optimal placement of actuators can be found, to achieve a certain frequency response. A very promising application within the manufacturing of microchips, where frequency response is very important. The main concern will be to reformulate the support location into an actuator location. Since a support is a fixed location, with no external loads applied, while an actuator location does apply a load to the structure.

In the next section these concerns will be explained even further.

4.6 Conclusions

In chapter 3, a variety of complex problems are described and solved using topology optimization. In this chapter the optimization process is taken a step further and is used to solve problems using variable support designs.

When determining the optimal place of supports, a new set of variables needs to be defined and optimized (4.1). By using this additional set of variables the program determines the optimal solution for both the topology, as well as the support lay-out.

An example of a simple bridge is shown (4.2), this bridge is optimized by using variable supports (4.2.1). This bridge example is extended for some advanced bridge design problems (4.3), where ratio of support cost is varied in different ways. Some more examples of bridge problems are explained, for example a hanging bridge (4.3.1), where supports can be placed on the upper sides of the design domain. Also, when placing a bridge with restricted areas, design of supports can be used to calculate the most ideal configuration (4.3.2).

Some compliant mechanism examples, as already described in this report (3.4) are optimized including design of supports, by placing the support domain within the original, topology design domain. Usage of design of supports is very promising in this area, since the displacement gains can be improved (4.4).

Design of supports can be helpful in a lot of different fields, some practical examples are given (4.5) and the idea of changing fixed supports to actuator locations is slightly touched (4.5.1). The optimal way of actuator placement will be the focus for the next chapter and will be used further on this research.

Design of Actuator Placement

In chapter 4 several cases of design of supports are explained. In this chapter the philosophy of extending the optimization problems is taken a step further. Instead of designing the best support locations, this chapter aims to design the optimal force locations. Design of actuator placement can also be used in combination with topology optimization, for example when volume constraints are involved.

In section 5.1 the basics of actuator placement are described. The need of sensitivity checking is explained in subsection 5.1.3. A simple cantilever beam is used to show the design of force distribution in section 5.2. Some advanced applications for actuator placement are shown in section 5.3.

Up to here, the topology of the previous examples remained fixed. An implementation of topology optimization is explained in section 5.4. Practical applications of actuator design are shown in section 5.5. Section 5.6 concludes this chapter.

The implementation and working MATLAB codes are attached (B.9, B.10, C.10 and C.11) and can be used to reproduce the problems of this chapter.

5.1 Actuator design formulation

This chapter has some similarities with respect to 4, where supports can be placed within a certain support design area, in order to achieve the best objective. In chapter 4 the support design is thus considered as design variable. In this chapter however, the support area remains fixed. The applied force is now considered as a free design variable.

There are some similarities between design of support and design of actuator placement. However, the behavior of both design variables are quite different, a fixed support cannot generate force, while an actuator does generate an active force.

This chapter will mainly focus on minimizing vertical displacement. In order to compare these results, in section 5.2 a simple cantilever beam will be optimized towards minimal compliance. Since the previous chapters mostly rely on compliance minimization, this knowledge can be used to create and solve a simple cantilever beam problem.

The minimal compliance problem is solved the same way using the SIMP method, as described in 2.2.1. This method is solved with a different continuation method for the penalty than described before in 3.2.4. More on this new approach can be found in subsection 5.1.2. The same approach will be used further in this chapter.

Another big difference, with respect to chapter 4 is the constraint function. When topology is not included, the constraint on the volume can be dropped. The supports remains fixed, so this constraint will also drop out. There is however, a new constraint function introduced on the force. Since in all cases the compliance and displacement will be minimized, the optimizer will tend to use as little force as possible. A minimum force should be introduced, in order to force the optimizer to use this minimum of actuator force (f_{min}). This is opposed to the previously used constraints on the volume and supports, where the optimizer wants to use as much volume and/or supports as possible.

In order to minimize the compliance by varying the actuator placement f_i , a minimization problem can be formulated.

$$\begin{aligned}
 \min_{f_i} \quad & \mathbf{f}^T \mathbf{u} \\
 \text{s.t.} \quad & \mathbf{K} \mathbf{u} = \mathbf{f} \\
 & \mathbf{K} = \sum_{e=1}^n \rho_e^p \mathbf{K}_e \\
 & \sum_{i=1}^n f_i \leq f_{min} \\
 & -1 \leq f_i \leq 0 \\
 & i = 1, \dots, N_i
 \end{aligned} \tag{5-1}$$

Where N_i reflects the number of nodes. The associated sensitivities can then be calculated, by differentiating the objective to the design variables. In this chapter, the force can be varied from $f_i = -1$ to $f_i = 0$, since the actuators are pointed downwards. In the upcoming figures, the mean absolute displacement is depicted in each figure.

5.1.1 Sensitivity selection

The same adjoint approach, as described in 2.2.3 will be used to evaluate the associated sensitivities. In this section, the sensitivity approach is used to calculate the sensitivity of the compliance problem, towards the actuator placement. This is done to make a good comparison with the previously calculated sensitivities.

Again, to prevent the calculation of the derivatives of the displacement explicitly, an adjoint method is used to achieve the correct sensitivity.

The minimum compliance problem can be rewritten by adding a zero function, including the Lagrange multiplier λ .

$$c(f_i) = \mathbf{f}^T \mathbf{u} + \lambda^T (\mathbf{K}\mathbf{u} - \mathbf{f}) \quad (5-2)$$

Now the corresponding sensitivity can be calculated by derivating to the design variable f_i .

$$\frac{\partial c}{\partial f_i} = (\mathbf{f}^T + \lambda^T \mathbf{K}) \frac{\partial \mathbf{u}}{\partial f_i} + (\mathbf{u}^T - \lambda^T) \frac{\partial \mathbf{f}}{\partial f_i} \quad (5-3)$$

Where $\lambda = -\mathbf{K}\mathbf{f}$, which on his turn is equal to $\lambda = -\mathbf{u}$, due to the equilibrium $\mathbf{K}\mathbf{u} = \mathbf{f}$. This will eventually lead to the sensitivity

$$\frac{\partial c}{\partial f_i} = 2\mathbf{u}^T \frac{\partial \mathbf{f}}{\partial f_i} \quad (5-4)$$

Where $\frac{\partial \mathbf{f}}{\partial f_i}$ can be seen as a selection vector of the participating force. This can be simply calculated by a column vector which yields a zero for non-participating force, which corresponds to a node that cannot design a force; and a value one for the nodes that corresponds to the actuator design area.

5.1.2 Arching continuation approach

The continuation method is a very powerful approach to prevent the optimizer getting local optima (3.2.4). There is however a big consideration for applying this continuation method in the force design domain. Since density can typically vary from zero to one, either void or solid a power factor will result in a tendency to create void or solid regimes. Since the optimizer wants to create a lot of density, a positive penalty factor $p > 1$ can be used to prevent the optimizer from creating gray regions.

When dealing with this force design variable in combination with minimizing compliance or displacements, the best result will be no force, the optimizer will thus remove as much force as possible. A penalty factor $p > 1$ will help the optimizer with this process, which is not desirable. A positive penalty factor $p < 1$ will send the optimizer towards the biggest value of the design variable.

When applying a power factor in this case, where force can vary from -1 to 0 , there is a problem to overcome. When factorizing a negative value of \mathbf{f} with a power factor $p < 1$, it will result in imaginary values, which is not desirable in this case. A simple way to overcome this would be to penalize the absolute value, this absolute values however, cannot be differentiated. A solution is using a new way of continuation, which is labeled as Arching continuation approach. This approach can deal with negative and positive input values and will give real penalized outputs. This Arching continuation approach can produce a penalized value by

$$\mathbf{f}_p = \frac{\arctan(\alpha \mathbf{f})}{\arctan(\alpha)} \quad (5-5)$$

The numerator will penalize the function, while the denominator creates a normalization. The variable α can be chosen, to increase or decrease the slope of the arches. A visual representation of this new continuation method can be seen in Figure A-12. As can be seen,

an increase of α will result in a higher slope, so a more aggressive penalization. The best way to use this Arching continuation is to exponentially increase α . The optimizations in this chapter are made using a starting value of $\alpha = 0.5$, which is then exponentially increased by a factor 1.06 each iteration, until the final value of α is achieved. $\alpha = 5$ seems to be a good number for force penalization. These values are found to somewhat mimic the continuation method as described before (3.2.4). This approach can also be used, when a design variable can vary from positive to negative, for example in compliant mechanisms. Also, the differential of this Arching continuation approach can be easily derived and used in further sensitivity analysis.

5.1.3 Finite difference method

Up to the design of actuator placement, sensitivities seem to be pretty straight-forward. In this section however, the sensitivity analysis becomes a bit more challenging. In order to deal with sensitivities with respect to different physical quantities, it is a good idea to check whether or not the applied sensitivities are well calculated and derived. One way to check this is using the Finite difference method. This is done by introducing a small perturbation h , which can typically vary between $h = 10^{-2}$ to $h = 10^{-6}$.

The function value f is used as starting point. The sensitivity can now be calculated. At the design variable point a , where the sensitivity yields the maximum value, a small perturbation h is added to this design variable point a . The function value f is then calculated again. The difference between these function values f are now subtracted and divided by h , which is basically the slope of the function. This slope can then be compared to the calculated sensitivities. This difference should be very small to confirm a correct sensitivity function. This finite difference method can be found in (5-6).

$$f'(a) \approx \frac{f(a+h) - f(a)}{h} \quad (5-6)$$

Where $h = 10^{-6}$ seems to be a good value. This finite difference can thus easily be used to check the sensitivities of the objective. But it can also be used to check the constraint sensitivity.

5.2 Simple cantilever beam

A simple cantilever beam is used to show the working principle of the design of optimal loading. Recap: the topology remains fixed. By introducing a force design domain and a minimum force value the MMA optimizer (3.1) can now solve the optimal actuator placement. Consider a simple cantilever beam as used before. The force design domain is in this example chosen to be from the down right point to the down middle point, as can be seen in Figure 5-1a. By using the optimization problem as explained in (5-1) and the corresponding sensitivities as described in (5-3) the optimal actuator placement can be found, as shown in Figure 5-1b. This results is in line with the preliminary thoughts. The most optimal place of force placement will be at the nearest point from the fixed support (left-hand side), to minimize the generated moment. The minimum force constraint is active, so the total force equals the minimum force, which is also expected. A minimum amount of force will result in a minimal compliance, or maximal stiffness.

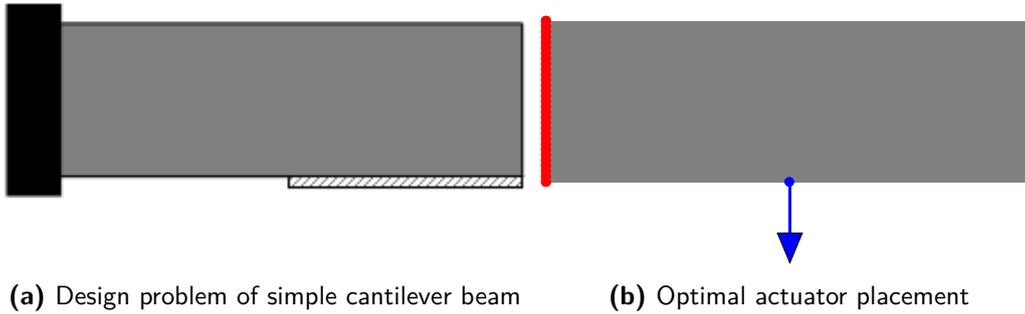


Figure 5-1: A simple cantilever beam example for actuator placement. The red dots indicated the fixed supports, the blue arrow corresponds to the force. The mean vertical displacement in the direction of the force is displayed below each solution. The beam is discretized by 90×30 elements. a) design problem, b) optimal solution.

5.2.1 Minimal displacement

In contrast to a minimal compliance problem, as described in 5.2, the focus will from now on changed to a minimum of vertical displacement. The optimal results should not differ much, with respect to the minimal compliance problem, since the displacement and compliance are correlated to each other. The minimization problem can be formulated again, but now for minimum displacement.

$$\begin{aligned}
 \min_{f_i} \quad & u_a \\
 \text{s.t.} \quad & u_a = \mathbf{L}^T \mathbf{u} \\
 & \mathbf{K} \mathbf{u} = \mathbf{f} \\
 & \mathbf{K} = \sum_{e=1}^n \rho_e^p \mathbf{K}_e \\
 & \sum_{i=1}^n f_i \leq f_{min} \quad -1 \leq f_i \leq 0 \\
 & i = 1, \dots, N_i
 \end{aligned} \tag{5-7}$$

Where \mathbf{L} is the selection vector of the displacement. If only the vertical displacement is involved, which is the case in this section, the selection vector will have the form $\mathbf{L} = [0 \ 1 \ 0 \ \dots \ 1 \ 0 \ 1]^T$. The minimum vertical displacement can be rewritten including an adjoint function.

$$u_a(f_i) = \mathbf{L}^T \mathbf{u} + \boldsymbol{\lambda}^T (\mathbf{K} \mathbf{u} - \mathbf{f}) \tag{5-8}$$

The sensitivity analysis of this minimization function is in line with the previously analysis (5-3), but is a bit more complicated.

$$\frac{\partial u_a}{\partial f_i} = (\mathbf{L}^T + \boldsymbol{\lambda}^T \mathbf{K}) \frac{\partial \mathbf{u}}{\partial f_i} - \boldsymbol{\lambda}^T \frac{\partial \mathbf{f}}{\partial f_i} \tag{5-9}$$

Where $\boldsymbol{\lambda} = -\mathbf{K}^{-1} \mathbf{L}$. In contrast to (5-4) this $\boldsymbol{\lambda}$ will not vanish and need to be calculated each iteration. Therefore the total running time for solving minimal displacement typically

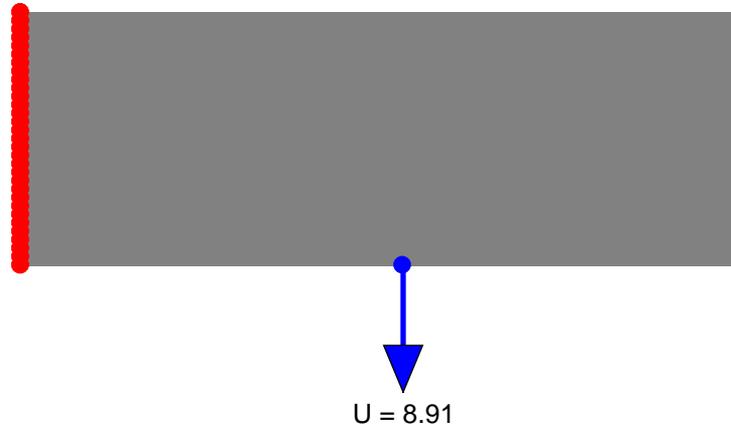


Figure 5-2: Example of minimal vertical displacement including actuator placement.

will be longer than that of solving minimal compliance problems. The final sensitivity can thus be rewritten as

$$\frac{\partial u_a}{\partial f_i} = -\lambda^T \frac{\partial \mathbf{f}}{\partial f_i} \quad (5-10)$$

As can be seen, the result of the actuator placement for minimal displacement (Figure 5-2) does not differ from the result for minimal compliance (Figure 5-1b). The displayed U reflects the mean displacement in the direction of the force.

5.3 Advanced applications

Design of actuator placement can be used to minimize a simple cantilever beam, while respecting a minimum force. This minimum force should be provided by an actuator. However, what would happen when the minimum applied force cannot be provided by a single actuator, due to its limit of power? An additional constraint should be included in the optimization, to deal with this problem. In Figure 5-3a a result of this problem statement is provided. In this problem, the same objective and design variables are provided, as described in 5.2.1, but a single actuator can only provide one fifth of the total minimal force. As can be seen, the best way to place the actuators is to place them five in a line, at the most left point of the actuator design area. This should be okay, since the actuators together want to minimize the moment exerted on the cantilever.

5.3.1 Maximal displacement

Up to here, only minimal compliance and displacement was considered. Most of the time a minimization is the best way to optimize, and most of the time we are looking for a minimum, think of minimum cost, minimal weight, minimal displacement etc. There are some cases arguable where a maximum of displacement is desirable. A simple actuator system within the manufacturing domain is a good example. There we want to maximize displacement with

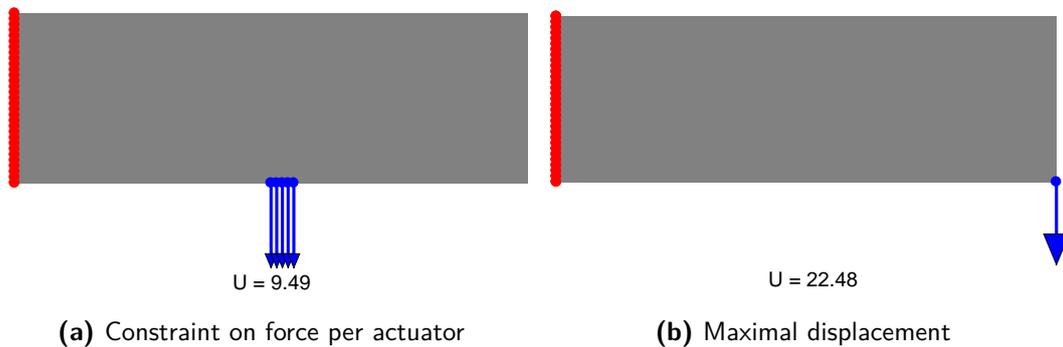


Figure 5-3: Optimal actuator placement of advanced applications for a) minimal displacement, with a constraint on the force per actuator, b) maximal displacement.

a minimum force, additional stiffness demands can be included as constraints. In Figure 5-3b a schematic of maximum displacement of a beam can be found. The force is pointed at the most right point of the design domain. This is in line with the preliminary thoughts, since a large distance between force and fixtures results in a maximum moment acting on the cantilever beam, which on his turn will result in maximum displacement.

5.3.2 Triple fixed beam

Design of Actuator placement is pretty straightforward for a simple cantilever beam, especially when the topology remains fixed. Therefore, a new design problem is considered and solved. Let's consider a triple fixed beam, which can be seen as a bridge structure, which is also completely fixed on the left and right sides. A schematic of this design problem can be seen in Figure 5-4a. As can be seen, the actuator design domain consist one third of the bottom row. The optimal solution for force placement will now be calculated. This is done by using the same objective as used before (5-7) and the same sensitivity analysis (5-10). A minimum force constraint is implemented with a minimum value of $\mathbf{f} = 1$. The result of the optimization can be found in Figure 5-4b. As can be seen, the most optimal solution is two distributed forces (each consist of $\mathbf{f} = 0.5$). This is a correct result, since the force needs to be placed as close as possible to the supports, which is in this case two points.

5.3.3 Minimal area displacement

Up to here, the main focus was to minimize the overall vertical displacement. In this section some words are spend on the ability of optimizing the actuator placement towards minimal displacement in a certain area. This is very useful in manufacturing technology, since most of the time engineers are interested in local effects. To demonstrate the working principle, the same optimization as in 5.3.3 is done, but now with a different selection vector \mathbf{L} , which will only select the vertical displacement of the striped area in Figure 5-4a. This selection vector is also used in the sensitivity analysis. The optimal actuator placement for this solution is depicted in Figure 5-4c. As can be seen, the force is placed at the center of the actuator design domain, which would be probably the worst solution of the regular minimal displacement

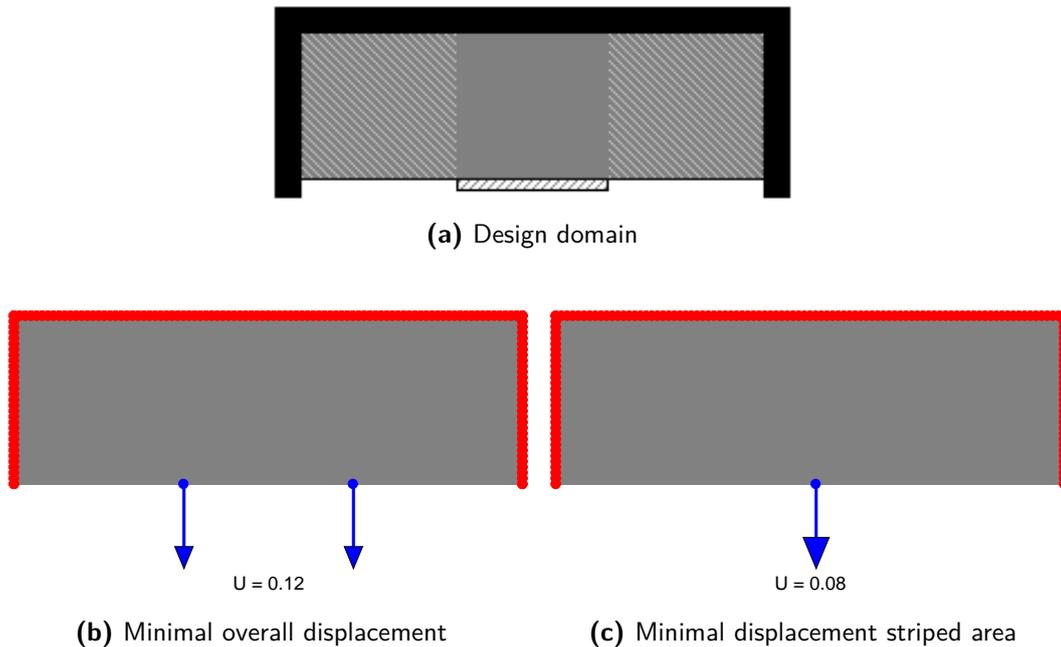


Figure 5-4: Triple Fixed beam, clamped at three sides. Optimal actuator placement of advanced applications for a) design domain, b) minimal displacement all grey area, c) minimal displacement of striped area only. The associated deformed geometry can be found in (Appendix A.6.1).

problem (5.3.2). Of course, the average displacement is somewhat lower ($U = 0.08$ vs $U = 0.12$), which can be explained by the fact that in the overall displacement, the displacements above the actuator application also contribute to the average, while in Figure 5-4c only the striped area is used to calculate the mean value.

5.4 Topology optimization for actuator placement

Up to here, the topology of the beam remains constant, namely completely solid. This was done to verify and demonstrate the working principle of design of actuator placement. It will become much more interesting if the topology is included in the design problem, while the placement of actuators can be optimized at the same time. The same objective holds, but an additional set of design variables is added to the problem. Also, an additional constraint is added to the problem, to limit the volume V that can be used. The optimization problem

can now be described as

$$\begin{aligned}
& \min_{f_i, \rho_e} u_a \\
& \text{s.t.} \quad u_a = \mathbf{L}^T \mathbf{u} \\
& \quad \mathbf{K} \mathbf{u} = \mathbf{f} \\
& \quad \mathbf{K} = \sum_{e=1}^n \rho_e^p \mathbf{K}_e \quad e = 1, \dots, N \\
& \quad \sum_{i=1}^n f_i \leq f_{min} \quad i = 1, \dots, N_i \\
& \quad \sum_{e=1}^n \nu_e \rho_e \leq V \\
& \quad -1 \leq f_i \leq 0
\end{aligned} \tag{5-11}$$

Where i still denotes the node numbers, and e denotes the number of elements. For optimizing this problem, the arching continuation method (5.1.2) is used to penalize the forces, the regular continuation method (3.2.4) is used to penalize the density.

The sensitivities of this problem will not change for the force design variables, but will change for the density design variable. By using the adjoint method again, the sensitivity from the displacement u_a (5-8) to the density variable can be calculated now

$$\frac{\partial u_a}{\partial \rho_e} = (\mathbf{L}^T + \boldsymbol{\lambda}^T \mathbf{K}) \frac{\partial \mathbf{u}}{\partial \rho_e} + \boldsymbol{\lambda}^T \frac{\partial \mathbf{K}}{\partial \rho_e} \tag{5-12}$$

By choosing again $\boldsymbol{\lambda} = -\mathbf{K}^{-1} \mathbf{L}$, the $\frac{\partial u_a}{\partial \rho_e}$ does not have to be calculated explicitly

$$\frac{\partial u_a}{\partial \rho_e} = \boldsymbol{\lambda}^T \frac{\partial \mathbf{K}}{\partial \rho_e} \mathbf{u} \tag{5-13}$$

Now formulate $\frac{\partial \mathbf{K}}{\partial \rho_e}$ as the derivative of the penalty-termed stiffness as derived in (2-5) the final sensitivity can be made as

$$\frac{\partial u_a}{\partial \rho_e} = p(\rho_e^{p-1}) \boldsymbol{\lambda}^T \mathbf{K}_e \mathbf{u} \tag{5-14}$$

The tolerance is updated to a summation of the difference of the force and density, with respect to the last iteration. Due to this tolerance update, and the addition of another set of design variables, the computational time will rise exponential.

The optimal result for a minimization of the overall displacement, so the selection vector will have the form $\mathbf{L} = [1 \ 1 \ 1 \ \dots \ 1 \ 1 \ 1]^T$, can be seen in Figure 5-5a. The result can be labeled as quite remarkable. A deeper investigation can explain this weird behavior. By minimizing the displacement in the direction of the force, the optimizer also wants to maximize in the opposite direction. That's exactly what happens in this problem. The optimizer discovers a maximization of the opposite direction can be achieved by adding force. However, since the force is pointed downwards, displacement in the opposite direction is not expected. This result is probably caused by a numerical issue of the optimizer linked by the FEM method.

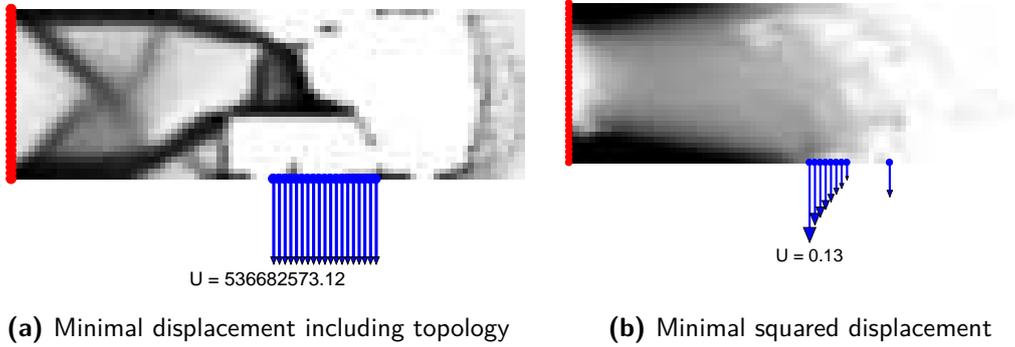


Figure 5-5: Optimal actuator placement including topology optimization for a) minimal displacement, b) minimal squared displacement. The associated displacement plots can be found in (Appendix A.6.2) and (Appendix A.6.4).

5.4.1 Displacement consideration

One way to overcome the problem of 5.4 is by simply minimizing the squared displacement. This will skip the tendency to maximize the opposite direction. It will change the minimization problem from (5-8) into a new formulation:

$$u_a(f_i, \rho_e) = (\mathbf{L}^T \mathbf{u})^2 + \boldsymbol{\lambda}^T (\mathbf{K} \mathbf{u} - \mathbf{f}) \quad (5-15)$$

With the corresponding sensitivities:

$$\frac{\partial u_a}{\partial f_i} = (2\mathbf{L}^T \mathbf{u} \mathbf{L}^T + \boldsymbol{\lambda}^T \mathbf{K}) \frac{\partial \mathbf{u}}{\partial f_i} - \boldsymbol{\lambda}^T \frac{\partial \mathbf{f}}{\partial f_i} \quad (5-16)$$

$$\frac{\partial u_a}{\partial \rho_e} = (2\mathbf{L}^T \mathbf{u} \mathbf{L}^T + \boldsymbol{\lambda}^T \mathbf{K}) \frac{\partial \mathbf{u}}{\partial \rho_e} + \boldsymbol{\lambda}^T \frac{\partial \mathbf{K}}{\partial \rho_e} \mathbf{u} \quad (5-17)$$

Where $\boldsymbol{\lambda} = -2\mathbf{K}^{-1} \mathbf{L} \mathbf{u}^T \mathbf{L}$ to remove the $\partial \mathbf{u}$ terms.

The corresponding result of this optimization can be seen in Figure 5-5b. The result is obviously a lot better than Figure 5-5a. We see an expected behavior, namely, the force is placed most left of the force design domain. The topology is then be used to counteract this force and make a stiff structure. Even after a maximum number of iterations, still a gray pattern remains for the topology. It seems like the optimizer simply does not want to create a black-and-white pattern. This behavior is in collaboration with the distribution of the force. A trade-off between counteracting the force by placing material and minimizing the moment exerted on the beam is made. This trade-off seems to be difficult to solve. However, as little force as possible is used, which makes the force constraint f_{min} an active constraint, which is in line with the theory. The figures does not converge to black and white regimes, in the next section a possible solution is explained.

5.4.2 Compliance constraint

The result in Figure 5-5b is quite good, but not perfect at all. It seems like the optimizer creates little material near to the force, and the structure is a little leaned over. Perhaps the

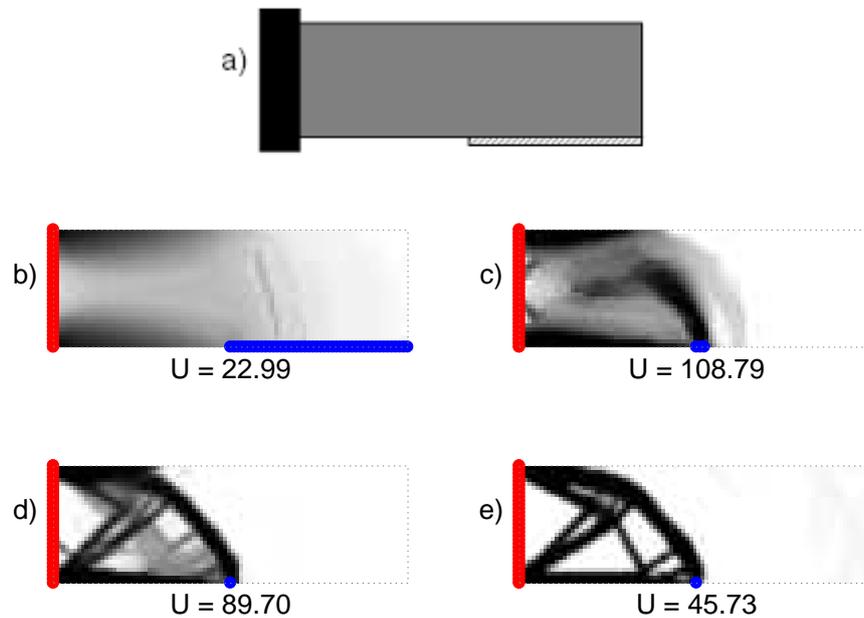


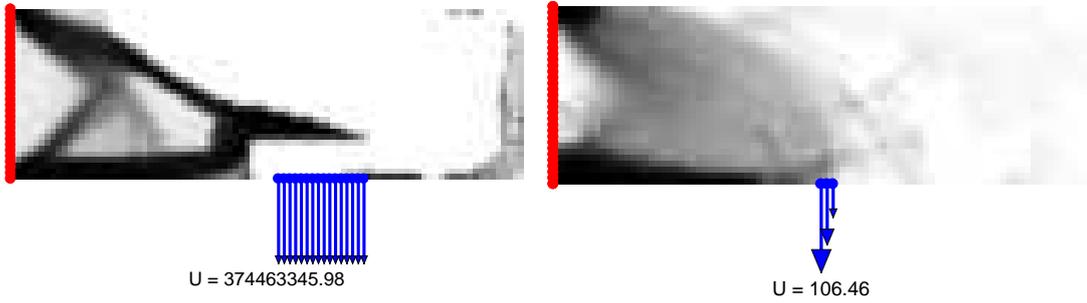
Figure 5-6: Design evolution history of the optimization process for minimal displacement using actuator placement and topology optimization. The beam is discretized by 90×30 elements. a) design problem, b) 5% of total iteration steps, c) 25% of total iteration steps, d) 50% of total iteration steps, e) final solution. The associated mean displacements are shown under each figure. The associated displacement plots can be found in (Appendix A.6.2) and (Appendix A.6.4).

FEM method could transport the external force through void regions to the structure. This results that a certain area, very close to the force, is displaced very much. This force however, is absorbed by the void regions, so the other solid regions are little effected by displacements due to the external force. The objective isn't determined as a minimal displacement for a local area near the force, but the objective is to minimize the overall displacement. When little elements are displaced very much, while the rest of the element displaces only a little, the overall displacement could be labeled as relatively low.

One way to overcome this problem is by adding another constraint function. It would be nice to implement within the optimization problem a compliance constraint ($c = \mathbf{f}^T \mathbf{u}$), which should not be too high. By using an arbitrary upper limit value of total compliance, we can prevent the optimization process to let the force pass through less dense regions.

During optimization it seems like the force and density constraints need to be upscaled, to make these two constraints more important than the compliance constraint. This is done due to the fact that the main constraints for this problem are the force and density constraints, while the compliance constraint is just added to prevent the optimizer creating non-physical solutions.

A design evolution history of this optimization process is depicted in Figure 5-6. As can be seen, the optimization process seems very nice and decent. The force is gradually placed to the left hand side of the actuator design domain, the topology is gradually optimized in a black-and-white structure, which is in line with the previous compliance problems. The additional constraint does result in a longer computational time, however.



(a) Minimal displacement including topology (b) Minimal squared displacement including density and density dependency

Figure 5-7: Optimal actuator placement including topology optimization with density dependency for a) minimal displacement, b) minimal squared displacement. The associated displacement plots can be found in (Appendix A.6.3) and (Appendix A.6.5).

5.4.3 Objective refinement

A pretty nice result of the design evolution history can be found in Figure 5-6. It seems like the result is very optimal. There is however one issue that can be improved. The calculation time takes very long. This can probably be ascribed to the formulation of the objective. In (5-15) the objective is described as minimizing the absolute displacement for the whole design domain. However, since the topology is included, the main point of interest is not the displacements of the design domain, but mostly the displacement field of the constructed structure itself. To refine the prescribed objective, it can be very interesting to include the density distribution in the objective. A simple multiplication of the topology distribution by the associated displacement field will result in a new objective. This objective will tend to minimize the displacement of the solid regions. So basically, it will minimize the actual displacements of constructed area. By applying this refinement, a speed improvement can be made. Since the optimizer is no longer interested in minimization of void regions, the computational load can be used for solid regions, which eventually will lead to shorter computation time. The updated minimization problem can now be re-formulated as:

$$u_a(f_i, \rho_e) = (\mathbf{L}^T \mathbf{u}_x)^2 + \boldsymbol{\lambda}^T (\mathbf{K} \mathbf{u} - \mathbf{f}) \quad (5-18)$$

Where \mathbf{u}_x can be seen as a Hadamard product of the node displacement and the node density. Since density is always element based (ρ_e), and node density does not have any physical interpretation, a transformation from the element density should be made into the virtual node density value ρ_n . This Hadamard product can be written as:

$$\mathbf{u}_x = \mathbf{u} \odot \boldsymbol{\rho}_n \quad (5-19)$$

By using this formulation, the corresponding sensitivities can be calculated as:

$$\frac{\partial u_a}{\partial f_i} = (2\mathbf{L}^T \mathbf{u}_x \mathbf{L}^T + \boldsymbol{\lambda}^T \mathbf{K}_x) \frac{\partial \mathbf{u}_x}{\partial f_i} - \boldsymbol{\lambda}^T \frac{\partial \mathbf{f}}{\partial f_i} \quad (5-20)$$

$$\frac{\partial u_a}{\partial \rho_e} = (2\mathbf{L}^T \mathbf{u}_x \mathbf{L}^T + \boldsymbol{\lambda}^T \mathbf{K}_x) \frac{\partial \mathbf{u}_x}{\partial \rho_e} + \boldsymbol{\lambda}^T \frac{\partial \mathbf{K}}{\partial \rho_e} \mathbf{u} \quad (5-21)$$

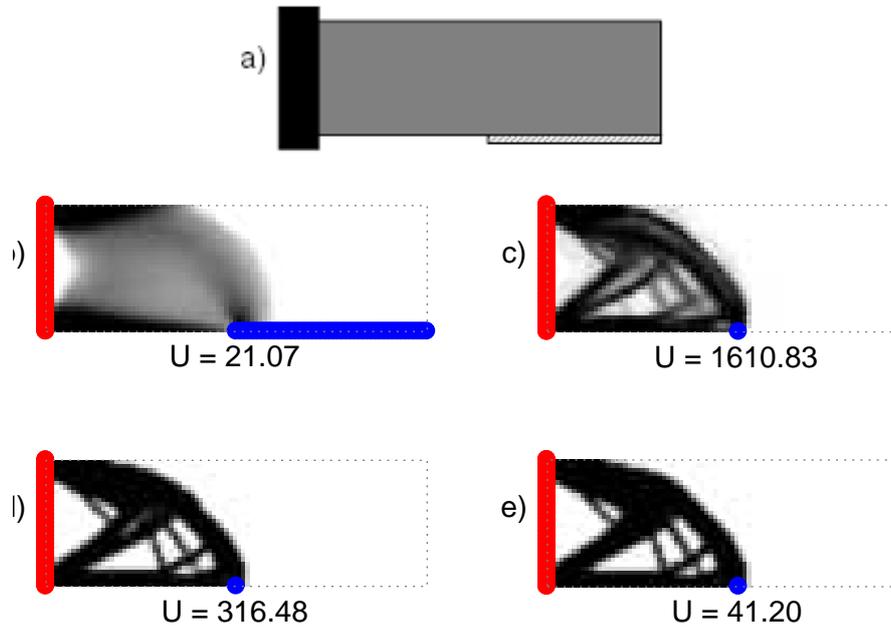


Figure 5-8: Design evolution history of the optimization process for minimal displacement using actuator placement, topology optimization and density dependency. The beam is discretized by 90×30 elements. a) design problem, b) 5% of total iteration steps, c) 25% of total iteration steps, d) 50% of total iteration steps, e) final solution. The associated mean displacements are shown under each figure. The associated displacement plots can be found in (Appendix A.6.3) and (Appendix A.6.5).

Where $\lambda = -2\mathbf{K}^{-1}\mathbf{L}\mathbf{u}_x^T\mathbf{L} \odot \rho_n$ to remove the $\partial\mathbf{u}_x$ terms, and $\mathbf{K}_x = \mathbf{K} \odot \frac{1}{\rho_n}$. The optimal results are depicted in Figure 5-7. The previous optimal results of Figure 5-7 are now updated, including the new objective from (5-18). In order to compare the difference between both formulations, the same lay-out is used in Figure 5-7, as in Figure 5-5.

As can be seen, the results have different solutions. There is less gray area and the calculation time is improved. Because of the implementation of the density dependency, the optimizer neglects void regions, which eventually leads to shorter computational time and better design of actuator placement. However, both results are still not optimal. One way to overcome this problem is to implement the compliance constraint, as explained in 5.4.2. The optimal result of this optimization problem, including this compliance constraint and including density dependency is depicted in Figure 5-8. As can be seen, the optimizer seems to reach its final stage much faster, than without using density dependency (Figure 5-6). Also, when looking at the deformed geometry, it can be concluded that the optimizer priorities minimal displacement of the solid regions. Therefore, the solution of Figure 5-8 differs from the previous solution and is better physically interpretable. The overall mean displacement is also improved by 10%.

5.5 Application of actuator placement

Design of actuator placement can be used in a variety of domains. Simple cantilever problems can be optimized using the combination of actuator design (Begg et al., 1997) and topology optimization. Especially within the manufacturing domain, actuator placement can be very promising. An optimal actuator layout can be used to minimize internal deformations, which contributes to a more reliable system. Besides to minimizing displacements and minimizing compliance, it can also be used to achieve dynamic performance using a harmonic response (Barboni et al., 2000). For example the frequency spectrum can be tuned using an actuator optimization model as mechanical filter, to ensure that certain mode shapes are not excited whereas other are. Altering eigenmodes can also be done by using actuator placement, in combination with topology optimization, for example to extend the bandwidth. These optimizations can be taken a step further by including control of these actuators (Alves da Silveira et al., 2015). By including this control functionality, it could be very promising to use optimal actuator placement in combination with piezoelectric materials (Foutsitzi et al., 2013). In this field, it should be possible to tune certain dynamic behavior of the material by optimizing the applied voltage to the piezoelectric elements.

Also in the thermal domain actuator placement can be used. In this field, heating can be seen as force application. The perfect heat locations can be found by using actuator placement, in order to maximize the thermal performance of a certain model (Sheng and Kapania, 2001). In the next chapter a complete case study will be made, which could be very promising in the nearby future. A wafer stage will be optimized to enhance its dynamical performance

5.6 Conclusions

In chapter 4, some words are spend on the design of supports. In this chapter the focus is changed to variable force applications. Optimization can be used to find the perfect actuator placement, in order to achieve an objective.

When determining the optimal place of actuators, the force is used as a set of design variables (5.1). A penalization problem comes up when using negative forces, or forces that are pointing downwards. A way to overcome this, is by using the new introduced Arching Continuation Method (5.1.2), for penalizing negative and positive forces.

A simple solid cantilever beam can be optimized for actuator placement, by minimizing compliance (5.2) or minimizing displacement (5.2.1). Also, design of actuator placement can be used to optimize a variety of advanced applications (5.3), by tweaking the objective and associated sensitivities.

Topology optimization can also be added to the problem. The placement of actuators will cooperate with the topology in order to achieve the best objective (5.4). Some changes should be made to the objective, however (5.4.1), to prevent the optimizer from searching for unwanted optima. A third constraint is sometimes needed, to force the structure being physically interpretable (5.4.2). It can also be helpful to include density dependency into the objective (5.4.3) for even better interpretable results.

Design of actuator locations can be promising in a lot of different fields, from mechanical to thermal problems (5.5). In the next chapter a case study is dedicated to this current chapter, where a wafer stage will be optimized, in order to maximize its dynamical behavior.

Part III

Dynamic Topology Optimization

Chapter 6

Case Study: Wafer Stage

In chapter 5 the design of actuator placement is studied for static problems. In this chapter this approach is taken further by considering dynamics. This placement of dynamic actuator force can also be used in combination with topology optimization, for example to reduce the applied dynamic load.

In section 6.1 an introduction of a wafer stage is made. Dynamics are introduced in section 6.2, where several dynamical aspects are investigated. These phenomena are demonstrated using three different examples. In section 6.3 the design of actuators is investigated to achieve better (dynamical) performance.

Up to here, the design domain is considered as a complete solid region. In section 6.4 topology optimization is included besides the design of actuators in a dynamical spectrum. The solid case examples are now all solved with topology optimization introduced. In section 6.5 a final, optimal solution is given, by making multiple sides of the domain available for actuator design.

In section 6.6 a nice lateral 2D case is made, with a nice 3D graphical representation. Section 6.7 concludes this chapter.

6.1 Case introduction

This section is dedicated to a dynamic actuation of a structure. For example a wafer stage. This stage is used as a driver for a wafer. This wafer is a thin slice of a semiconductor, for example a thin plate of high pure crystalline silicon, which is used in electronics for the

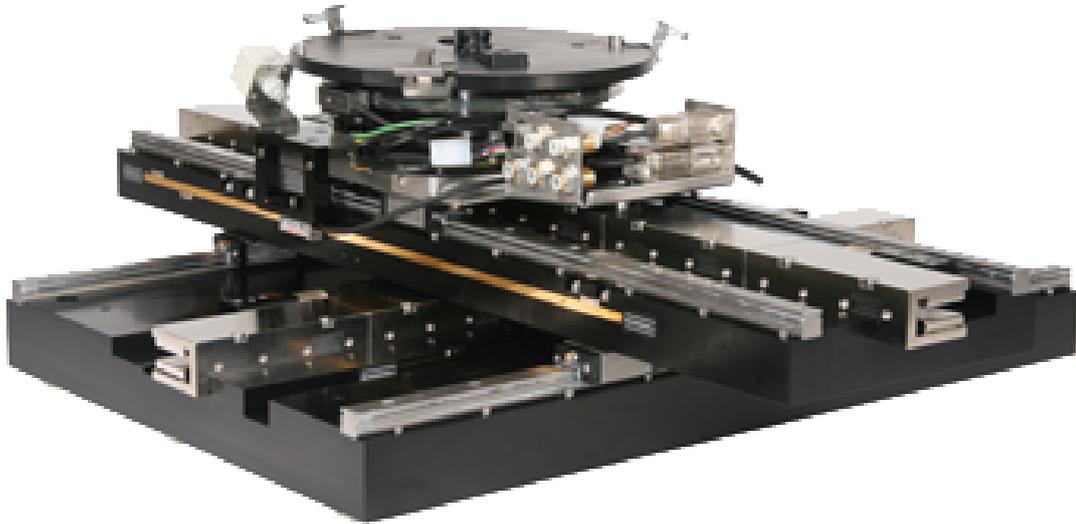


Figure 6-1: A wafer stage and its surrounding complexity (Reliant Systems Inc., 2017)

manufacturing of integrated circuits, as seen in electronic chips. A picture of a wafer stage and its corresponding complexity can be found in Figure 6-1

This wafer stage is actuated and accelerated in order to create a certain motion pattern. This motion pattern can then be used with highly precision positioning to expose the wafer to ultra-violet light, in order to create a certain etching pattern. Extreme precision is required, and any unwanted displacement can result in errors that deteriorate the performance of the electronic circuits. Displacements can arise from small deformations within the wafer stage or from the heat production by the actuator, which results in thermal expansion of the wafer stage. Due to the small size of the integrated circuits, very small deformations in the material can have a big impact. The aim is therefore on a reliable system. The bandwidth and speed of the wafer stage is also a big challenge these days. Time is money, so a faster system will result in more cashflow.

This chapter investigates a new approach to make an improvement on the current wafer stages, by making use of actuator design and topology optimization.

6.2 Dynamics

To understand the way placement of actuators is working in combination with dynamics, let us first have a closer look at the dynamics of this system. The stage should move from left to right, by using an actuator. In this research, we assume the stage to be actuated harmonically, as a model for a cyclic production process. The general dynamic equilibrium equation is given by:

$$\mathbf{K}\mathbf{u} + \mathbf{M}\ddot{\mathbf{u}} = \mathbf{f} \sin(\omega t) \quad (6-1)$$

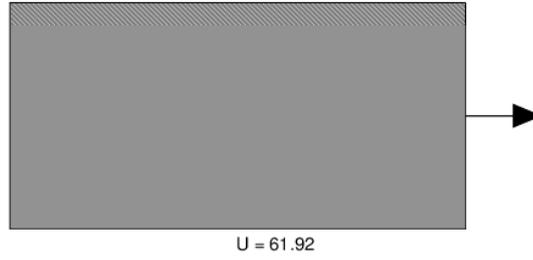


Figure 6-2: Design domain of single force case example. The striped area indicated the objective area. The associated (absolute) vertical displacement of the objective area U is depicted below the figure.

Where \mathbf{M} is the global mass matrix, which is a combination of all elemental mass matrices \mathbf{M}_e . In the model we use a lumped mass matrix. This is an easy and fast way of building up a mass matrix, by simply placing a quarter of the element mass along the eight degrees of freedom of that element.

This dynamic equation of motion is only correct when neglecting damping, which is the case in the considered application. The harmonic excitation will result in a harmonic response. By choosing a harmonic oscillation for the displacement vector \mathbf{u} , the second derivative can be calculated accordingly:

$$\begin{aligned}\mathbf{u} &= \mathbf{u} \sin(\omega t) \\ \dot{\mathbf{u}} &= \omega \mathbf{u} \cos(\omega t) \\ \ddot{\mathbf{u}} &= -\omega^2 \mathbf{u} \sin(\omega t)\end{aligned}\tag{6-2}$$

Substituting these expressions in (6-1) and removing the $\sin(\omega t)$ terms yields:

$$\begin{aligned}\mathbf{K}\mathbf{u} + \mathbf{M}(-\omega^2 \mathbf{u}) &= \mathbf{f} \\ (\mathbf{K} - \omega^2 \mathbf{M})\mathbf{u} &= \mathbf{f}\end{aligned}\tag{6-3}$$

6.2.1 Single force actuator

For a given desired acceleration of the stage mass, the minimum applied force can be calculated accordingly. By making use of Newton's second law ($\mathbf{f} = m \cdot \mathbf{a}$), where m indicates the total mass of the body, the minimum force which should be applied to the body is found. We add this as a constraint to force optimization problem, which will prevent the optimizer from creating a zero force. Without this constraint, the zero force solution is an attractive solution for the optimizer, as it results in minimal (zero) displacements.

To understand the behavior of the dynamic force optimization problem, we deliberately start with a solid stage, so the topology cannot change here. Additionally, a force can be attached to the middle of the right hand side of the body, to let the body move harmonically. A schematic of the first investigation is drawn in Figure 6-2. As can be seen, the stage consist of a solid body and is actuated with one force on the side. The striped area on the top of the design domain represents the area of the objective. In the simplified stage example the objective is to minimize vertical displacement on the top of the wafer stage, where the thin plate of silicon lays. To be able to minimize this value further on in this thesis research, the displacement is squared, as also described in 5.4.1. However, to compare the several cases

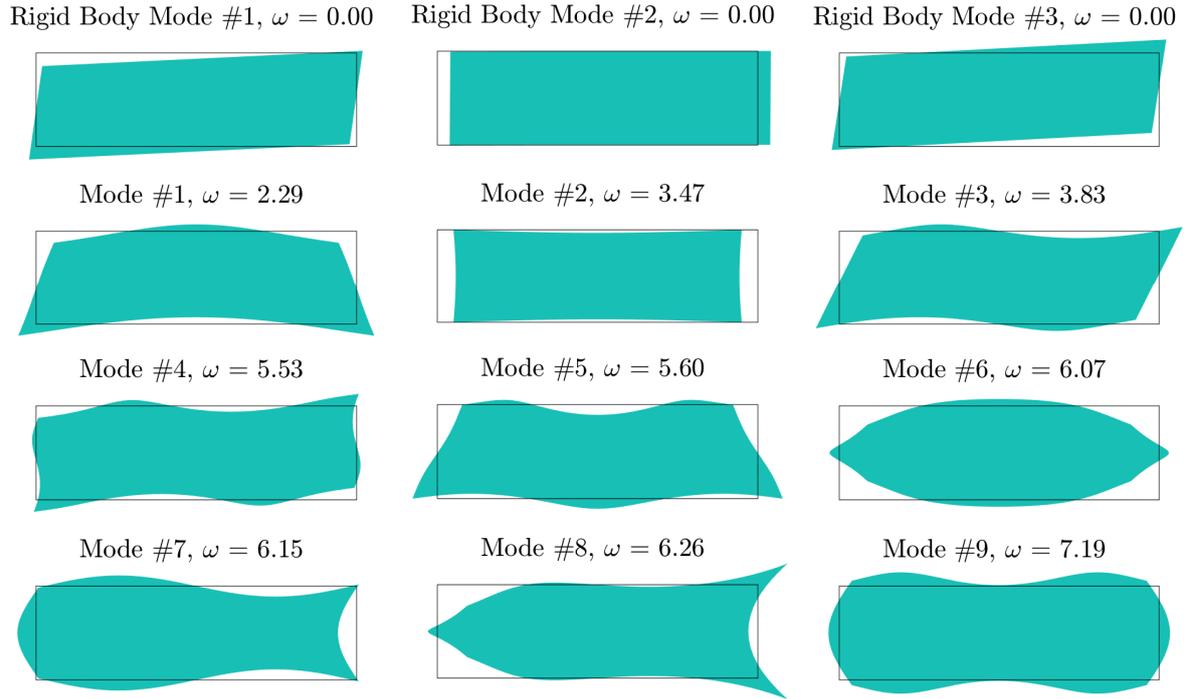


Figure 6-3: Eigenmodes for the first twelve eigenvectors. The upper three eigenmodes are rigid body modes, where zero or very little deformations are involved. The mode shapes in this figure should be combined with Figure 6-4 to get the correct insight in the behavior of each eigenmode.

in this chapter, the sum of the absolute displacements of the top layer, U , is depicted below each figure.

Starting with (6-3), the stiffness matrix \mathbf{K} and mass matrix \mathbf{M} remain constant, since the density, and thus the stiffness and mass distribution, will not change. This means the displacement solution can only be solved using the actuation frequency ω .

6.2.2 Eigenmodes

For a set of frequencies ω the expression $(\mathbf{K} - \omega^2\mathbf{M})$ can result in zero. In this case the displacement solution for a nonzero excitation does not exist. This corresponding frequencies are called eigenvalues, or in this particular case, eigenfrequencies. Each eigenfrequency has its own characteristic displacement field, called its eigenmode. This eigenmode can be seen as a natural vibration of the system, where all parts move together at the same frequency, the eigenfrequency. The corresponding shape of the behavior can be depicted by a so-called mode shape. Additionally, the following equation (6-4) can be solved:

$$(\mathbf{K} - \omega_i^2\mathbf{M})u_i = \mathbf{0} \quad (6-4)$$

This equation will result in a set of eigenvectors u_i and corresponding eigenvalues where this equation holds. This set of solved displacement vectors \mathbf{u} are called eigenvectors and will be displayed as ϕ in the remainder of this thesis.

The mode-shapes can be divided in rigid body modes and structural modes. Rigid body modes

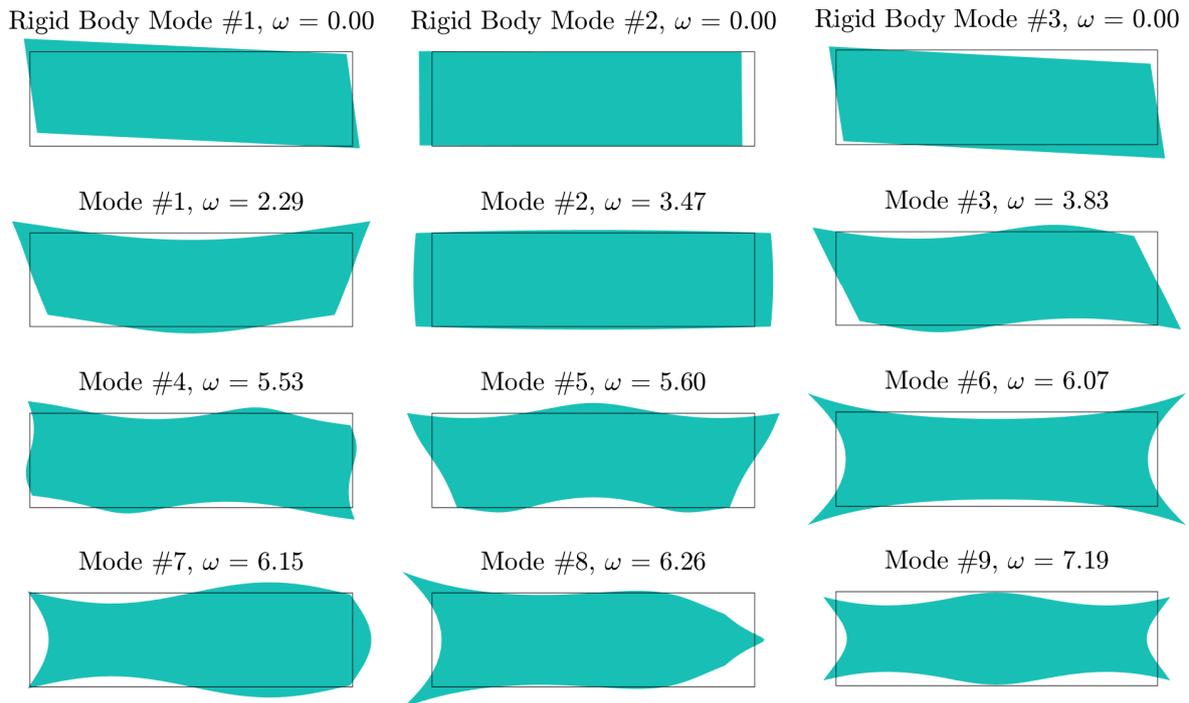


Figure 6-4: Eigenmodes for the first twelve eigenfrequencies. The upper three eigenmodes are rigid body modes, where zero deformations are involved. The mode shapes in this figure should be combined with Figure 6-3 to get the correct insight in the behavior of each eigenmode.

seem to show deformations, but in fact this is rigid rotation. In this case three rigid body modes are involved. When tuning the frequency up, at certain levels, the frequency equals an of eigenfrequency. The associated eigenmodes, the natural vibrations can be described by the modeshapes as depicted in Figure 6-3 and Figure 6-4. Since the rigid body modes do not involve structural deformations, it is very common to start counting eigenfrequencies and eigenmodes from the first structural eigenfrequencies.

Some of the eigenmodes, described in Figure 6-3 and Figure 6-4 are typical bending modes; these eigenmodes exist in beam examples. For example mode number 1, 3, 4 and 5.

6.2.3 Frequency response

A frequency response can be seen as quantitative measure of the output spectrum in response to a certain input. This frequency response is very helpful to characterize the dynamics. In this case, the input is the exerted force. The output of interest is the displacement field. A characteristic way of displaying a frequency response is by using a Bode plot. A Bode plot of this case is depicted in Figure 6-5. In this figure the horizontal output displacement of a point, just above the force application point, is plotted to a certain frequency spectrum. This point, just above the force is chosen, since this point is most of the time displaced. The bottom of the bode plot describes the phase behavior of the system. Zero degree phase means the system is in-phase, the body moves in the same direction as the force. -180° phase means

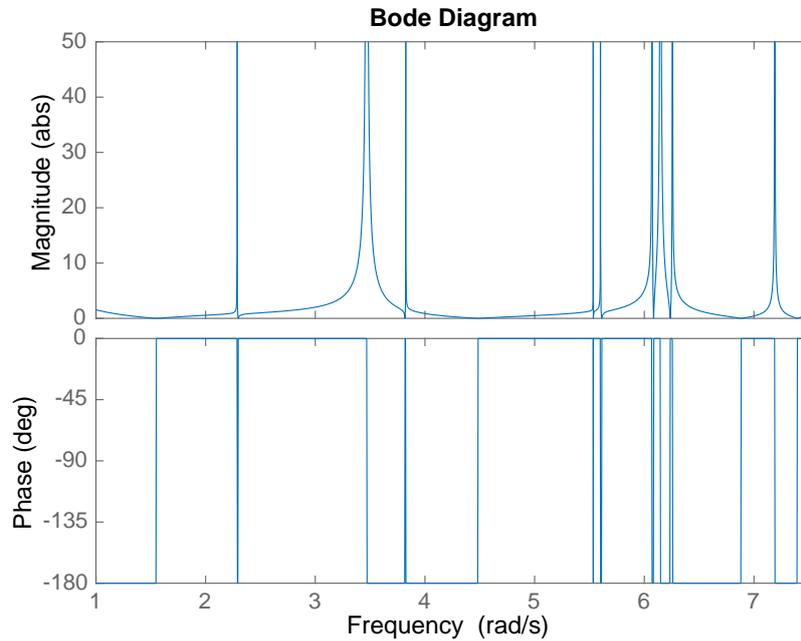


Figure 6-5: Frequency response of a solid body, excited by one harmonic force. The output point is chosen just above the point the force attaches.

the system is completely out of phase and the body moves in the opposite direction of the applied force.

6.2.4 Dynamic mode dependency

Another point of interest in optimization with dynamics, is the influence per mode on the final result. This mode influence can be calculated. By taking the dot product for each eigenvector ϕ_i , as described in (6-4) and the applied force vector \mathbf{f} , the degree to which each mode is excited the force placement can be calculated.

Now, by implementing a weight factor, the influence per mode η_i for the applied excitation frequency ω_i can be calculated accordingly (Rixen, 2008).

$$\eta_i = \frac{\phi_i^T \mathbf{f}}{(\omega_i^2 - \omega^2)} \quad (6-5)$$

In this equation (6-5) can be seen, that modes (actuated at their corresponding eigenfrequency ω_i) far away from the excitation frequency ω , will result in a larger denominator and thus in a smaller contribution η_i of this mode. On the other hand, the closer the excitation frequency ω approaches an eigenfrequency ω_i , the smaller the denominator get and thus the influence on this corresponding mode will be larger. In this section an excitation frequency of $\omega^2 = 8$ is used. This frequency lies just between the first and second eigenmode of the solid stage and thus can give us a good view on the dynamic behavior.

The mode contribution for the case depicted in Figure 6-2 is displayed in Table 6-1. In this first

column the mode number can be seen, the second column holds the associated eigenfrequency. In the third column the mode contribution $\phi_i^T \mathbf{f}$, followed by the scaled contribution η_i , as described in (6-5). This is done, so the difference between $\phi_i^T \mathbf{f}$ and the scaling of the mode contribution can be seen very clearly. In the last column the relative contribution of this mode influence can be found. This contribution is normalized by taking the sum of these first twelve eigenmodes.

As can be seen, the central force placement, is mostly affecting the second (structural) mode, and the second rigid body mode. This means, that the current placement of the single force will result in a displacement field which largely consists of these two modes. A corresponding mode contribution for the six most important modes, over a spectrum of frequencies can be found in Figure A-19. In this schematic it can perfectly be seen which mode contributes how much on every frequency. When the excitation frequency approaches an eigenfrequency, the corresponding mode will be actuated the most and will thus take the largest relative contribution of the total modes. When using this graph and take for example $\omega^2 = 8$, which is used for producing the objective function and also for producing Table 6-1, this frequency can be chosen and the relative contribution values can be seen accordingly, these are in line with Table 6-1.

6.2.5 Double actuator

In the previous example Figure 6-2 only one force is considered. In this subsection however, the force is divided by two and placed at the lower-right and upper-right corner of the objective area (striped area). The distance from the top to the upper force application point is the same as the distance between the bottom and the lower force application point. attachment Since the force is divided by two, the total force remains the same. A schematic of this case is depicted in Figure 6-6. Keep in mind, since the design domain is still solid, the stiffness and mass matrices will not change. The modeshapes in this case are thus the same as in the single force problem (Figure 6-2). As can be seen in Figure 6-6, the objective is improved by almost 30%, with respect to Figure 6-2. The associated mode contribution can be found in

Mode	Eigenfrequency	$\phi_i^T \mathbf{f}$	Mode contribution η_i	Contribution (%)
Rigid #1	1.24e-07	0.20	-0.03	0.82
Rigid #2	1.34e-07	5.05	-0.63	20.40
Rigid #3	2.63e-07	0.07	-0.01	0.30
#1	2.29	0	0	0
#2	3.47	-7.36	-1.82	58.89
#3	3.83	0	0	0
#4	5.53	0	0	0
#5	5.60	0	0	0
#6	6.07	3.89	0.13	4.36
#7	6.15	7.44	0.25	8.07
#8	6.26	-3.82	-0.12	3.96
#9	7.19	4.33	0.10	3.20

Table 6-1: Mode contribution of single force case as described in (Figure 6-2) and taking a frequency of $\omega^2 = 8$

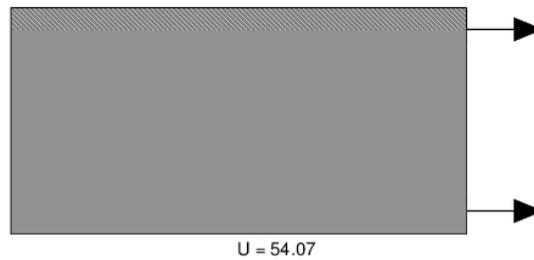


Figure 6-6: Design domain of two forces case example. The striped area indicates the objective area.

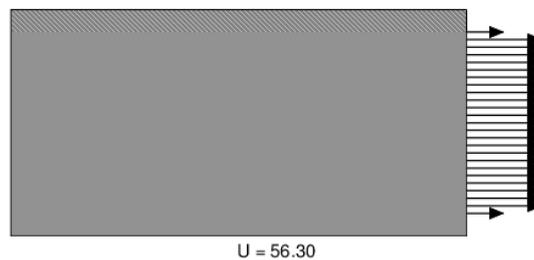


Figure 6-7: Design domain of distributed force case example. The striped area indicates the objective area.

Table A-34. By compare this table with the Table 6-1 conclusions on the mode actuation can be made. By placing the force away from the middle, the second mode is less actuated (η_i). Since this mode is close to the actuation frequency, the total displacement will be lower. The associated mode contribution plot can be found in Figure A-20.

6.2.6 Distributed actuators

In this section the actuation force is distributed on the right side of the domain. Since this distributed load is placed on the elements, the upper and lower nodes only have only one contribution from the elements.

As can be seen, the depicted objective in Figure 6-7 is somewhat worse than using two actuators (6.2.5), but better than only using one force (6.2.2). The corresponding mode contribution can be found in Table A-35. By turning the two force case to a distributed force case, the second mode is actuated more, and since this mode is dominant for this excitation frequency $\omega^2 = 8$, which can be the cause for the larger objective value. The mode contribution graph can be found in Figure A-21.

Now using the information of these three force cases, perhaps an even better solution is possible, by making a mixture of the two force and distributed force cases. In the next section design of actuators will be used to optimize the case.

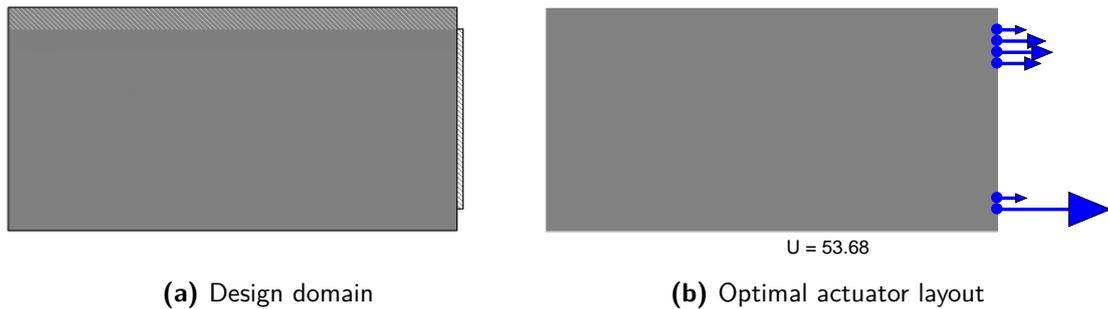


Figure 6-8: Design domain and optimal actuator layout. The gray striped area indicates the objective area. The white striped area indicates the (positive) actuator design domain. The size and placement of the arrows represent the location and magnitude of the optimized force layout. The associated mode contribution can be found in Table A-36.

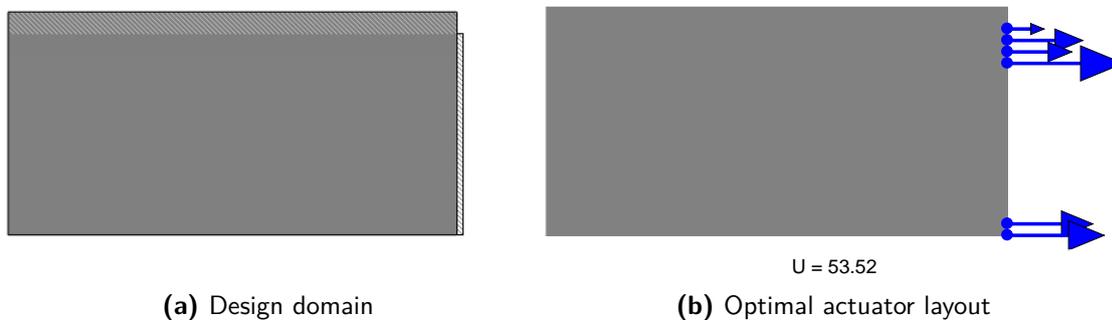


Figure 6-9: Design domain and optimal actuator layout. The gray striped area indicates the objective area. The white striped area indicates the (positive) actuator design domain. The size and placement of the arrows represent the location and magnitude of the optimized force layout. The associated mode contribution can be found in Table A-37.

6.3 Design of actuators

Up to here, the force is applied at a certain location(s). This section is dedicated to the design of actuator placement, which is already explained in Chapter 5. By making a combination of this design of actuator placement with the described dynamics in 6.2 a solution of the best placement of actuators can be determined, while respecting the dynamics.

The design domain looks like before (Figure 6-2), but now it includes a design domain for actuators on the right hand side. This new design case is depicted in Figure 6-8a. Of course, the main target will be to improve the previous objective. In line with the previous chapters, a new optimization formulation (6-6) can be made, including the design of actuators. As already described in 5-15, the objective is minimizing the squared displacement. The total mass of the structure can be calculated by making a summation of elemental mass. This elemental mass is made of the element's density ρ_e and the material density ρ_0 . The total applied force should be enough to move the body with a pre-defined acceleration vector a , in this case it would be a horizontal acceleration of $\omega^2 = 8$, in order to compare the results with the previous examples. The force that can be used to meet this constraint can vary between 0 and 10 for each actuator location (depicted by the white striped area in Figure 6-8a).

$$\begin{aligned}
& \min_{f_i, \rho_e} u_a \\
& \text{s.t.} \quad u_a = (\mathbf{L}^T \mathbf{u})^2 \\
& \quad (\mathbf{K} - \omega^2 \mathbf{M}) \mathbf{u} = \mathbf{f} \\
& \quad \mathbf{K} = \sum_{e=1}^n \rho_e^p \mathbf{K}_e \\
& \quad \mathbf{M} = \sum_{e=1}^n \mathbf{M}_e \\
& \quad \sum_{e=1}^n \nu_e \rho_e \leq V \\
& \quad m \cdot \mathbf{a} \leq \sum_{i=1}^n f_i \quad i = 1, \dots, N_i \quad a = \omega^2 \\
& \quad m = \sum_{e=1}^n \nu_e \rho_e \rho_0 \quad e = 1, \dots, N \\
& \quad 0 \leq f_i \leq 10
\end{aligned} \tag{6-6}$$

This optimization problem is then solved using the MMA-solver, which is used along this report. The minimum vertical displacement of the top layer can now be rewritten including an adjoint function.

$$u_a(f_i) = (\mathbf{L}^T \mathbf{u})^2 + \boldsymbol{\lambda}^T [(\mathbf{K} - \omega^2 \mathbf{M}) \mathbf{u} - \mathbf{f}] \tag{6-7}$$

Where \mathbf{L} is the selection vector of the displacement, in this case the top layer of the design domain.

The corresponding objective sensitivities can now be calculated accordingly.

$$\frac{\partial u_a}{\partial f_i} = \left[2\mathbf{L}^T \mathbf{u} \mathbf{L}^T + \boldsymbol{\lambda}^T (\mathbf{K} - \omega^2 \mathbf{M}) \right] \frac{\partial \mathbf{u}}{\partial f_i} - \boldsymbol{\lambda}^T \frac{\partial \mathbf{f}}{\partial f_i} \tag{6-8}$$

Where $\boldsymbol{\lambda} = -2(\mathbf{K} - \omega^2 \mathbf{M})^{-1} \mathbf{L} \mathbf{u}^T \mathbf{L}$ to remove the $\partial \mathbf{u}$ terms.

The optimization result is depicted in Figure 6-8b. The blue arrows indicates the force applications, where the magnitude of the force is proportional to the size of the arrowhead and the total length of the arrow. A threshold value of 0.05 is chosen. This means that a force arrow is only displayed when its value represents a minimum of five percent of the maximum force applied.

As can be seen in Figure 6-8b, a big force is attached at the bottom right, and some cluster of forces at the top right. The objective is slightly better than the two forces case, as described in Figure 6-6. The total designed force equals the $m \cdot \mathbf{a}$ term, which means the optimizer does not use more force than strictly needed to meet the constraint. This is in line with the preliminary thoughts, since additional force will result in additional stresses and thus additional deformations.

The improvement on the objective is made, but some more improvement should be possible. By extending the actuator design domain to include the bottom right corner as actuator design domain this improvement could be possible. The updated design domain is depicted in Figure 6-9a. The optimal actuator layout can be seen in Figure 6-9b. As can be seen, the

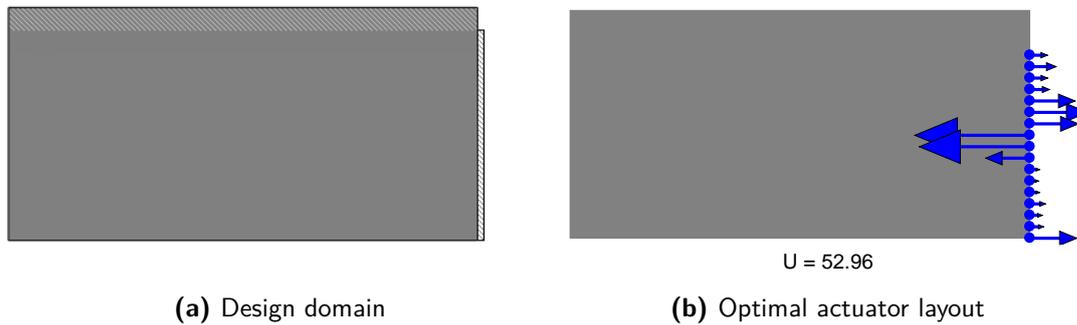


Figure 6-10: Design domain and optimal actuator layout, while enabling negative forces design. The gray striped area indicates the objective area. The white striped area indicates the actuator design domain. The size, placement and direction of the arrows represent the location, magnitude and orientation of the optimized force layout. The associated mode contribution can be found in Table A-38.

extension of the actuator design domain leads to force attachment at that additional area and thus a very different force layout. Thanks to this extension, an improvement to the objective can be made. A very small improvement, but an improvement.

6.3.1 Design of negative forces

In 6-9 an improvement for the objective value is made available. An even further improvement should be possible. By enabling the possibility to create negative forces (forces in the opposite direction of the acceleration) an improvement can be made, which may seem counterintuitive at first. The same design domain as depicted in Figure 6-9a and almost the same formulation as 6-6. Only one adjustment should be made here, by changing the magnitude of the actuator design domain to $-10 \leq f_i \leq 10$. The updated optimization result can be found in Figure 6-10b. As can be seen, there is a big negative force at the mid-half. In general, it seems a bit unexpected the optimal result would even use negative forces. Since the total force should still at least equal the ($\mathbf{f} = m \cdot \mathbf{a}$) term, a negative force will thus also lead to larger positive forces. The reason to create negative forces is to counteract the dynamic eigenmodes. A negative force can be used to counteract or reduce the dynamical effects, although a larger amount of forces should be used.

It can be concluded that improvements in the objective can be made by placing forces in other direction than the acceleration force. In the next section this approach is taken a step further.

6.3.2 Design of force at multiple sides

Up to here, the force application could only be attached at the right-half side of the design body. However, improvements can be made by making multiple sides of the body available for actuator placement. In this section, bottom force can be applied at the bottom-side of the design domain. These forces can be upwards (positive) or downwards (negative). Of course, these vertical forces are not contributing to the ($\mathbf{f} = m \cdot \mathbf{a}$) expression. But these forces can be used to reduce mode excitations. An updated design domain can be found in Figure 6-11a,

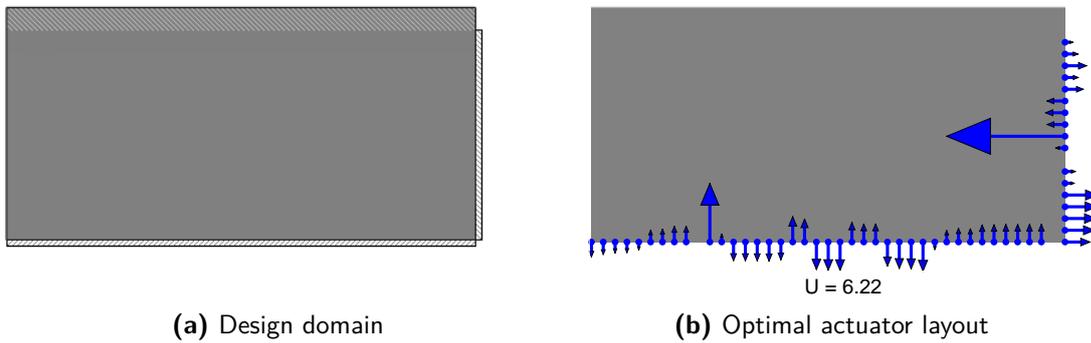


Figure 6-11: Design domain and optimal actuator layout, while enabling negative forces design. The gray striped area indicates the objective area. The white striped area indicates the actuator design domain. The size, placement and direction of the arrows represent the location, magnitude and orientation of the optimized force layout. The associated mode contribution can be found in Table A-39.

the corresponding optimal actuator location layout can be found in Figure 6-11b. As can be seen, a huge improvement can be made to the objective value. The extension of the actuator design domain indeed leads to a very big improvement of this optimization problem.

Another possibility could be to include the left-half side of the body in the actuator design domain. An even larger improvement of the objective value U can be made. The updated design domain can be seen in Figure 6-12a. The optimal actuator layout is depicted next to it in Figure 6-12b. As can be seen, a variety of forces are applied to the body. The total applied force is almost exactly the minimal force needed, to accelerate the body with an acceleration of $w^2 = 8$. Some big forces at the mid-half of both sides are pointing left, which is in opposite direction of the acceleration, to reduce the excitations caused by the dynamic behavior. Another two big forces are needed to actually achieve the minimal total horizontal force. Another point of interest is the steadily decreasing of the contribution of the second mode, which can be seen in Table A-40. For the design result depicted in Figure 6-11b this contribution is almost zero. This means the optimizer want to make a design which has very little impact from this second mode. The fact it almost hit zero means the optimizer did a very good job at this one.

By enabling the left-half side of the body for actuator design domain, a very nice objective improvement can thus be achieved.

An even better solution could be to combine Figure 6-11 and Figure 6-12. The result is depicted in Figure A-24. Here, a big problem when optimizing this type of design problem, is the possibly overfitting of the model. The optimizer has just too many variables and the optimizer is more likely to approach a (high) local optimum. The result depicted in Figure A-24 shows a distribution along all sides of the design domain. The horizontal force is almost twice the minimum needed force to achieve the prescribed acceleration. This could also be a symptom of the overfitting of the model. It can be concluded that, in order to achieve a maximal optimization result, the design domain should not be too vague or too big.

Another option to optimize, is actuating at the natural frequency. Of course, it is not common to actuate at or near an eigenfrequency. But in some cases, when the material and frequency are given, it could be possible we need to optimize the actuator layout in order to trigger the modes as little as possible.

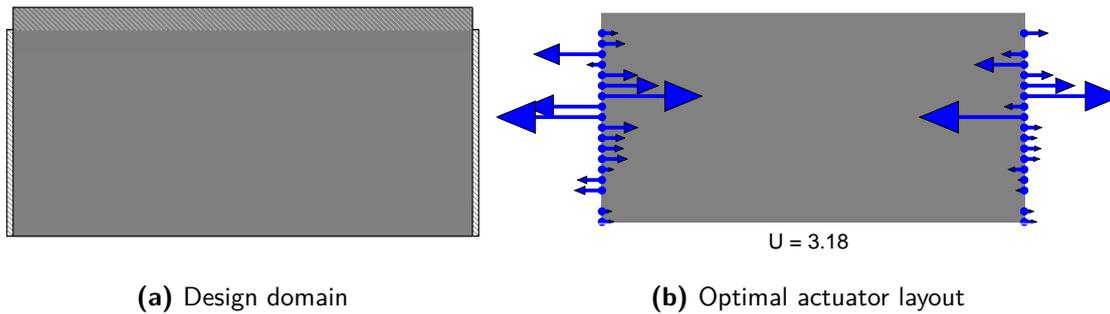


Figure 6-12: Design domain and optimal actuator layout, while enabling negative forces design. The gray striped area indicates the objective area. The white striped area indicates the actuator design domain. The size, placement and direction of the arrows represent the location, magnitude and orientation of the optimized force layout. The associated mode contribution can be found in Table A-40.

6.4 Topology optimization for dynamic performance

Up to here, the stage consisted of a completely solid stage. This thesis research is based on topology optimization, however. By enabling the possibility of changing the topology in the design domain, even better results can probably be achieved. The combination of actuator placement and topology optimization is already touched in 5.4 for static problems. By enabling dynamics (6.2), a more advanced optimization problem can be set up.

The objective in this section remains the same, namely minimizing vertical displacement of the top layer. This top layer should be a solid area. Since in this section the topology can be changed, a restrictive area (3.2.1) is introduced at the top of the design domain.

The main focus will be to look for a better performing wafer stage example, in terms of the vertical displacements of the top layer. Eigenmodes and frequency response are already described in 6.2.2 and 6.2.3. These dynamic properties depend on the stiffness and mass distribution. Since the topology can now be changed, the eigenfrequencies, eigenmodes and frequency response will also change during the optimization process.

6.4.1 Topology optimization for fixed force

To understand the behavior of topology optimization, in this section a topology optimization example for a fixed force case is considered. Since the two force example (6.2.5) seems to be a good starting point, we will use this example for topology optimization. In this example the force remains the same, the magnitude is based on moving a solid stage ($\mathbf{f} = m \cdot \mathbf{a}$). This means the optimizer could make a complete solid stage. On the other hand, removing material does not contribute to a lower force application in this example. This example is created to see whether or not the optimizer wants to remove material and what regions should be void. The design domain can be found in Figure 6-6. As can be seen, two problems seem to come up. At first, the lower force is not directly connected to the structure by solid regions. This means the force attachment has no physical interpretation. This problem is already seen in Figure 5-5b, with a possible solution as described in 5.4.2, to overcome this problem.

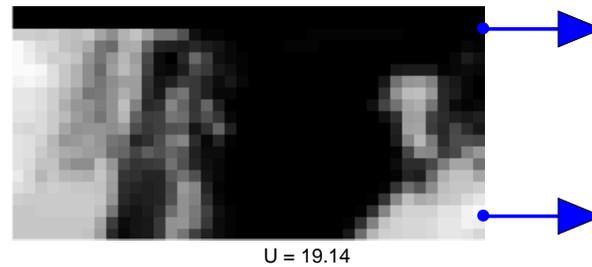


Figure 6-13: Optimal topology for minimal displacement using static forces. The associated total vertical displacement of the top layer is depicted under the figure. The total horizontal force used is $\mathbf{f} = 3.20$.

A compliance constraint should be implemented, in order to ensure the force is attached to the structure. By implementing a compliance constraint, the optimizer is prohibited from creating very large displacements at the point of force attachment.

Another topology phenomenon comes up in Figure 6-13, namely gray regions. Gray regions also have no physical interpretation. A penalty-term of $p = 3$ is already implemented, but still a lot of gray regions exist. Since the optimization problem depends on the topology and subsequently on the frequency response, it could be possible the optimizer only wants part of the stiffness (and mass) of certain elements, in order to reduce certain mode excitations.

6.4.2 Topology optimization for double actuator

Enabling topology optimization can indeed enhance the result and could contribute to a smaller objective value, hence less displacements. In Figure 6-13 an example of this topology optimization result is depicted. Here, the force remained fixed. Note that the applied force here, does not change, while the weight is reduced. Smaller mass means less force required ($\mathbf{f} = m \cdot \mathbf{a}$).

It could therefore be helpful, to implement this equation in the optimization routine. A weight reduction could therefore result in a force reduction. This force reduction could lead to less deformation in the material and therefore in a smaller displacement field of the top layer. Design of actuators (6.3) could be very helpful also, to calculate the optimal actuator layout. As already described in 6.4.1, two problems should be overcome. In this section a compliance constraint is implemented, the same way as introduced before in 5.4.2. As can be seen in Figure 6-14, the force seems to be attached to the structure. Since the minimum force needed is from now on coupled to the mass, a mass reduction could thus lead to a force reduction. Note that the minimum force to accelerate the solid body is $\mathbf{f} = 3.20$. To get an insight in the force reduction that can be achieved, the associated applied total horizontal force is depicted in the legend of each optimization case.

The second problem, gray regions, is also investigated in Figure 6-14. Here, the top layer is still solid material and the displacements of this top layer should be reduced. The topology can be varied in the design domain and actuators can be designed at the two points as depicted in Figure 6-6. Although the density distribution is different for the cases as depicted in Figure 6-14, the volume fraction is around the same value. This also holds for the minimum horizontal applied force. Typically, it can be concluded, that weight reduction results in force reduction.

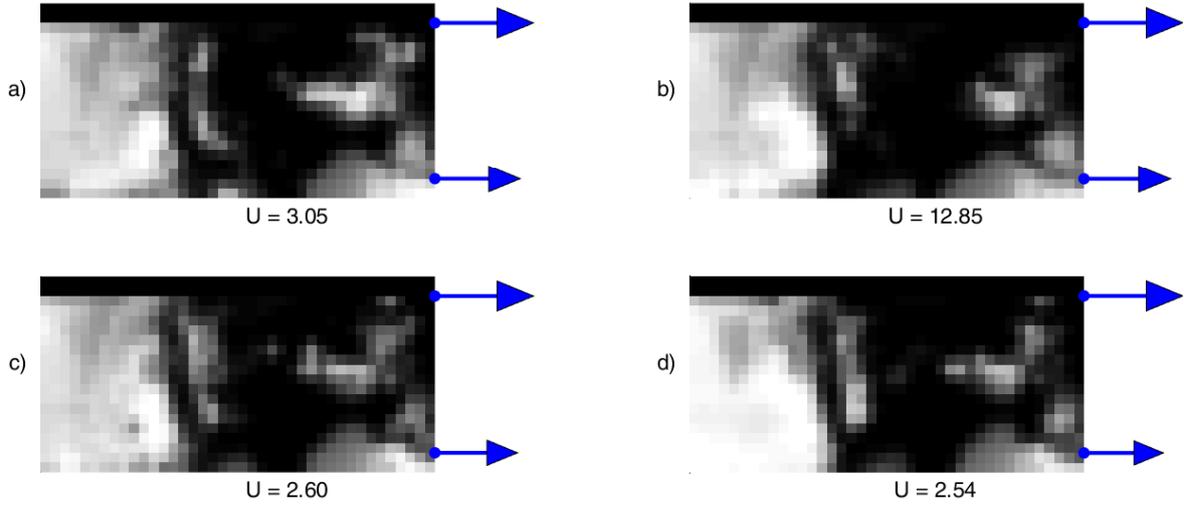


Figure 6-14: Optimal actuator placement including topology optimization for minimal displacement using two forces, including a compliance constraint. The penalty is defined by a) $p = 3$, b) $p = 4$, c) $p = 5$, d) $p = 6$. The associated total vertical displacements of the top layer are shown under each figure. The total horizontal force used is a) $\mathbf{f} = 2.16$, b) $\mathbf{f} = 2.16$, c) $\mathbf{f} = 2.17$, d) $\mathbf{f} = 2.13$.

The same optimization problem as stated in 6-6 holds, with the notation that ρ_e can now be varied. This means the same vertical displacement of the top layer is considered. The objective thus remain the same.

$$u_a(f_i, \rho_e) = (\mathbf{L}^T \mathbf{u})^2 + \boldsymbol{\lambda}^T [(\mathbf{K} - \omega^2 \mathbf{M}) \mathbf{u} - \mathbf{f}] \quad (6-9)$$

This equation should also be differentiated to the density variable ρ_e . This sensitivity can be calculated as:

$$\frac{\partial u_a}{\partial \rho_e} = \left[2\mathbf{L}^T \mathbf{u} \mathbf{L}^T + \boldsymbol{\lambda}^T (\mathbf{K} - \omega^2 \mathbf{M}) \right] \frac{\partial \mathbf{u}}{\partial \rho_e} + \boldsymbol{\lambda}^T \frac{\partial \mathbf{K}}{\partial \rho_e} \mathbf{u} - \omega^2 \boldsymbol{\lambda}^T \frac{\partial \mathbf{M}}{\partial \rho_e} \mathbf{u} \quad (6-10)$$

Where $\boldsymbol{\lambda} = -2(\mathbf{K} - \omega^2 \mathbf{M})^{-1} \mathbf{L} \mathbf{u}^T \mathbf{L}$ to remove the $\partial \mathbf{u}$ terms.

By varying the SIMP penalty-term, some insight in the behavior of the structure can be achieved. While increasing the penalty-term from $p = 3$ (Figure 6-14a), to $p = 6$ (Figure 6-14d), it can be clearly seen that the behavior tend to optimize towards a black-and-white solution, which is better physically interpretable. Although a high penalty is implemented at Figure 6-14d, the structure still wants to create gray regions. This means the optimizer want some stiffness in that particular region, even when this will have a big trade-off. It can be concluded, the total vertical displacement of the top layer is decreasing by the implementation of topology optimization, when compared to the massive stage from Figure 6-6.

6.4.3 Side force and topology optimization

As already concluded in 6.3, design of actuators along the side can be helpful in achieving lower displacements. In this section the complete righthand side of the design domain can be used

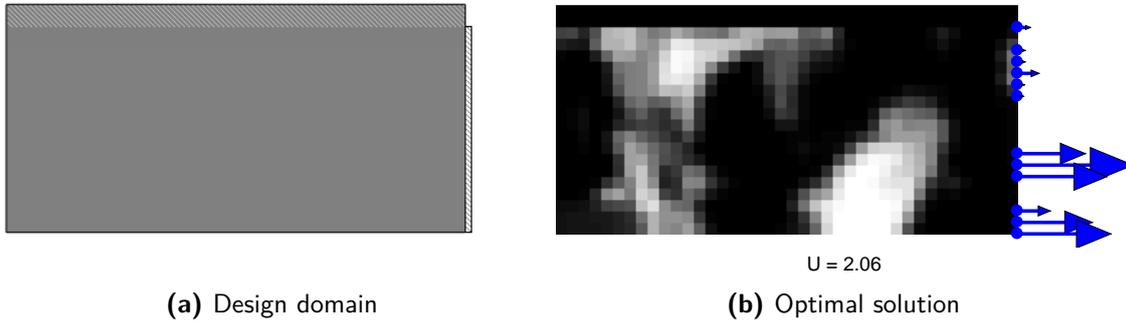


Figure 6-15: Design domain and optimal actuator layout, including topology optimization. The gray striped area indicates the objective area. The white striped area indicates the (positive) actuator design domain. The size and placement of the arrows represent the location and magnitude of the optimized force layout. The total horizontal force used is $\mathbf{f} = 2.51$.

for force actuation. By enabling this option in combination with topology optimization, it allows the optimizer to avoid low eigenmodes and actuating at points where less excitation is experienced. The topology can be used to avoid certain modes, the force can be used to avoid excitations of certain modes. The combination can be used to create an efficient frequency response for the particular case. An example of this problem is depicted in Figure 6-15. In the design domain (Figure 6-15a) the design domain for actuators can be found. The optimal topology and actuator distribution is depicted in Figure 6-15b. As can be seen from this solution, enabling topology optimization can enhance performance, compared to the massive stage example without topology optimization (Figure 6-9b). Also, by creating a bigger force design domain, reducing displacements of the top layer can be achieved, compared to the double actuator design case (Figure 6-14).

6.4.4 Negative forces and topology optimization

Up to here, this section (6.4) only includes (design of) positive forces. However, as can be seen in 6.3.1, enabling the possibility for creating negative forces could counteract or reduce certain mode excitations.

A problem comes up here, when implementing the compliance constraint (5.4.2). This compliance constraint is defined as:

$$c = \mathbf{f}^T \mathbf{u} \quad (6-11)$$

This formula is pretty straightforward, but a problem comes up when creating negative forces. The point of negative force attachment, could have a positive displacement at that particular point. This is especially true in this case, since the body needs to move to the right. The negative contribution could make it easier to meet the compliance constraint, and allow again forces that act on gray/void elements. To overcome this problem, we want to calculate the compliance as the absolute values of f and u . A simple multiplication of these absolute values gives a problem, since this function is not differentiable. Note the definition of an absolute value:

$$|x| = \sqrt{x^2} \quad (6-12)$$

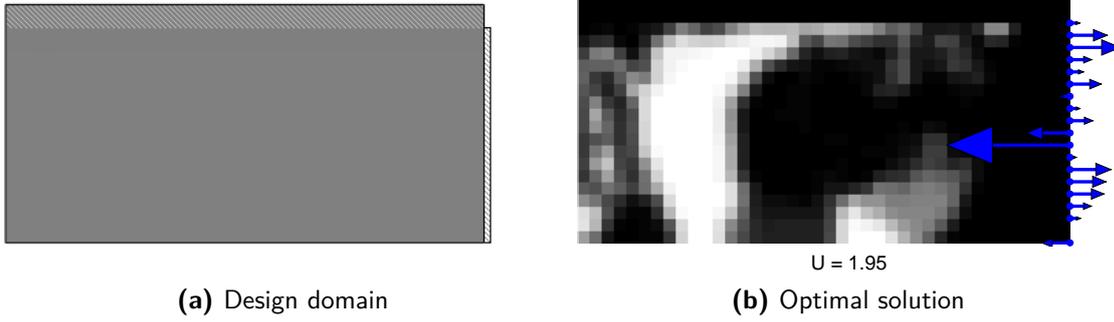


Figure 6-16: Design domain and optimal actuator layout, while enabling negative forces design and using topology optimization. The gray striped area indicates the objective area. The white striped area indicates the actuator design domain. The size, placement and direction of the arrows represent the location, magnitude and orientation of the optimized force layout. The total horizontal force used is $\mathbf{f} = 2.35$.

A possible solution is to introduce a very small value ϵ to the value that needs to become absolute. This ϵ can be implemented in 6-12 and subsequently in 6-11:

$$\begin{aligned} c &= |\mathbf{f}^T \mathbf{u}| \\ &= \left(\sqrt{\mathbf{f}^2 + \epsilon} \right)^T \left(\sqrt{\mathbf{u}^2 + \epsilon} \right) \end{aligned} \quad (6-13)$$

When choosing the value ϵ small enough, the influence will become very small and can be neglected. The main problem is here the multiplication of vectors f and u . These vectors should be squared element-wise, by making use of the Hadamard product, which was already introduced in 5-19. The compliance in correct vector notation can now be rewritten as:

$$c = \left(\sqrt{\mathbf{f} \odot \mathbf{f} + \epsilon} \right)^T \left(\sqrt{\mathbf{u} \odot \mathbf{u} + \epsilon} \right) \quad (6-14)$$

This compliance constraint is differentiable to the force and density variable. By adding again an adjoint vector, the sensitivities can be calculated.

$$c(f_i, \rho_e) = \left(\sqrt{\mathbf{f} \odot \mathbf{f} + \epsilon} \right)^T \left(\sqrt{\mathbf{u} \odot \mathbf{u} + \epsilon} \right) + \boldsymbol{\lambda}^T \left[(\mathbf{K} - \omega^2 \mathbf{M}) \mathbf{u} - \mathbf{f} \right] \quad (6-15)$$

with the corresponding sensitivities:

$$\frac{\partial c}{\partial \rho_e} = \left[\frac{\mathbf{u} \sqrt{\mathbf{f} \odot \mathbf{f} + \epsilon}}{\sqrt{\mathbf{u} \odot \mathbf{u} + \epsilon}} + \boldsymbol{\lambda}^T (\mathbf{K} - \omega^2 \mathbf{M}) \right] \frac{\partial \mathbf{u}}{\partial \rho_e} + \boldsymbol{\lambda}^T \frac{\partial \mathbf{K}}{\partial \rho_e} \mathbf{u} - \omega^2 \boldsymbol{\lambda}^T \frac{\partial \mathbf{M}}{\partial \rho_e} \mathbf{u} \quad (6-16)$$

$$\frac{\partial c}{\partial f_i} = \left[\frac{\mathbf{u} \sqrt{\mathbf{f} \odot \mathbf{f} + \epsilon}}{\sqrt{\mathbf{u} \odot \mathbf{u} + \epsilon}} + \boldsymbol{\lambda}^T (\mathbf{K} - \omega^2 \mathbf{M}) \right] \frac{\partial \mathbf{u}}{\partial f_i} + \left[\frac{\mathbf{f} \sqrt{\mathbf{u} \odot \mathbf{u} + \epsilon}}{\sqrt{\mathbf{f} \odot \mathbf{f} + \epsilon}} - \boldsymbol{\lambda}^T \right] \frac{\partial \mathbf{f}}{\partial f_i} \quad (6-17)$$

Where $\boldsymbol{\lambda} = - \left(\mathbf{K} - \omega^2 \mathbf{M} \right)^{-1} \left(\frac{\mathbf{u} \sqrt{\mathbf{f} \odot \mathbf{f} + \epsilon}}{\sqrt{\mathbf{u} \odot \mathbf{u} + \epsilon}} \right)$ to remove the $\partial \mathbf{u}$ terms.

This compliance constraint is now used to optimize the case depicted in Figure 6-16a. As

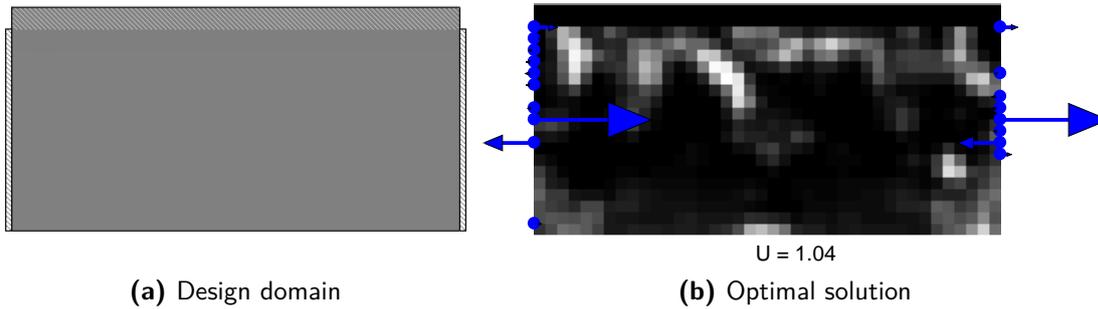


Figure 6-17: Design domain and optimal actuator layout, while enabling negative forces design and using topology optimization. The gray striped area indicates the objective area. The white striped area indicates the actuator design domain. The size, placement and direction of the arrows represent the location, magnitude and orientation of the optimized force layout. The total horizontal force used is $\mathbf{f} = 2.89$.

already stated, the force design can vary from positive to negative values. The optimal actuator layout and corresponding topology can be found in Figure 6-16b. As can be seen here, the total vertical displacement of the top layer is again decreased, when compare to the previous case in Figure 6-15b where only positive forces can be created along the right side of the design domain. Also, an big improvement with respect to the massive case Figure 6-12b can be achieved.

6.5 Topology optimization for actuator placement

In this section a combination of all previously described knowledge, examples and case studies will come together. The main focus is still improving the objective value by minimizing vertical displacements of the top layer. We have already seen an example of using multiple sides for actuator placement in 6.3.2. By adding the design of density, by terms of topology optimization (6.4) and using the compliance constraint described in 6-15 some promising improvements are already shown. In this section, design of forces at multiple sides is combined with topology optimization. Preliminary thoughts tells us that a combination of these options can improve the objective even further.

In Figure 6-12 an example of using both sides of the design domain for actuator placement is shown. This same actuator design domain is used, but now enabling topology optimization. The result is depicted in Figure 6-17b. The force distribution is somewhat different from the massive case (Figure 6-12). The objective improvement is made, however. The volume fraction used to achieve this can be labeled as large, compared to the previous topology examples in 6.4.

Another option could be to design at the right side of the domain and the bottomside of the domain. This example for a massive stage is already depicted in Figure 6-11. Now by implementing topology optimization perhaps even better results can be achieved. The design domain is depicted in Figure 6-17a, with the corresponding optimized result depicted in Figure 6-17b. As can be seen here, the design of actuators differs from the massive stage example with the same actuator design domain (Figure 6-11). Also, the objective value, the vertical displacement of the top layer is reduced even further, compared to Figure 6-11 and

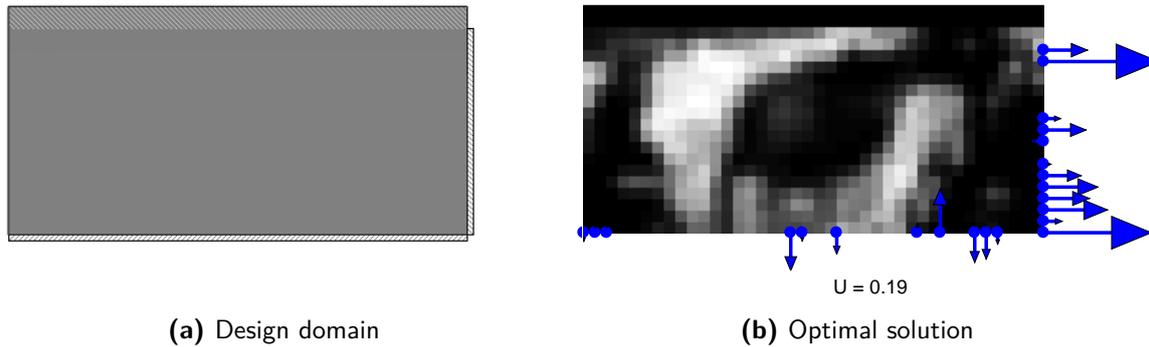


Figure 6-18: Design domain and optimal actuator layout, while enabling negative forces design and using topology optimization. The gray striped area indicates the objective area. The white striped area indicates the actuator design domain. The size, placement and direction of the arrows represent the location, magnitude and orientation of the optimized force layout. The total horizontal force used is $\mathbf{f} = 2.52$.

Figure 6-16. Although the penalty term is set to a high value ($p = 6$), the optimal result still shows some gray regions, even after a high number of iterations, the gray regions still have the preference.

The overall vertical displacement of the top layer is reduced to $U = 0.19$, as can be seen in Figure 6-18b. When a comparison is made to the original model, a massive stage with two point forces ($U = 54.07$), the change is substantial. The total reduction of the displacements is -99.6% . So it can be stated, using topology optimization and actuator placement can have a huge impact on reducing displacements. An overview of all the produced examples in this chapter can be found in Table 6-2.

6.5.1 Improving gray regions

As already stated before, the best solution as depicted in Figure 6-18 includes gray regions. One way to improve this result for manufacturing, it could be a good idea to increase the penalty term even further. In Figure 6-19a an example of an increasing penalty ($p = 11$) is given. As can be seen, there is some improvement in terms of black-and-white solutions. This results however, in a larger displacement field, since the optimizer is even more forced to create black-and-white solution.

The results as achieved in this chapter are filtered during the optimization process using a density filter (3.2.5), using another filter, in this case the Heaviside filter (3.2.5) is used to force the optimizer towards black-and-white solutions. The result of using this Heaviside filter is depicted in Figure 6-19b. As can be seen, the solution is improved even more, in terms of black-and-white regions. This also results in a larger displacement field of the top layer, but it is better physically interpretable.

6.5.2 Changing conditions

Up to here, we let the optimizer choose the best solution, with no maximum weight restrictions (Although the solution is limited to use 100% of the material). In some case however, it could

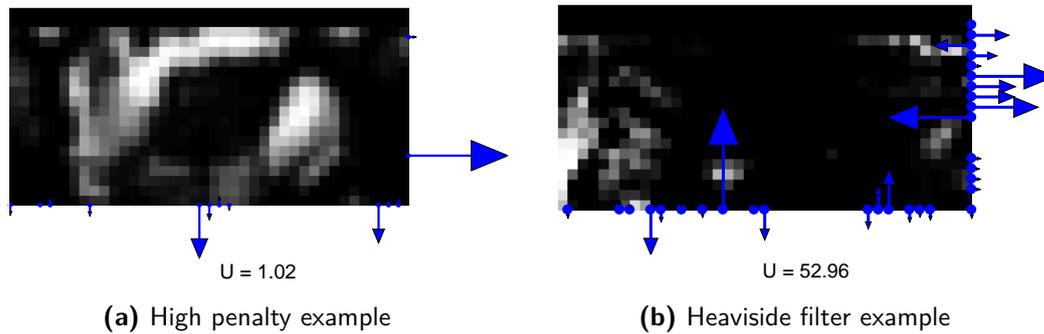


Figure 6-19: Design domain and optimal actuator layout, while enabling negative forces design and using topology optimization for two different solving situations: a) high penalty example ($p = 11$), b) Heaviside filter example. The size, placement and direction of the arrows represent the location, magnitude and orientation of the optimized force layout.

be possible that weight reductions are required. In Figure A-26 some examples with these restrictions are solved. As can be seen, a weight reduction as constraint results in larger displacement fields. This is in line with the preliminary thoughts, since these additional restrictions causes less stiffness properties.

Another changing condition could be changing actuation frequency. When the actuation frequency is pre-required, the optimal solution will be different. In Figure A-27 two examples for different actuation frequencies can be found. The examples consists of taking the half of the original actuation frequency ($\omega^2 = 4$) and taking the double of the original actuation frequency ($\omega^2 = 16$). The behavior of the optimizer will be the same, namely placing as much as eigenfrequencies in front of the actuation frequency, in order to reduce the mode excitations. The optimal result is, however, heavily dependent on the mass and the applied force of the structure. As can be seen, the volume fraction that is used by the optimizer is around the same. Changing actuation frequency however can thus result in more or less force needed to achieve the desired acceleration ($\omega^2 = a$). This same approach can also be used to optimize the problem sketched in A.8.1.

It can be concluded that the optimizer can handle multiple restrictions, for example a weight restriction, or a desired actuation frequency. Both can be implemented and the algorithm can calculate the optimal solution for each particular case.

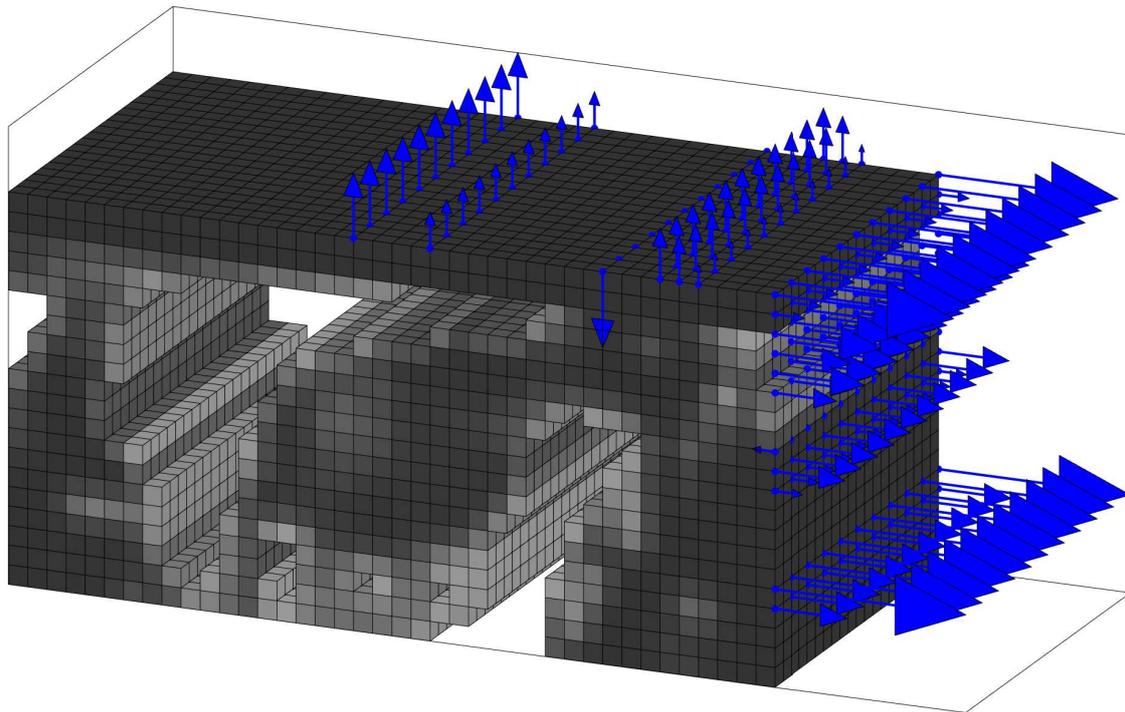


Figure 6-20: A 2D lateral extrusion of the optimal wafer stage as depicted in Figure 6-18b.

6.6 3D extrusion

Up to here, only 2D situations are considered. There is no step taken towards a three dimensional case. The main reason for this is the computational time. In this section however, a 3D extrusion is made. This extrusion is just a 2D lateral case in another dimension. A threshold value of 0.5 is chosen. This means all densities below this threshold value are displayed as void regions, for a better visual representation. A same threshold method is implemented for the force distribution. Note that the lateral extrusion contains one half of the width of the wafer stage. This is only done for a better view to the user. As already explained in 6.1 this wafer stage should be used to produce circular wafers. The width and depth of the wafer should thus be the same size. A full representation of this lateral extrusion can be found in A.8.5.

6.7 Conclusions

In chapter 5, some words are spent on the design of actuator placement. A simple cantilever beam was optimized using design of actuators. Later on, topology optimization was included also.

In this chapter a wafer stage, as described in 6.1 is introduced. This wafer stage is simplified in a 2D example, which is used as design domain. This wafer stage is harmonically actuated by the implementation of dynamics (6.2). By investigating dynamical phenomena like eigenmodes (6.2.2) and frequency responses (6.2.3) these dynamics are investigated for three

simple cases. The double actuator case (6.2.5) seems to be the best solution of these three considered cases.

An interesting field is the design of actuators in this dynamic spectrum. By taking several design cases (6.3), a distributed force on the left- and righthand side of the design domain (Figure 6-12) seems to be very promising.

Since dynamics are involved in this wafer stage, the design is heavily dependent on its frequency response. This frequency response is determined by its stiffness and mass properties. It could therefore be very helpful to have a look at the volume distribution this material by the implementation of topology optimization (6.5). Several cases are considered and the best solution to this dynamic wafer stage design problem is by designing actuators on the right-hand side and the bottomside of the spectrum (Figure 6-18). A problem that could come up is the translation to additive manufacturing. For example the existence of gray regions should be solved (6.5.1). A translation to this additive process also needs to be made by introducing a third dimension (6.6), although this is not explicitly solved in this chapter.

Using topology optimization in combination with actuator placement can be very promising in dynamic problems. In this chapter a displacement reduction of 99.6% is made. Although a translation to the real world and additive manufacturing needs to be made, still a lot of benefits can be achieved by the combination of these optimizations.

Design problem	Displacement top	Applied force	Relative improvement
Figure 6-2	61.92	3.20	+14.5%
Figure 6-6	54.07	3.20	0
Figure 6-7	56.30	3.20	-4.1%
Figure 6-8	53.68	3.20	-0.1%
Figure 6-9	53.52	3.20	-0.1%
Figure 6-10	52.96	3.20	-0.2%
Figure 6-11	6.22	3.20	-88.5%
Figure 6-12	3.18	3.20	-94.1%
Figure 6-13	19.14	3.20	-64.6%
Figure 6-14d	2.54	2.13	-95.3%
Figure 6-15	2.06	2.51	-96.2%
Figure 6-16	1.95	2.35	-96.4%
Figure 6-17	1.04	2.89	-98.1%
Figure 6-18	0.19	2.52	-99.6%

Table 6-2: Displacement overview for all considered cases from Chapter 6. The first column states the design result, the second column states the total absolute vertical displacement of the top layer. The third column states the used force (note that the solid stages all used the minimum required force \mathbf{f} to achieve the desired acceleration ω^2). The last column shows the relative change, as compared to the massive case, with two forces (Figure 6-6), since this seems to be a good starting solution in prior.

Part IV

Closure

Conclusions and Recommendations

This chapter concludes this Master of Science graduation project. All the chapters are concluded in 7.1. Next, some recommendation for future research are made in 7.2.

7.1 Conclusions

In this thesis, topology optimization is used for a variety of design problems. At first, design of supports is considered. When the placement of supports is not prescribed, design of supports tells us the best support layout. By making the combination with topology optimization, the design of supports cooperates with the topology to create a structure which optimizes static behavior.

The classical approach of constructing a bridge does not include support design. By the implementation of a support cost function, the best support layout can be determined, while respecting the surroundings. Design of supports can for example be used to minimize environmental damage, without conflicting the objective of a bridge.

Design of actuator placement can be used to determine the best actuator layout for a given objective. When optimizing towards minimal displacement or compliance problems, a minimum force constraint should be introduced. This minimum introduction is necessary, to prevent the optimizer creating trivial solutions by placing zero force, which can be the best solution (in a static domain), for the minimization of displacements.

The combination with topology optimization has shown cooperation between both design variables. When optimizing in a certain design domain, it can be helpful to introduce density dependency. It has shown that density dependency is very effective in minimizing displacements of the constructed area only.

In chapter 6, dynamics are considered by introducing a harmonic excitation. Design of actuators is shown to be very effective in reduction of a certain displacement field. The frequency response, caused by the harmonic excitation at a certain frequency, can be used to place forces

in a smart way, in order to reduce the objective. The actuator placement can be placed in a such a manner, that modes are exerted in a way that contributes to improving an objective. Implementation of Newton's second law is necessary, to ensure the applied force is large enough to excite the body with the desired frequency. This is done by introducing an additional acceleration constraint. In general, more design space for actuator placement results in better objectives. There are some situations however, where overfitting occurs. When giving the optimizer too much freedom, the result is more likely stuck at a local minimum.

The combination of design of actuator placement with topology optimization is performed in a dynamic domain. Since the topology can change, the frequency response can change. This change of frequency response is combined with actuator placement, to get even better results, with respect to the objective. The optimizer can efficiently place and remove material on places which contributes to the objective, while the force excitations at the same time help reducing unwanted behavior. The combination of optimizing both design variables was shown to be very effective in reducing a certain displacement field.

7.2 Recommendations

Although this thesis contributes to reduction of a certain displacement field, there are numerous of challenges to consider for future research. The implementation of design of supports is demonstrated in a static domain. It will be interesting to expand this implementation to a dynamic setting. The shown examples of bridge challenges are all based on static loads. If there is some traffic crossing this bridge however, some additional dynamic forces will be exerted on the road. This dynamic force should be included here. Another example of design of supports is shown for compliant mechanisms, also here, dynamics should be included to represent the physics better.

Design of force in a static domain is investigated. In these examples the supports remain at the same locations. It could be interesting to investigate the optimization process of both design of supports and design of actuators simultaneously. This could be helpful regarding design of compliant mechanism.

This thesis has shown reduction of a certain displacement field, some challenges need to be investigated, before this approach can be implemented for the design of accurate wafer stages. The model does not contain any damping, which should be implemented accordingly, in order to represent a physical example. The implementation of damping could lead to phase differences and additional behavior. This should be investigated also.

The case study in chapter 6 is made using a 2D element which is discretized by 40×20 elements. This discretization could be made much bigger. By enhancing this mesh, more details can be displayed, since the resolution becomes larger. This will lead unfortunately to a larger computational time, in order to solve the desired problem. Also the introduction of a third dimension should be helpful. Not only for better visual insight in the behavior, but also to make the model physically better interpretable. However, as we have already shown in A.4, this introduction will lead to an even larger computational time.

The existence of gray regions should be investigated even further, for example by taking a

bigger resolution. Also, looking for different filter techniques could be helpful towards this problem. Note that both solutions could lead to larger need of computational sources. Although the MMA solver seemed to help me quite well with my design problems, investigation of other solving techniques should be considered also.

A different interpolation scheme, for example the RAMP approach, could be investigated to achieve better black and white regions. Although the SIMP approach is a very effective interpolation scheme for solving static topology optimization problems, for dynamic cases the RAMP method results possibly in a better black and white solution.

There are some challenge regarding the actuator layout. Instead of creating a distributed force, it can be interesting to investigate the possibility to cluster forces (gradually) into a few points, in order to give a more realistic actuator placement.

The overfitting case as shown in A.8.2 and A.8.3 can maybe be solved by taking the optimal result from 6.5 and then gradually add different locations of actuator placement to this design. By using this different approach of solving, even better dynamic behavior could be achieved.

The dynamic force is implemented using a harmonic excitation. It can be interesting to investigate the optimal actuator placement for a transient response. This transient response could also lead to an investigation of a dynamic actuator pattern. Forces are turned on and off at a certain time. This will lead to an even larger computational load, but could be really helpful in the high-precision industry. Finally, Instead of looking for minimal displacements, it could be interesting to solve different objectives. For example using actuator placement and topology optimization to achieve a certain displacement field at a certain frequency, with a given weight.

Appendix A

Appendix

In this chapter all additional information for this research project can be found. First, the computer configuration is shown in (A.1).

This configuration is used to produce the wanted optimizations, which are represented by figures during this report. The numerical results of all these figures can be found in (A.2). For the first chapters convergence is shown, in order to get more insight in different optimization methods, which can be found in (A.3). A graphical representation of the implementation of a third dimension can be seen in (A.4).

The introduced arching continuation method is graphically represented by (A.5). Deformed geometry for some interesting examples are depicted in (A.6).

When dynamics are introduced, mode contributions can be found in (A.7). Additional examples of the wafer stage are described in (A.8).

A graphical representation of the difference between a SIMP and BESO method can be found in (A.9).

A.1 Computational setup

Although not explicitly documented, there are some numerical results available of all the executed optimizations.

These calculation results are derived from my personal computer. The associated computer and program specification can be found in A-1. For these numerical results, the draw and output options are set to disabled, to increase speed.

Program	MATLAB R2016b 64-bit (9.1)
Operating System	Windows 10 Pro 64-bit
CPU	Intel Core i7 @ 2.30GHz
RAM	16.0 GB DDR3 @ 1600MHz
Hard-Disc	Samsung 840 EVO 250GB SSD

Table A-1: Computer resources

A.2 Numerical results

First, let's have a look at the evolutionary example, and focus on the final result, as depicted in Figure 2-1e. The following parameters are here used, and will for this section considered as standard.

The chosen penalty $p = 3$, the mesh is discretized by 90 x 30 elements, the filter radius $r_{min} = 1.5$. The volume constraint is kept at 50% of the original design. Using this parameters, the following table can be made, just to get a clear vision on the numerical results. In A-2 the parameters from the associated example are depicted. Followed by the number of iterations, the optimization time (in seconds) and the final compliance. In the upcoming tables, some numerical results of the depicted examples in the report can be seen.

A.2.1 Chapter 2 results

Numerical results for chapter 2.

Parameter	Figure 2-1e
mesh	90 x 30
vol	0.5
p	3
r_{min}	1.5
iter	87
time	8.0
comp	188.9

Table A-2: Standard compliance example

Parameter	Figure 2-2b	Figure 2-2c	Figure 2-2d	Figure 2-2e
mesh	30 x 10	60 x 20	90 x 30	120 x 40
vol	0.5	0.5	0.5	0.5
p	3	3	3	3
r_{min}	1.5	1.5	1.5	1.5
iter	104	61	87	118
time	0.8	1.0	3.3	8.3
comp	219.7	195.0	188.9	185.5

Table A-3: Mesh refinement example

Parameter	Figure 2-3b	Figure 2-3c	Figure 2-3d	Figure 2-3e
mesh	90 x 30	90 x 30	90 x 30	90 x 30
vol	0.5	0.5	0.5	0.5
p	3	3	3	3
r_{min}	1.5	1.5	1.5	1.5
iter	187	106	87	179
time	7.3	4.1	3.3	7.0
comp	505.4	266.9	188.9	152.7

Table A-4: Volume fraction example

Parameter	Figure 2-4b	Figure 2-4c	Figure 2-4d	Figure 2-4e
mesh	90 x 30	90 x 30	90 x 30	90 x 30
vol	0.5	0.5	0.5	0.5
p	1	2	3	5
r_{min}	1.5	1.5	1.5	1.5
iter	15	158	87	187
time	0.9	6.0	3.3	7.3
comp	160.7	185.3	188.9	192.1

Table A-5: Penalty example

Parameter	Figure 2-6b	Figure 2-6c	Figure 2-6d	Figure 2-6e
mesh	90 x 30	90 x 30	90 x 30	90 x 30
vol	0.5	0.5	0.5	0.5
p	3	3	3	3
r_{min}	1.0	1.25	1.5	3.0
iter	54	158	87	66
time	2.4	6.2	3.3	2.5
comp	189.6	185.7	188.9	203.6

Table A-6: Filter example

A.2.2 Chapter 3 results

Numerical results for chapter 3.

Parameter	Figure 3-1c	Figure 3-1e
mesh	90 x 30	90 x 30
vol	0.5	0.5
p	3	3
r_{min}	1.5	1.5
iter	205	118
time	8.6	34.6
comp	196.0	195.0

Table A-7: OC vs MMA

Parameter	Figure 3-2	Figure 3-3
mesh	90 x 30	90 x 30
vol	0.5	0.5
p	3	3
r_{min}	1.0	1.25
iter	155	305
time	83.1	190.3
comp	288.3	227.2

Table A-8: Passive and active examples

Parameter	Figure 3-4b	Figure 3-4c
mesh	90 x 30	90 x 30
vol	0.5	0.5
p	3	3
r_{min}	1.5	1.5
iter	167	134
time	97.9	52.1
comp	121.3	227.2

Table A-9: Multiple load cases

Parameter	Figure 3-5b	Figure 3-5c	Figure 3-5d	Figure 3-5e
mesh	90 x 30	90 x 30	90 x 30	90 x 30
vol	0.5	0.5	0.5	0.5
p	5	5	5	5
r_{min}	1.5	1.5	1.5	1.5
ρ	0	$7.6 \cdot 10^{-5}$	$1.5 \cdot 10^{-4}$	$3.8 \cdot 10^{-4}$
iter	190	166	256	238
time	140.7	141.0	193.3	202.7
comp	193.4	318.5	440.9	844.2

Table A-10: Self-weight example

Parameter	Figure 3-6b	Figure 3-6c	Figure 3-6d
mesh	90 x 30	90 x 30	90 x 30
vol	0.5	0.5	0.5
p_{max}	3	3	3
r_{min}	1.5	1.5	1.5
iter	80	109	500
time	5.7	15.0	85.5
comp	190.2	18.1	179.3

Table A-11: Different filters example

Parameter	Figure 3-8b	Figure 3-8c	Figure 3-8d	Figure 3-8e
mesh	30 x 10 x 1	30 x 10 x 3	30 x 10 x 5	30 x 10 x 10
vol	0.5	0.5	0.5	0.5
p	3	3	3	3
r_{min}	1.5	1.5	1.5	1.5
iter	106	95	115	75
time	15.6	32.7	77.9	117.1
comp	124.0	49.6	25.0	13.6

Table A-12: 3D mesh refinement example

Parameter	Figure 3-11a	Figure 3-11b	Figure 3-13a	Figure 3-14a
mesh	80 x 80	80 x 80	120 x 120	120 x 120
vol	0.5	0.5	0.5	0.5
p	4	4	4	4
r_{min}	1.5	1.5	1.4	1.4
iter	1000	316	147	1000
time	114.42	31.93	342.1	253.2
d_{in}	891.35	18.42	76.48	75.21
d_{out}	-898.99	-40.89	-28.06	-30.48
G_d	-1.01	-2.22	-0.73	-0.81

Table A-13: Complaint mechanism example

A.2.3 Chapter 4 results

Numerical results for chapter 4.

Parameter	Figure 4-4b	Figure 4-4c	Figure 4-4d	Figure 4-4e
mesh	80 x 40	80 x 40	80 x 40	80 x 40
vol	0.2	0.2	0.2	0.2
p	3	3	3	3
r_{min}	1.5	1.5	1.5	1.5
vol_z	0.2	0.2	0.2	0.2
q	5	5	5	5
r_c	1	5	10	50
iter	157	211	200	192
time	43.8	58.2	68.8	79.2
comp	25921.6	45186.0	76166.3	83838.5

Table A-14: Optimal bridge

Parameter	Figure 4-5b	Figure 4-5c	Figure 4-5d	Figure 4-5e
mesh	80 x 40	80 x 40	80 x 40	80 x 40
vol	0.2	0.2	0.2	0.2
p	3	3	3	3
r_{min}	1.5	1.5	1.5	1.5
vol_z	0.2	0.2	0.2	0.2
q	5	5	5	5
r_c	1	5	10	20
iter	157	206	249	268
time	43.8	64.4	90.4	99.0
comp	25921.6	48837.2	77233.8	81539.9

Table A-15: Optimal bridge example 1

Parameter	Figure 4-6b	Figure 4-6c	Figure 4-6d	Figure 4-6e
mesh	80 x 40	80 x 40	80 x 40	80 x 40
vol	0.2	0.2	0.2	0.2
p	3	3	3	3
r_{min}	1.5	1.5	1.5	1.5
vol_z	0.2	0.2	0.2	0.2
q	5	5	5	5
r_c	1	3	5	10
iter	157	233	238	200
time	43.8	60.0	65.8	56.1
comp	25921.6	38811.0	52214.6	47032.7

Table A-16: Optimal bridge example 2

Parameter	Figure 4-7b	Figure 4-7c	Figure 4-7d	Figure 4-7e
mesh	80 x 80	80 x 80	80 x 80	80 x 80
vol	0.1	0.1	0.1	0.1
p	3	3	3	3
r_{min}	1.5	1.5	1.5	1.5
vol_z	0.2	0.2	0.2	0.2
q	5	5	5	5
r_c	1	2	4	6
iter	226	244	199	250
time	137.1	148.7	121.7	160.2
comp	28373.2	41730.5	34946.0	44885.9

Table A-17: Hanging bridge

Parameter	Figure 4-8b	Figure 4-8c	Figure 4-8d	Figure 4-8e
mesh	80 x 40	80 x 40	80 x 40	80 x 40
vol	0.2	0.2	0.2	0.2
p	3	3	3	3
r_{min}	1.5	1.5	1.5	1.5
vol_z	0.2	0.2	0.2	0.2
q	5	5	5	5
d_{tunnel}	32	32	53.3	53.3
iter	299	278	283	173
time	88.1	71.9	85.7	45.8
comp	31379.7	26120.6	36911.1	19059.3

Table A-18: Train tunnel example

Parameter	Figure 4-10b	Figure 4-11b
mesh	120 x 120	120 x 120
vol	0.2	0.2
p	3	3
r_{min}	1.5	1.5
vol_z	0.2	0.2
q	3	3
iter	135	145
time	327.5	262.5
d_{in}	14.15	52.23
d_{out}	-83.40	51.40
G_d	-5.89	1.97

Table A-19: Optimal compliant mechanisms

A.2.4 Chapter 5 results

Numerical results for chapter 5.

Parameter	Figure 5-1b
mesh	90 x 30
vol	1.0
p	3
r_{min}	1.5
F	-1.0
comp	44.3

Table A-20: Minimal compliance beam

Parameter	Figure 5-2	Figure 5-3a	Figure 5-3b
mesh	90 x 30	90 x 30	90 x 30
vol	1.0	1.0	1.0
p	3	3	3
r_{min}	1.5	1.5	1.5
F	-1.0	-1.0	-1.0
U	8.9	9.5	22.5

Table A-21: Simple cantilever beam

Parameter	Figure 5-4b	Figure 5-4c
mesh	90 x 30	90 x 30
vol	1.0	1.0
p	3	3
r_{min}	1.5	1.5
F	-1.0	-1.0
U	0.12	0.08

Table A-22: Triple fixed beam

Parameter	Figure 5-5b	Figure 5-6e	Figure 5-7b	Figure 5-8e
mesh	90 x 30	90 x 30	90 x 30	90 x 30
vol	0.25	0.30	0.24	0.30
p	3	3	3	3
r_{min}	1.5	1.5	1.5	1.5
F	-1.06	-1.05	-1.04	-1.01
U	0.13	45.73	106.46	41.20

Table A-23: Cantilever beam with topology optimization

A.2.5 Chapter 6 results

Numerical results for chapter 6.

Parameter	Figure 6-2	Figure 6-6	Figure 6-7
mesh	40 x 20	40 x 20	40 x 20
vol	1.0	1.0	1.0
p	3	3	3
r_{min}	1.5	1.5	1.5
ω^2	8	8	8
F_{tot}	3.20	3.20	3.20
F_{hor}	3.20	3.20	3.20
U	61.92	54.07	56.30

Table A-24: Dynamic solid beam

Parameter	Figure 6-8b	Figure 6-9b
mesh	40 x 20	40 x 20
vol	1.0	1.0
p	3	3
r_{min}	1.5	1.5
ω^2	8	8
F_{tot}	3.20	3.20
F_{hor}	3.20	3.20
U	53.68	53.52

Table A-25: Design of dynamic actuator placement

Parameter	Figure 6-10b	Figure 6-11b	Figure 6-12b
mesh	40 x 20	40 x 20	40 x 20
vol	1.0	1.0	1.0
p	3	3	3
r_{min}	1.5	1.5	1.5
ω^2	8	8	8
F_{tot}	3.20	2.41	3.20
F_{hor}	3.20	3.20	3.20
U	52.96	6.22	3.18

Table A-26: Design of dynamic actuator placement

Parameter	Figure 6-13	Figure 6-14a	Figure 6-14b
mesh	40 x 20	40 x 20	40 x 20
vol	0.72	0.66	0.68
p	3	3	4
r_{min}	1.5	1.5	1.5
ω^2	8	8	8
F_{tot}	3.20	2.16	2.16
F_{hor}	3.20	2.16	2.16
U	19.14	3.05	12.85

Table A-27: Dynamic actuator placement and topology

Parameter	Figure 6-14c	Figure 6-14d	Figure 6-15b
mesh	40 x 20	40 x 20	40 x 20
vol	0.67	0.65	0.78
p	5	6	6
r_{min}	1.5	1.5	1.5
ω^2	8	8	8
F_{tot}	2.17	2.13	2.51
F_{hor}	2.17	2.13	2.51
U	2.60	2.54	2.06

Table A-28: Dynamic actuator placement and topology

Parameter	Figure 6-16b	Figure 6-17b	Figure 6-18b
mesh	40 x 20	40 x 20	40 x 20
vol	0.73	0.90	0.74
p	6	6	6
r_{min}	1.5	1.5	1.5
ω^2	8	8	8
F_{tot}	2.35	2.89	2.89
F_{hor}	2.35	2.89	2.52
U	1.95	1.04	0.19

Table A-29: Dynamic actuator placement and topology

Parameter	Figure 6-19a	Figure 6-19b	Figure A-26a
mesh	40 x 20	40 x 20	40 x 20
vol	0.82	0.93	0.77
p	11	6	6
r_{min}	1.5	1.5	1.5
ω^2	8	8	8
F_{tot}	5.03	2.04	3.53
F_{hor}	2.68	3.01	2.51
U	1.02	1.37	1.59

Table A-30: Dynamic actuator additional cases

Parameter	Figure A-26b	Figure A-27a	Figure A-27b
mesh	40 x 20	40 x 20	40 x 20
vol	0.49	0.87	0.80
p	6	6	6
r_{min}	1.5	1.5	1.5
ω^2	8	4	16
F_{tot}	1.83	1.31	4.60
F_{hor}	1.51	1.49	5.23
U	15.28	0.98	3.19

Table A-31: Dynamic actuator additional cases

Parameter	Figure A-24	Figure A-25
mesh	40 x 20	40 x 20
vol	1.0	0.94
p	3	6
r_{min}	1.5	1.5
ω^2	8	8
F_{tot}	5.33	4.25
F_{hor}	4.19	3.00
U	13.71	1.95

Table A-32: Dynamic actuator overfitting cases

A.3 Convergence graph

To achieve more insight in the convergence process, for a number of examples, convergence graphs are plotted. The associated examples can be found in the legend of each picture. Using these graphs, some conclusions can be made regarding the need of using many iterations, which results into only a very minor benefit, with respect to the compliance.

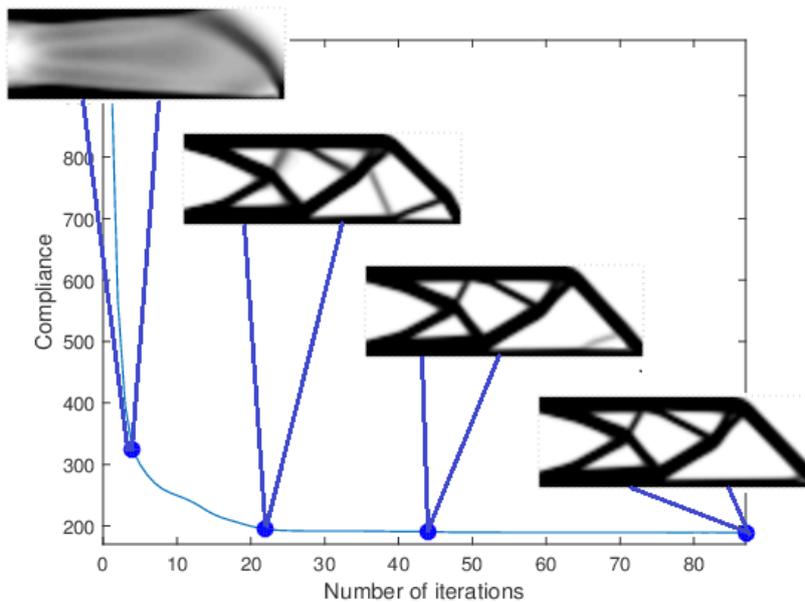


Figure A-1: Convergence plot of Figure 2-1

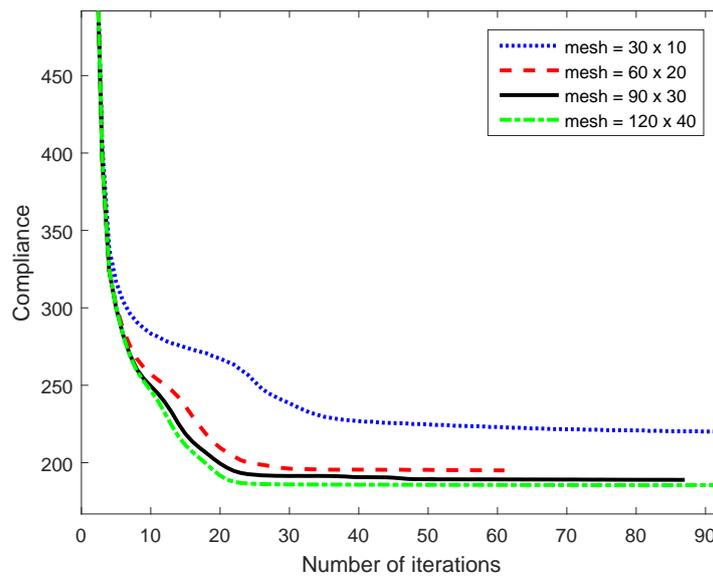


Figure A-2: Convergence plot of Figure 2-2

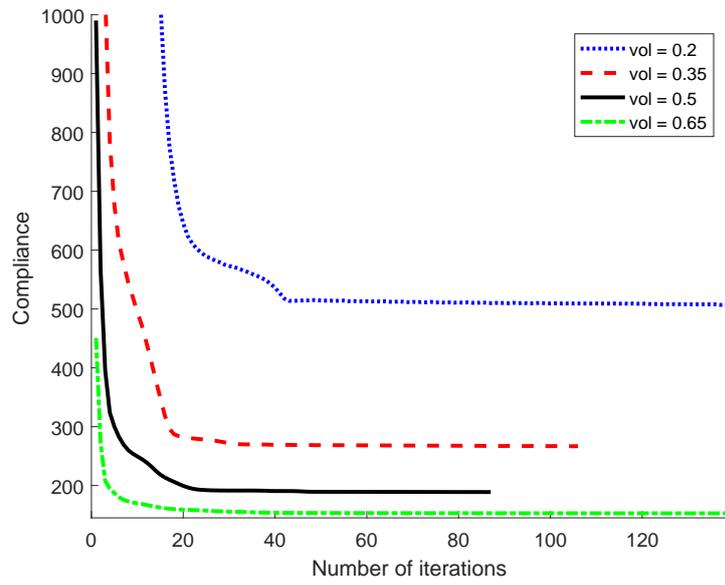


Figure A-3: Convergence plot of Figure 2-3

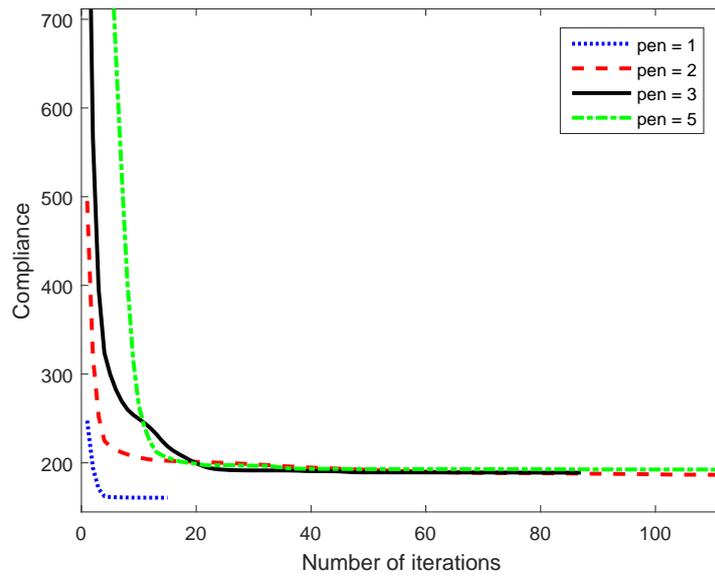


Figure A-4: Convergence plot of Figure 2-4

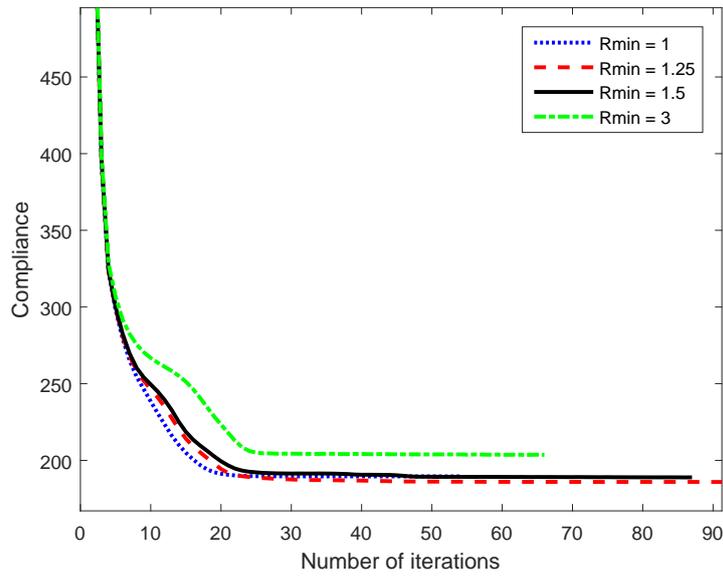


Figure A-5: Convergence plot of Figure 2-6

A.3.1 Chapter 3 graphs

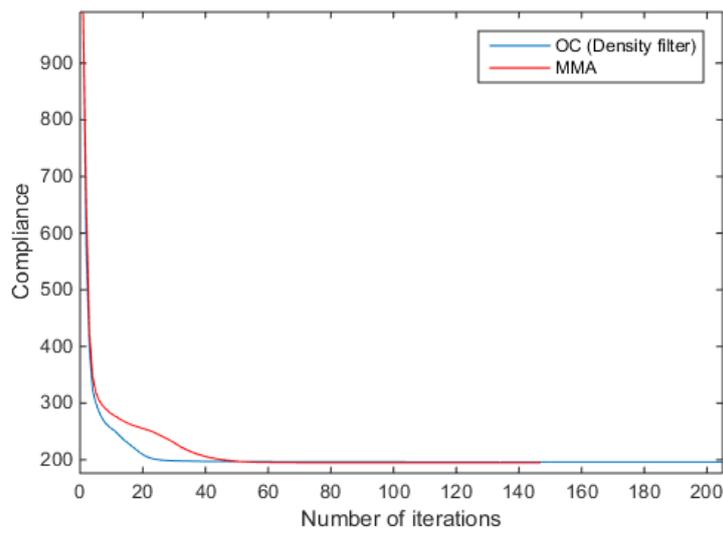


Figure A-6: Convergence plot of Figure 3-1c vs Figure 3-1e

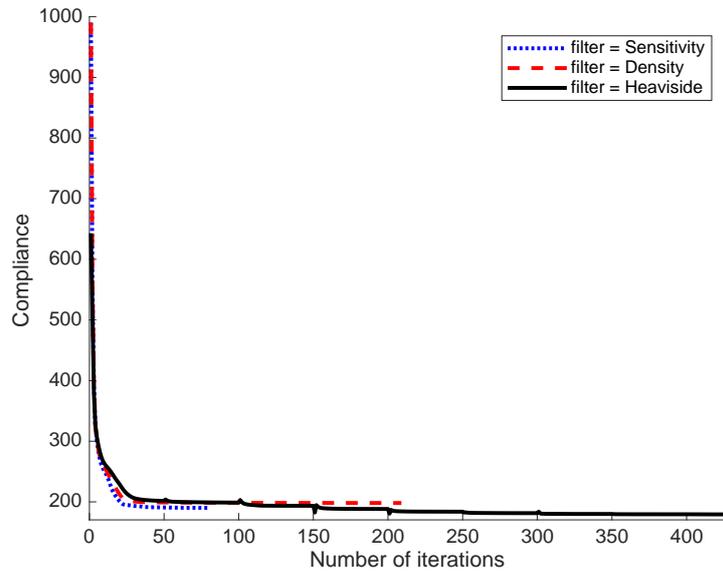


Figure A-7: Convergence plot of Figure 3-6

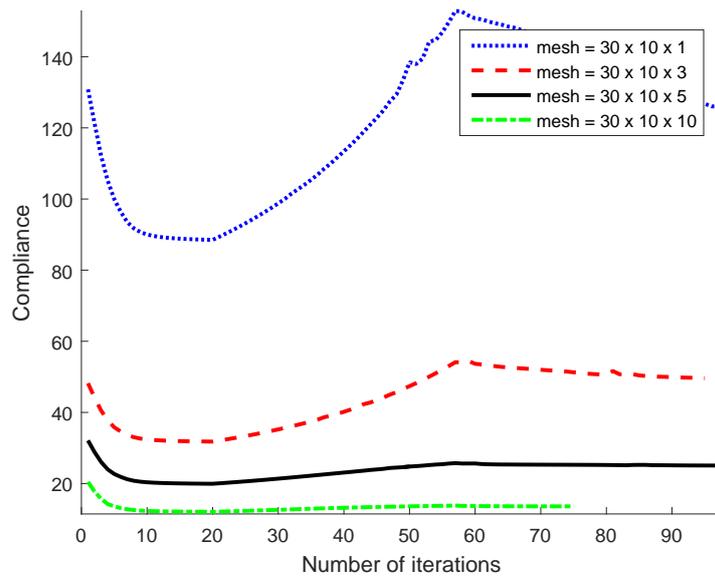


Figure A-8: Convergence plot of Figure 3-8

A.3.2 Chapter 4 graphs

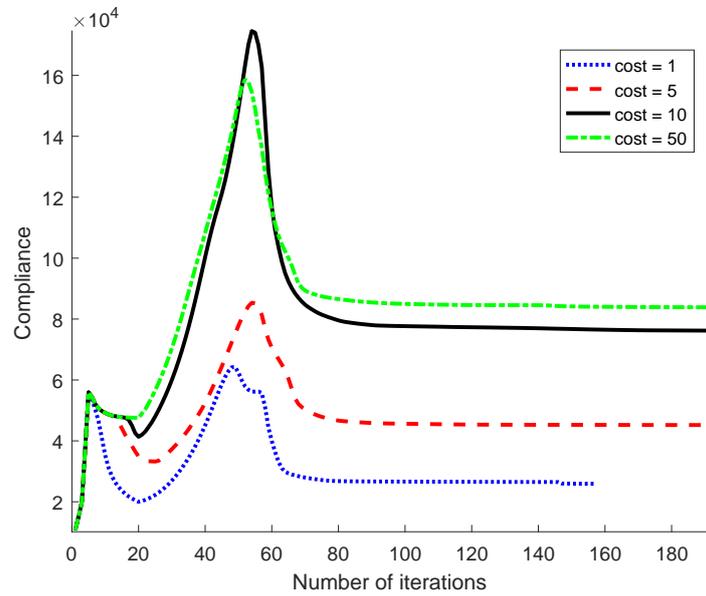


Figure A-9: Convergence plot of Figure 4-4

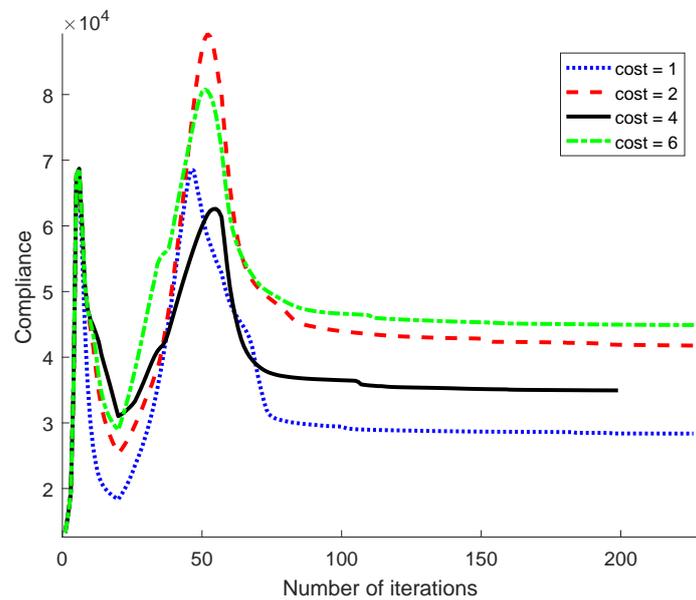


Figure A-10: Convergence plot of Figure 4-7

A.4 Computational graph

The third dimension is pretty shiny, but the computational time seems to increase exponentially, in this section a graph is shown, which includes a computational time comparison of a simple compliance problem, discretized by $30 \times 10 \times n_z$ elements. This number of lateral elements n_z is varied, to see the differences in computational time.

This timing example is done with two different types of outputs, namely, the No Output option (draw = 0, dis = 0) and the newly introduced Partial Output option (draw = 2, dis = 2).

After, an exponential fit seems to fit the best results. This is created using the Curve Fitting tool in MATLAB, after which this graph is made using the outputted parameters.

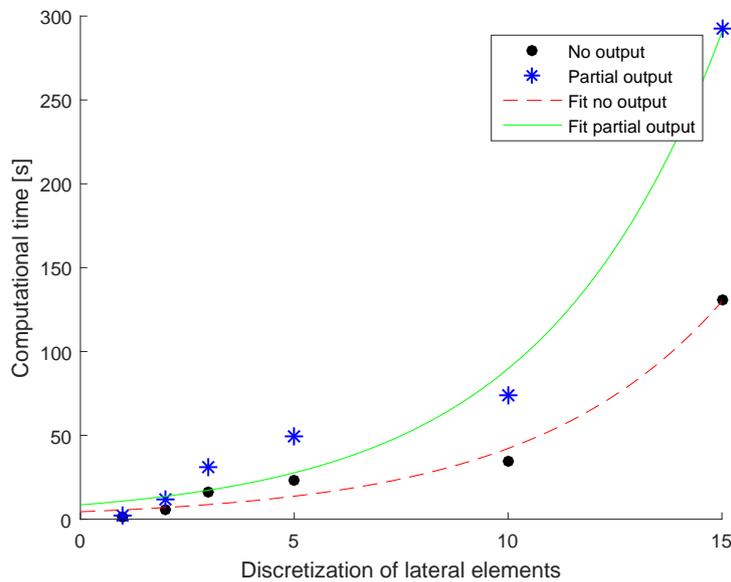


Figure A-11: Computational Example by variation of lateral elements.

A.5 Arching continuation

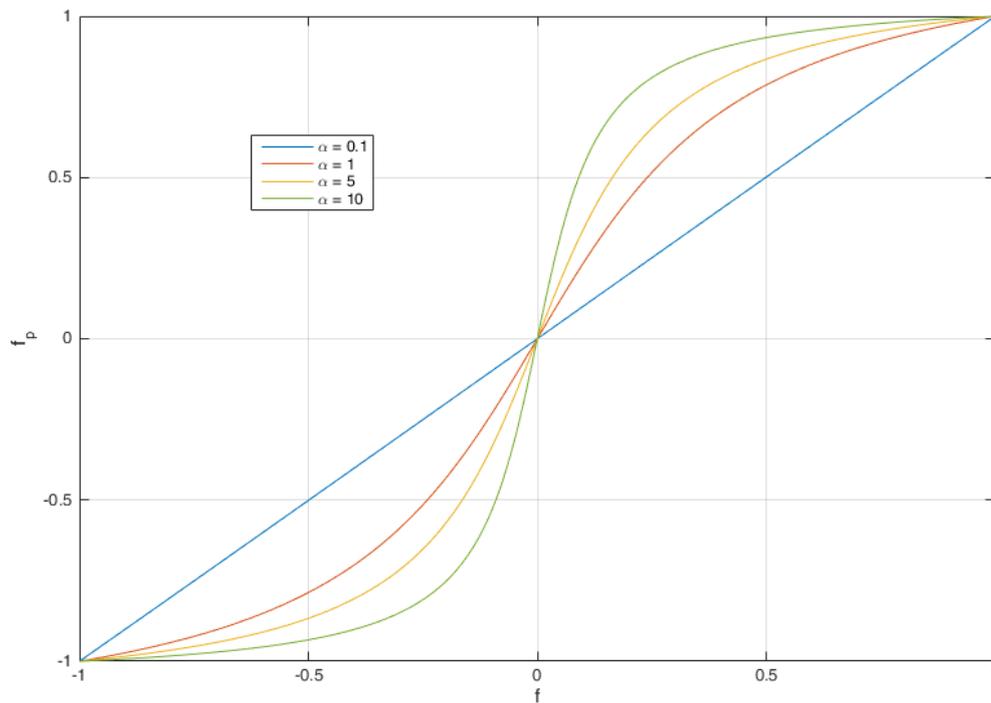


Figure A-12: Example of the arching continuation method. Equation 5-5 is displayed for different values of α .

A.6 Deformed geometry

This section displays the deformed geometry of calculated structures. The colors represent the associated displacements. The displacement of each element is calculated by taking an average of its eight surrounding node displacements.

A.6.1 Deformed triple fixed beam

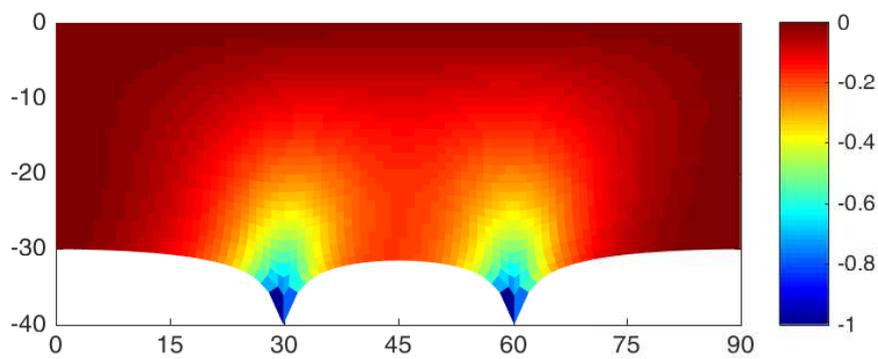


Figure A-13: Deformed geometry of Figure 5-4b. Displacements are normalized by taking the maximum absolute displacement as 1.

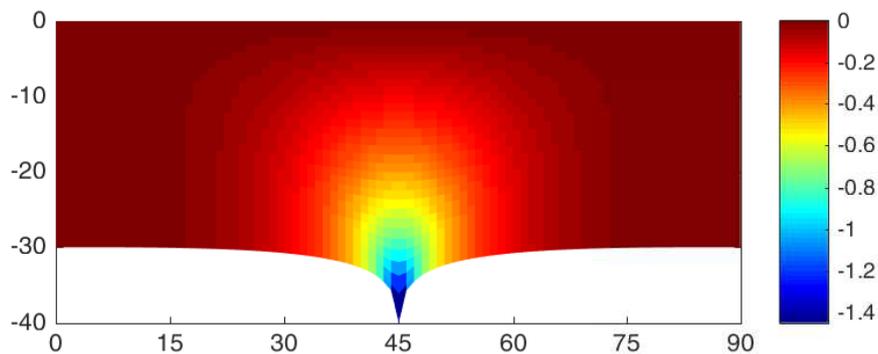
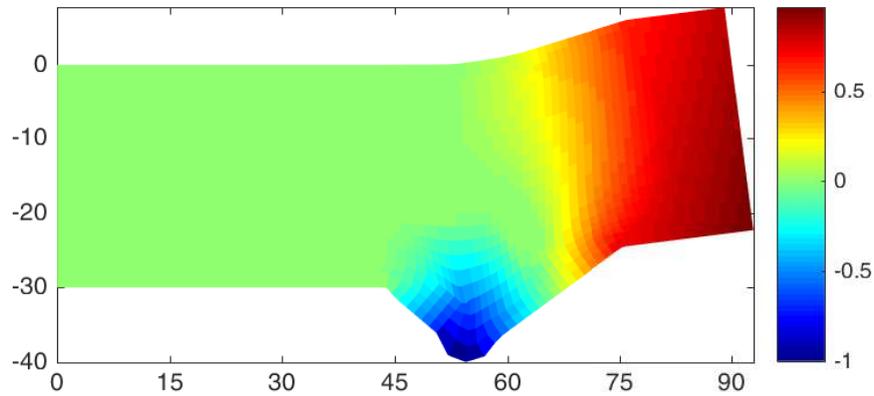
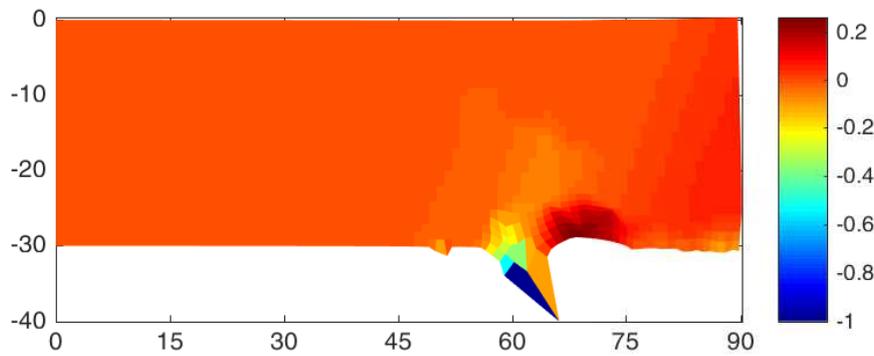


Figure A-14: Deformed geometry of Figure 5-4c. Displacements are normalized by taking the maximum absolute displacement of Figure A-13 as 1. The displacement in this figure are above 1, which means more displacement in the exerted area; however, the overall displacement is smaller than displayed in Figure A-13.

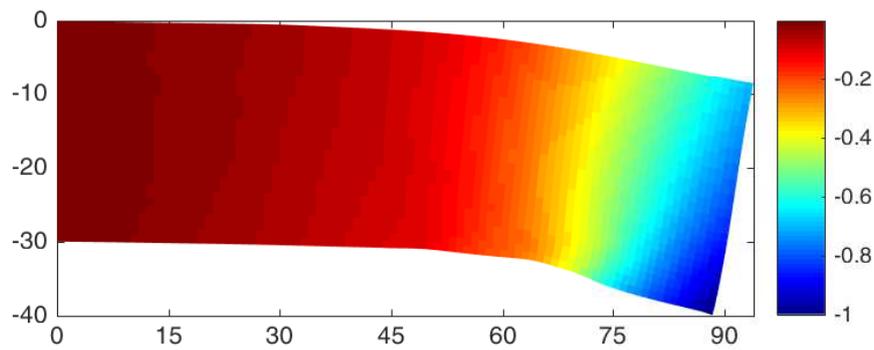
A.6.2 Deformed cantilever beam



(a) Deformed geometry of Figure 5-5a.



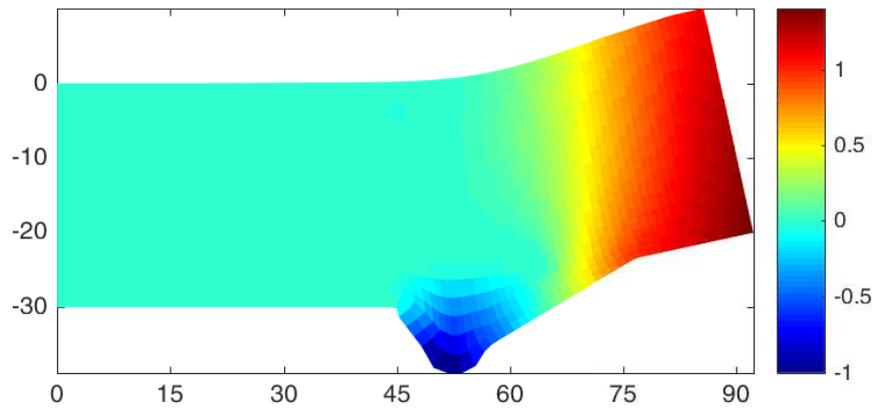
(b) Deformed geometry of Figure 5-5b.



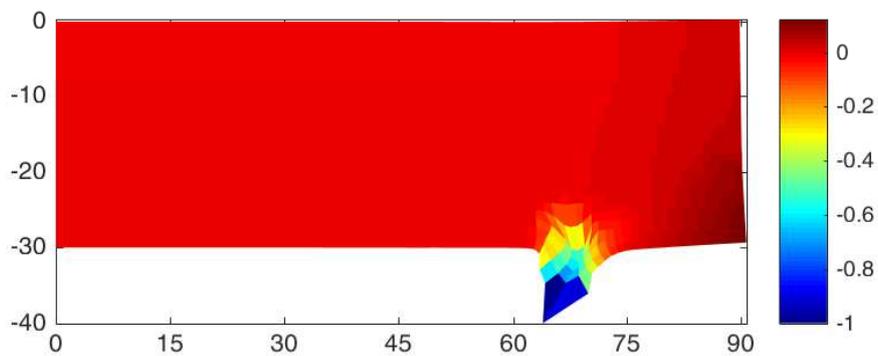
(c) Deformed geometry of Figure 5-6e.

Figure A-15: Deformed geometry of cantilever beam examples from Chapter 5. Displacements are normalized for each plot, by taking the maximum displacement of each structure as 1.

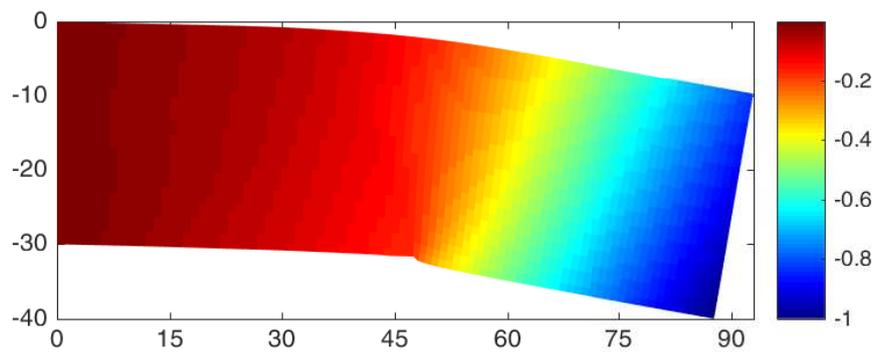
A.6.3 Deformed cantilever beam with density dependency



(a) Deformed geometry of Figure 5-7a.



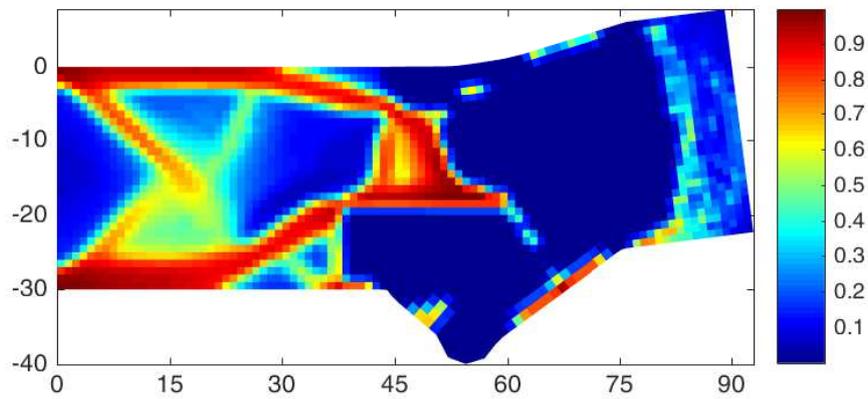
(b) Deformed geometry of Figure 5-7b.



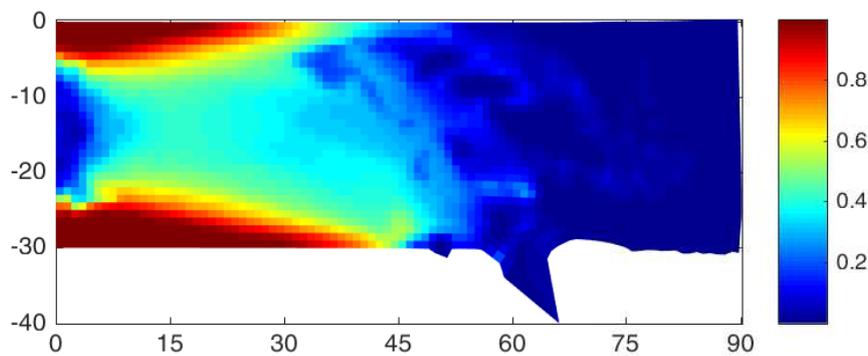
(c) Deformed geometry of Figure 5-8e.

Figure A-16: Deformed geometry of cantilever beam examples from Chapter 5. The optimal result is achieved using the object refinement from 5.4.3. Displacements are normalized for each plot, by taking the maximum displacement of each structure as 1.

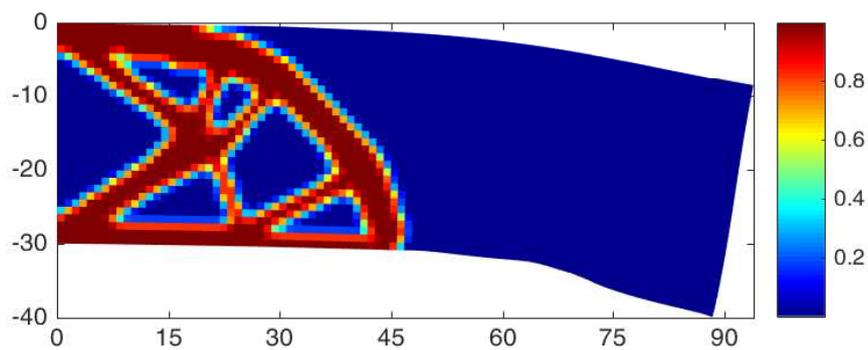
A.6.4 Deformed cantilever beam topology



(a) Deformed geometry of Figure 5-5a.



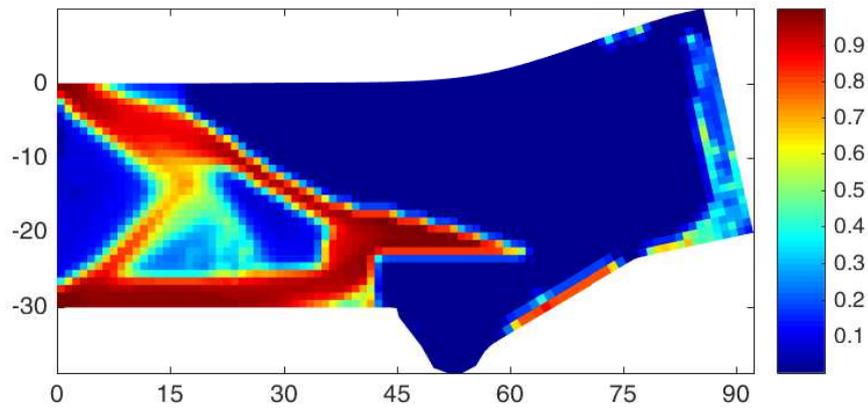
(b) Deformed geometry of Figure 5-5b.



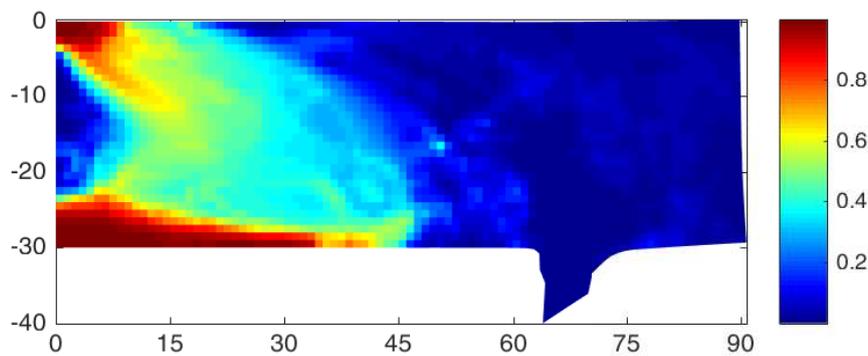
(c) Deformed geometry of Figure 5-6e.

Figure A-17: Deformed geometry of cantilever beam examples from Chapter 5. In these figures the topology is used as color reference, while the displacements represent deformed geometry of the design domain.

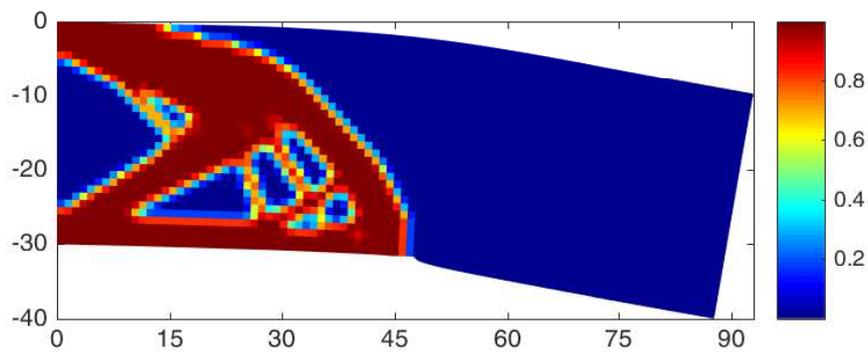
A.6.5 Deformed cantilever beam topology with density dependency



(a) Deformed geometry of Figure 5-7a.



(b) Deformed geometry of Figure 5-7b.



(c) Deformed geometry of Figure 5-8e.

Figure A-18: Deformed geometry of cantilever beam examples from Chapter 5. The optimal result is achieved using the object refinement from 5.4.3. In these figures the topology is used as color reference, while the displacements represent deformed geometry of the design domain.

A.7 Mode contribution

This section gives a tabular and graphical representation of the contribution of modes. The tables shows the mode contribution and some additional values, while the graphics display the mode dependency of a frequency spectrum.

A.7.1 Mode contribution tables

The mode contribution for several cases is displayed over here. In this first column the mode number can be seen, the second column holds the associated eigenfrequency. In the third column the mode contribution $\phi_i^T \mathbf{f}$, followed by the scaled contribution η_i , as described in (6-5). This is done, so the difference between scaling and the scaling of the mode can be seen very clearly. In the last column a weight factor of this mode influence can be found. This weight factor is normalized by taking the sum of these first twelve eigenmodes.

Mode	Eigenfrequency	$\phi_i^T \mathbf{f}$	Mode contribution η_i	Contribution (%)
Rigid #1	1.24e-07	0.20	-0.03	0.82
Rigid #2	1.34e-07	5.05	-0.63	20.40
Rigid #3	2.63e-07	0.07	-0.01	0.30
#1	2.29	0	0	0
#2	3.47	-7.36	-1.82	58.89
#3	3.83	0	0	0
#4	5.53	0	0	0
#5	5.60	0	0	0
#6	6.07	3.89	0.13	4.36
#7	6.15	7.44	0.25	8.07
#8	6.26	-3.82	-0.12	3.96
#9	7.19	4.33	0.10	3.20

Table A-33: Mode contribution of single force case as described in (Figure 6-2) and taking a frequency of $\omega^2 = 8$

Mode	Eigenfrequency	$\phi_i^T \mathbf{f}$	Mode contribution η_i	Contribution (%)
Rigid #1	1.24e-07	0.18	-0.02	0.78
Rigid #2	1.34e-07	5.06	-0.63	21.50
Rigid #3	2.63e-07	0.08	-0.01	0.32
#1	2.29	0	0	0
#2	3.47	-6.74	-1.67	56.87
#3	3.83	0	0	0
#4	5.53	0	0	0
#5	5.60	0	0	0
#6	6.07	7.21	0.25	8.50
#7	6.15	1.64	0.06	1.88
#8	6.26	-7.63	-0.25	8.34
#9	7.19	-2.32	-0.05	1.80

Table A-34: Mode contribution of two forces case as described in (Figure 6-6) and taking a frequency of $\omega^2 = 8$

Mode	Eigenfrequency	$\phi_i^T \mathbf{f}$	Mode contribution η_i	Contribution (%)
Rigid #1	1.24e-07	0.18	-0.02	0.85
Rigid #2	1.34e-07	5.06	-0.63	23.55
Rigid #3	2.63e-07	0.07	-0.01	0.35
#1	2.29	0	0	0
#2	3.47	7.15	1.77	66.01
#3	3.83	0	0	0
#4	5.53	0	0	0
#5	5.60	0	0	0
#6	6.07	-0.27	-0.01	0.34
#7	6.15	-5.38	-0.18	6.72
#8	6.26	-0.46	-0.01	0.55
#9	7.19	1.90	0.04	1.62

Table A-35: Mode contribution of distributed force case as described in (Figure 6-7) and taking a frequency of $\omega^2 = 8$

Mode	Eigenfrequency	$\phi_i^T \mathbf{f}$	Mode contribution η_i	Contribution (%)
Rigid #1	1.24e-07	4.97	-0.62	20.68
Rigid #2	1.34e-07	0.68	-0.08	2.83
Rigid #3	2.63e-07	63	-0.08	2.63
#1	2.29	-0.05	0.02	0.55
#2	3.47	6.87	1.7	56.65
#3	3.83	0.01	0	0.03
#4	5.53	-0.39	-0.02	0.57
#5	5.60	-0.18	-0.01	0.25
#6	6.07	-5.19	-0.18	5.98
#7	6.15	2.79	0.09	3.11
#8	6.26	5.54	0.18	5.92
#9	7.19	-1.05	-0.02	0.8

Table A-36: Mode contribution of distributed force case as described in (Figure 6-8) and taking a frequency of $\omega^2 = 8$

Mode	Eigenfrequency	$\phi_i^T \mathbf{f}$	Mode contribution η_i	Contribution (%)
Rigid #1	1.24e-07	0.18	-0.02	0.76
Rigid #2	1.34e-07	5.06	-0.63	21.25
Rigid #3	2.63e-07	.04	-0.01	0.19
#1	2.29	-0.05	0.02	0.6
#2	3.47	-6.8	-1.69	56.67
#3	3.83	0.1	0.01	0.49
#4	5.53	-0.99	-0.04	1.47
#5	5.60	0.53	0.02	0.76
#6	6.07	-6.06	-0.21	7.07
#7	6.15	-2.21	-0.07	2.5
#8	6.26	-6.45	-0.21	6.97
#9	7.19	-1.66	-0.04	1.27

Table A-37: Mode contribution of distributed force case as described in (Figure 6-9) and taking a frequency of $\omega^2 = 8$

Mode	Eigenfrequency	$\phi_i^T \mathbf{f}$	Mode contribution η_i	Contribution (%)
Rigid #1	1.24e-07	5.04	-0.63	15.39
Rigid #2	1.34e-07	0.46	-0.06	1.4
Rigid #3	2.63e-07	37	-0.05	1.14
#1	2.29	0.03	-0.01	0.31
#2	3.47	6.27	1.55	38.02
#3	3.83	0.92	0.14	3.41
#4	5.53	-5	-0.22	5.41
#5	5.60	-3.27	-0.14	3.42
#6	6.07	-15.17	-0.53	12.86
#7	6.15	-2.66	-0.09	2.18
#8	6.26	15.86	0.51	12.45
#9	7.19	-7.18	-0.16	4.02

Table A-38: Mode contribution of distributed force case as described in (Figure 6-10) and taking a frequency of $\omega^2 = 8$

Mode	Eigenfrequency	$\phi_i^T \mathbf{f}$	Mode contribution η_i	Contribution (%)
Rigid #1	1.24e-07	-0.99	0.12	1.5
Rigid #2	1.34e-07	5.13	-0.64	7.74
Rigid #3	2.63e-07	-2.99	0.37	4.52
#1	2.29	-1.35	0.49	5.89
#2	3.47	-6.62	-1.64	19.84
#3	3.83	6.59	0.99	12
#4	5.53	-12.09	-0.53	6.46
#5	5.60	-29.86	-1.28	15.44
#6	6.07	-22.62	-0.78	9.47
#7	6.15	6.36	-0.21	2.58
#8	6.26	14.08	0.45	5.46
#9	7.19	32.96	0.75	9.11

Table A-39: Mode contribution of distributed force case as described in (Figure 6-11) and taking a frequency of $\omega^2 = 8$

Mode	Eigenfrequency	$\phi_i^T \mathbf{f}$	Mode contribution η_i	Contribution (%)
Rigid #1	1.24e-07	5.05	-0.63	45.42
Rigid #2	1.34e-07	0.42	-0.05	3.78
Rigid #3	2.63e-07	33	-0.04	2.96
#1	2.29	-0.03	0.01	0.89
#2	3.47	0.06	0.01	1.03
#3	3.83	1.04	0.16	11.28
#4	5.53	-2.71	-0.12	8.61
#5	5.60	0.57	0.02	1.74
#6	6.07	1.7	0.06	4.25
#7	6.15	2.91	0.1	7.02
#8	6.26	4.9	0.16	11.32
#9	7.19	1.03	0.02	1.69

Table A-40: Mode contribution of distributed force case as described in (Figure 6-12) and taking a frequency of $\omega^2 = 8$

Mode	Eigenfrequency	$\phi_i^T \mathbf{f}$	Mode contribution η_i	Contribution (%)
Rigid #1	1.24e-07	6.62	-0.83	8.08
Rigid #2	1.34e-07	6.44	-0.81	7.86
Rigid #3	2.63e-07	.13	0.77	7.48
#1	2.29	2.14	-0.77	7.55
#2	3.47	-19.06	-4.72	46.13
#3	3.83	-2.03	-0.31	2.99
#4	5.53	6.23	0.28	2.69
#5	5.60	26.74	1.14	11.17
#6	6.07	0.45	0.02	0.15
#7	6.15	0.14	0	0.05
#8	6.26	9.58	0.31	3
#9	7.19	-12.74	-0.29	2.85

Table A-41: Mode contribution of distributed force case as described in (Figure A-24) and taking a frequency of $\omega^2 = 8$

Mode	Eigenfrequency	$\phi_i^T \mathbf{f}$	Mode contribution η_i	Contribution (%)
Rigid #1	1.24e-07	5.14	-0.43	20.26
Rigid #2	1.34e-07	1.39	-0.12	5.47
Rigid #3	2.63e-07	1.45	-0.12	5.72
#1	2.29	-0.05	0.01	0.34
#2	3.47	0	0.08	3.99
#3	3.83	-2.49	-0.96	45.66
#4	5.53	0.91	0.05	2.33
#5	5.60	0.12	0.01	0.29
#6	6.07	-0.24	-0.01	0.46
#7	6.15	3.11	0.12	5.74
#8	6.26	5.46	0.2	9.57
#9	7.19	-0.15	0	0.17

Table A-42: Mode contribution of distributed force case as described in (Figure A-23) and taking a actuation frequency very close to the second eigenfrequency, $\omega = 3.47$

A.7.2 Mode contribution graphics

A corresponding mode contribution for the six most important modes, over a spectrum of frequencies can be found here. In this schematic it can perfectly be seen which mode contributes how much on every frequency. When the excitation frequency approaches an eigenfrequency, the corresponding mode will be actuated the most and will thus take the most relative contribution of the total modes.

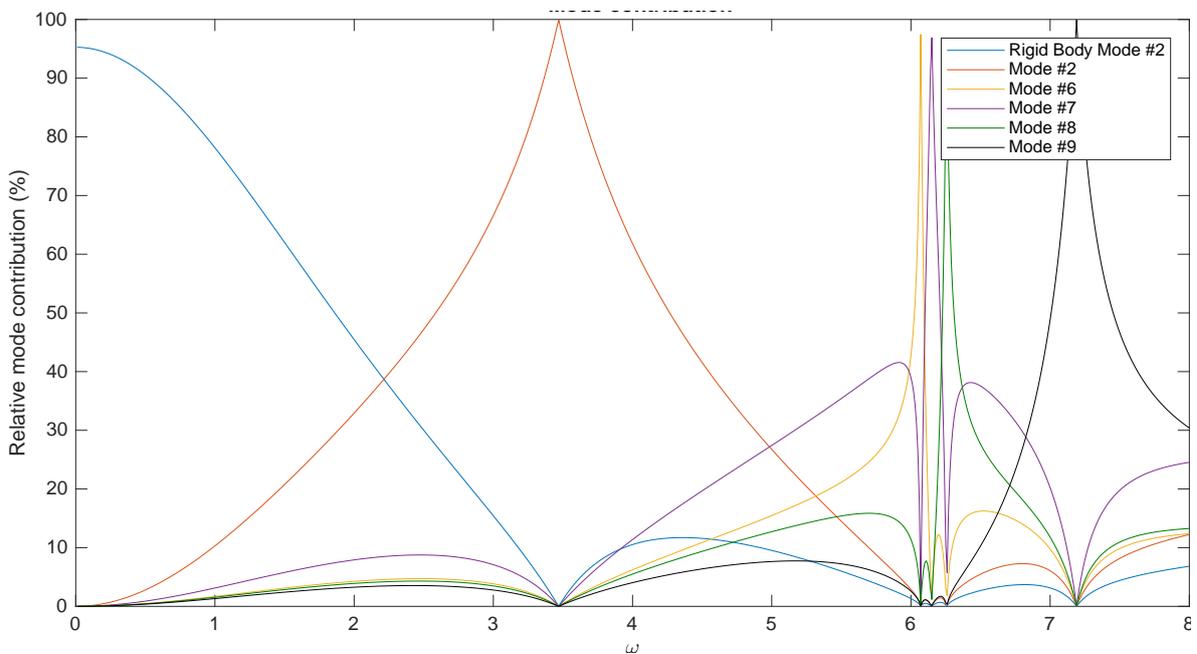


Figure A-19: Mode contribution for the six most important modes, using a single force case as depicted in (Figure 6-2) for a frequency spectrum.

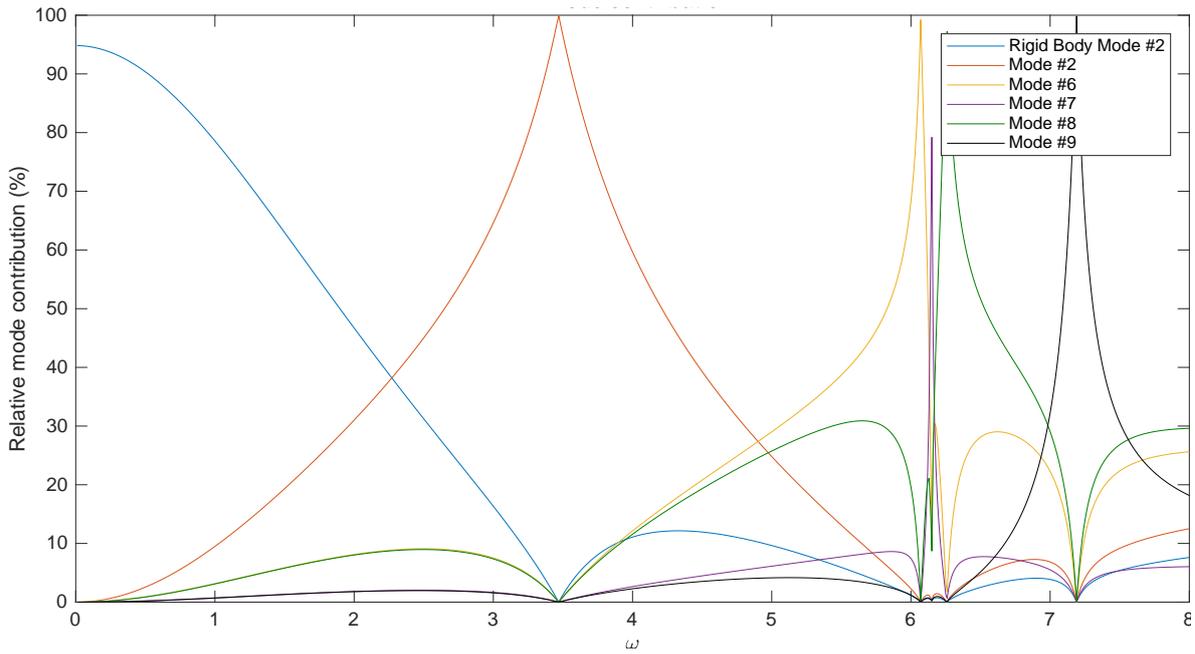


Figure A-20: Mode contribution for the six most important modes, using a two forces case as depicted in (Figure 6-6) for a frequency spectrum.

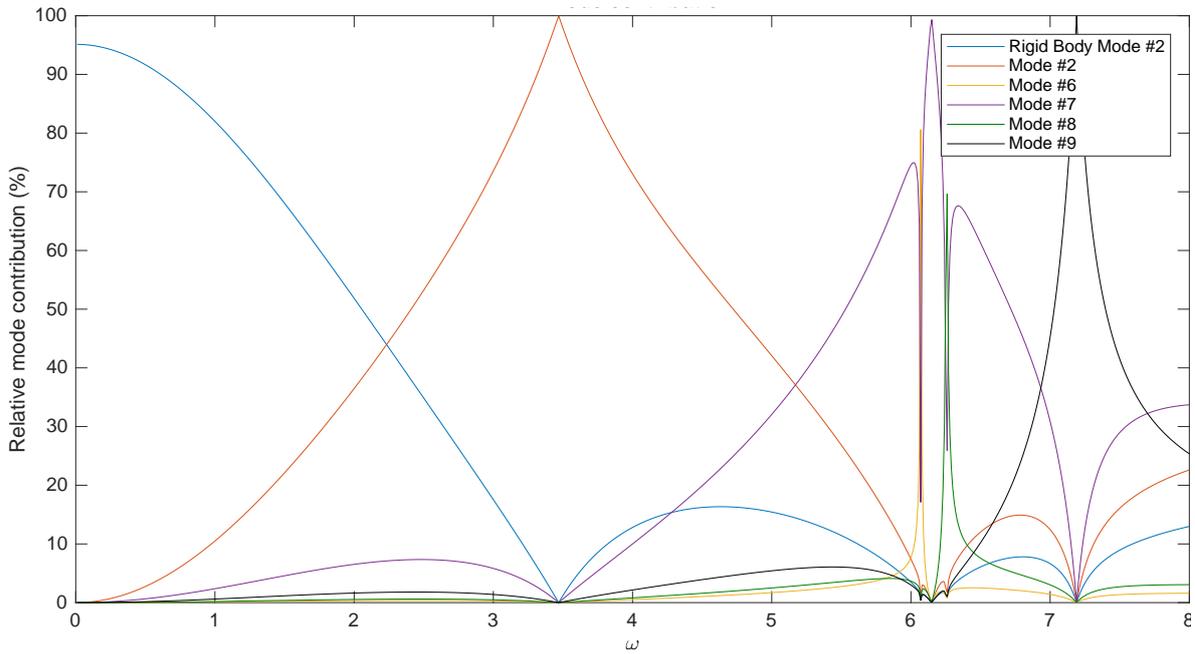


Figure A-21: Mode contribution for the six most important modes, using a distributed force case as depicted in (Figure 6-7) for a frequency spectrum.

A.7.3 Mode contribution progress plots

In this section a mode contribution progress plot can be found. This is basically a progress plot from the optimization problem depicted in Figure A-23a towards the optimal solution as depicted in Figure A-23b.

In this figure (Figure A-22) the (absolute) mode contribution $\phi_i^T \mathbf{f}$ is plotted over time. The mode contribution is plotted on a log-scale, for better visual reasons. As can be seen, the sum of the five most important modes is decreasing over time, although the second mode is increasing. This means, when iterations increasing, the total mode contribution is decreasing, which could benefit the objective value to minimize.

To help the optimizer a little bit, to win some time, after 20 iterations, the total placed force is scaled down to its minimum force value ($\mathbf{f} = m \cdot \mathbf{a}$). This manipulation process is only performed once during the optimization. The manipulation leads to a faster optimization result.

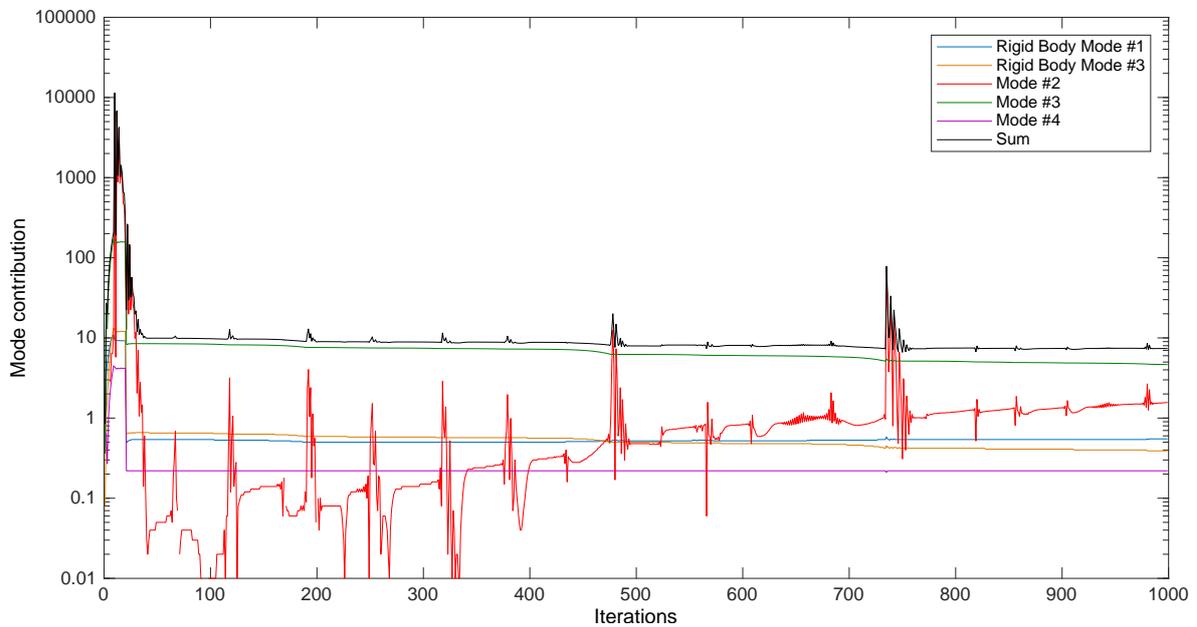


Figure A-22: Mode contribution of the five most important modes. The black line indicates the sum of the mode contribution $\phi_i^T \mathbf{f}$ of these five mode contributions values. These mode contributions are made using the optimal actuator layout as depicted in (Figure A-23).

A.8 Additional stage examples

In this section some additional variations of the stage examples from 6 are given. For references please review 6 to gain some knowledge on the background of these examples and problems.

A.8.1 Optimizing at eigenfrequency

Up to here, the actuation frequency ω was taken at a value $\omega^2 = 8$, which is just between the first and second eigenfrequencies. It is most of the time a good thing to refrain from actuating near or at a particular eigenfrequency. However, in some cases, it could be necessary to actuate near a certain eigenfrequency. For example, when the target frequency is close to an eigenfrequency. A change of the material distribution can be made, to change the dynamic response. If this change is not allowable, the only way to deal with this problem, is by changing the force actuation. This change of force layout can reduce or suppress certain dynamic behavior, to counteract that particular mode shape. This type of situation is investigated in this section.

In the example depicted in Figure A-23a the body is actuated at the second eigenfrequency. This second eigenfrequency seems to have a big influence on the total dynamic spectrum, so it is the most interesting frequency to investigate. The actuation frequency is very close to this eigenfrequency, because actuation exactly at the eigenfrequency is not solvable.

The same force design domain as explained in Figure 6-12 is used, so on the both sides, positive and negative forces are allowed. The optimal actuator layout can be found in Figure A-23b. Note that this particular eigenmode example has a big objective value. This is caused by the fact we are actuating almost at an eigenmode itself, which has very large displacements at that particular frequency. Additionally, the actuation frequency is increased, so more force is needed to fulfill this constraint. This is another reason why it is not a fair comparison to Figure 6-12b.

The left-hand side and right-hand side depicted in Figure A-23b show almost the same behavior, but some very little change in the values can be found. This is possibly caused by the very small interval between the eigenfrequency and the actuation frequency.

The optimization process starts with an initial distributed force on the left- and right-hand side of the design domain. The sum of this distributed force equals the minimum required force ($\mathbf{f} = m \cdot \mathbf{a}$). The optimizer then looking for an optimal force application. A schematic of this process can be found in Figure A-22.

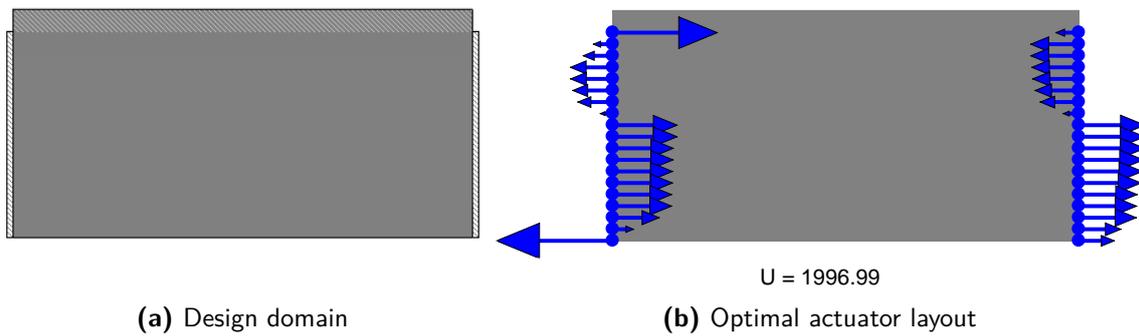


Figure A-23: Design domain and optimal actuator layout, while enabling negative forces design. The gray striped area indicates the objective area. The white striped area indicates the actuator design domain. The size, placement and direction of the arrows represent the location, magnitude and orientation of the optimized force layout. The actuation frequency is very close to the second eigenfrequency $\omega = 3.47$. The associated mode contribution can be found in Table A-42.

A.8.2 Overfitting design of actuators

The result of the optimization for the design domain as depicted in Figure 6-12a, while also enabling actuator design at the bottom of the body, in positive (upward) and negative (downward) direction is depicted in Figure A-24. Here, a big problem when optimizing this type of design problem, is the possibly overfitting of the model. The optimizer has just too much variables and the optimizer is more likely to approach a (high) local optimum. The result depicted in Figure A-24 shows a distribution along all sides of the design domain. The horizontal force is almost twice the minimum needed force to achieve the prescribed acceleration. This could also be a symptom of the overfitting of the model. It can be concluded that, in order to achieve a maximal optimization result, the design domain should not be too vaguely or too big.

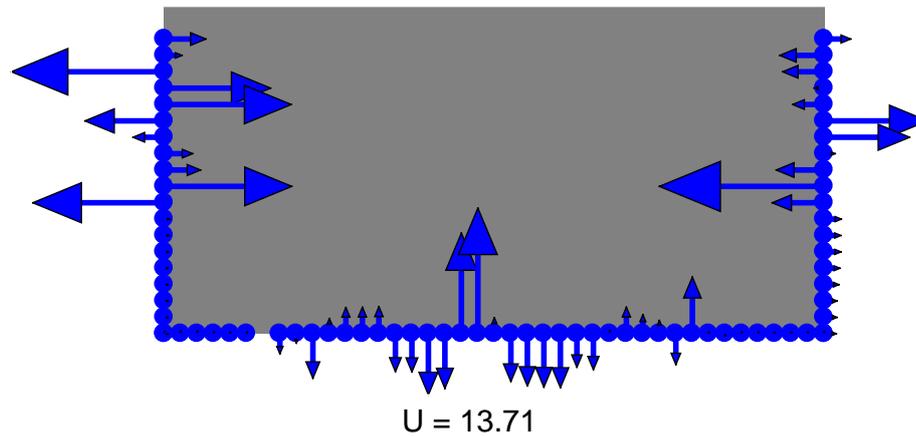


Figure A-24: Optimal actuator layout as depicted in Figure 6-12a, while also enabling actuator design at the bottom of the body, in positive (upward) and negative (downward) direction. The size and placement of the arrows represent the location and magnitude of the optimized force layout. The associated mode contribution can be found in Table A-41.

A.8.3 Overfitting design of actuators with topology optimization

The result of the optimization for the design domain as depicted in Figure 6-12a, while also enabling actuator design at the bottom of the body, in positive (upward) and negative (downward) direction and enabling topology optimization. The result is depicted in Figure A-25. Here, a big problem when optimizing this type of design problem, is the possibly overfitting of the model. The optimizer has just too much variables and the optimizer is more likely to approach a (high) local optimum. The result depicted in Figure A-25 shows a distribution along all sides of the design domain.

Some better results can be achieved, for example only using the left- and righthand side of the domain (Figure 6-17b). The result depicted over there is even better than the result depicted in Figure A-25.

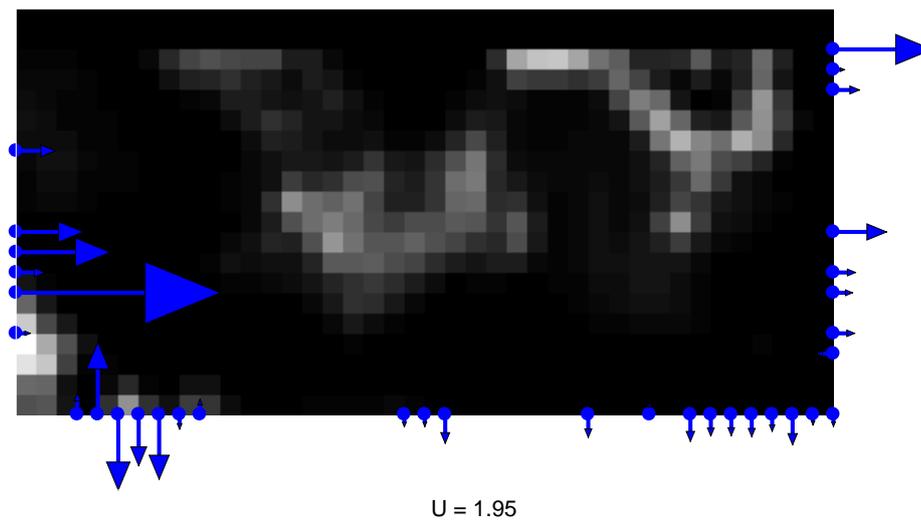


Figure A-25: Optimal actuator layout as depicted in Figure 6-12a, while also enabling actuator design at the bottom of the body, in positive (upward) and negative (downward) direction. The size and placement of the arrows represent the location and magnitude of the optimized force layout. The total horizontal force used is $\mathbf{f} = 3.00$.

A.8.4 Changing conditions representations

In this section some examples of changing conditions on the solved problems from 6 are given. These cases are described in 6.

For example changing the maximum volume:

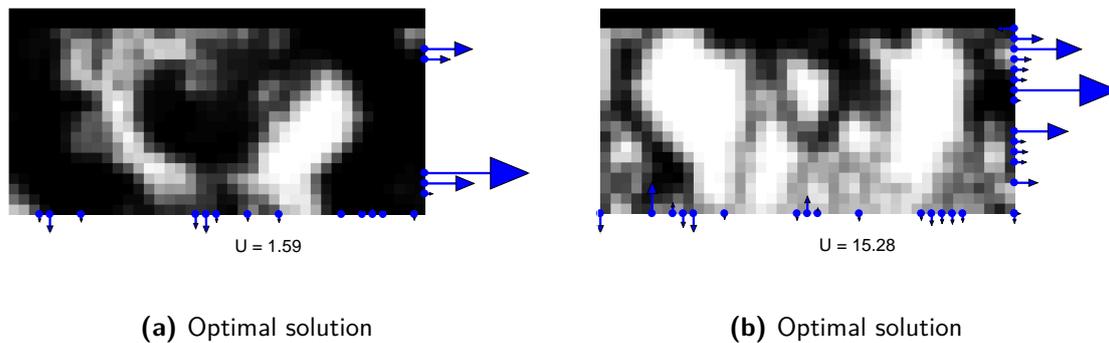


Figure A-26: Design domain and optimal actuator layout, while enabling negative forces design and using topology optimization with two different volume constraints, with a maximum volume of: a) 80%, b) 50%. The size, placement and direction of the arrows represent the location, magnitude and orientation of the optimized force layout.

And the solution for different actuation frequencies:

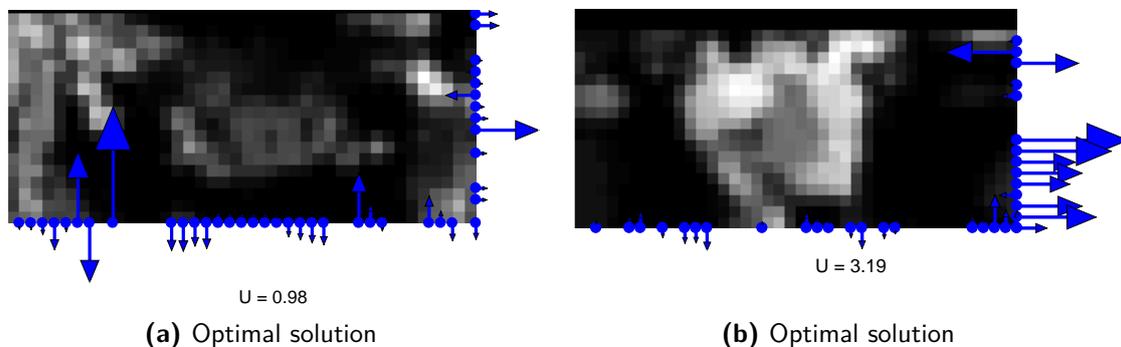


Figure A-27: Design domain and optimal actuator layout, while enabling negative forces design and using topology optimization for two different actuation frequencies: a) $\omega^2 = 4$, b) $\omega^2 = 16$. The size, placement and direction of the arrows represent the location, magnitude and orientation of the optimized force layout.

A.8.5 3D Extrusion

In this subsection however, a 3D extrusion is made. This extrusion is just a 2D lateral case in another dimension. A threshold value of 0.5 is chosen. This means all densities below this threshold value are displayed as void regions, for a better visual representation. As already explained in 6.1 this wafer stage should be used to produce circular wafers. The width and depth of the wafer should thus be the same size.

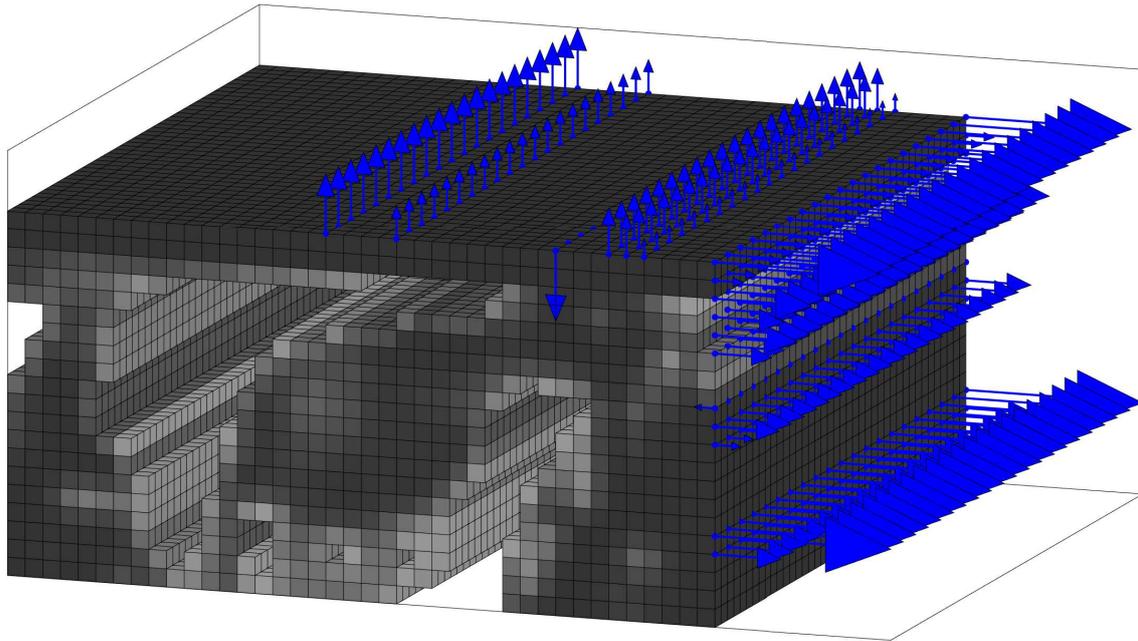


Figure A-28: A 2D lateral extrusion of the optimal wafer stage as depicted in Figure 6-18b.

A.9 Flowcharts

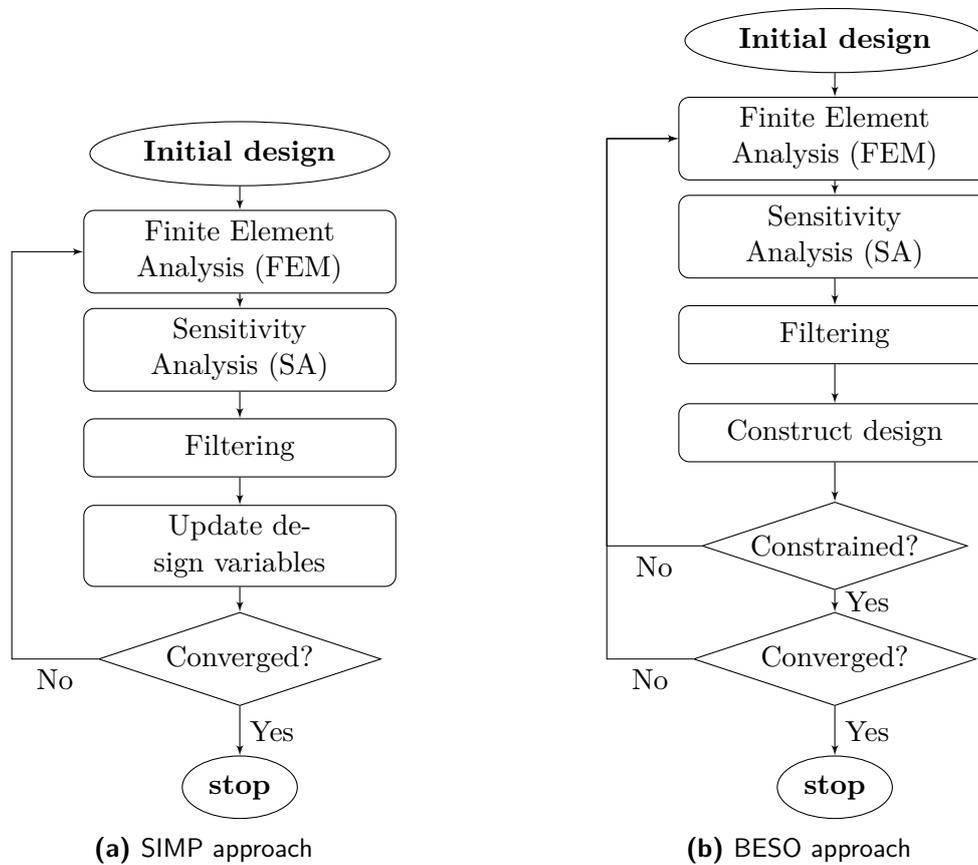


Figure A-29: Flowchart of topology optimization methods

Appendix B

Matlab Codes

In this and upcoming sections the created and used MATLAB codes are provided. In this Appendix the complete codes can be found. In Appendix C add-ins can be found. These add-ins can be used to create certain functionality. In Appendix D supplementary codes can be found.

In this section Basic.m (B.1) can be found. Using Basic.m a typical problem can be solved easily. All lines consists of helpful comments. The Advanced.m can be used to plot progression pictures, as shown in Chapter 1 of the Literature Survey. A big reference should be made to (Sigmund, 2001a) and (Andreassen et al., 2011) for providing a kick-start of the codes in this appendix.

By implementing the add-ins from (C.1) to (C.6), the final M-codes BASIC.m (B.3) and ADVANCED.m (B.2) can be made.

A third dimension could be added and so the matlab code also need some extensions. A simple add-in to extend 2D to 3D is made available in (C.7), with a reference to (Liu and Tovar, 2014). The final code of using the BASIC-code for three dimensional cases can be found in (B.4), and also, the ADVANCED version of this 3D code can be found in (B.5).

When dealing with compliant mechanisms. The BASIC-code needs to be updated using the simple add-in (C.8) for an inverter case. When producing a micro-gripper, one can also grab the final code immediately (B.6).

By making use of (C.9) a complete code of the implementation of design of supports can be made (B.7), with the associated advanced code for design of supports (B.8).

The implementation of design of actuator placement can be made (B.9) using the provided code. When also introducing topology optimization, one can grab the final code right away (B.10).

B.1 Basic.m

The working principle of the Basic-code will be explained in this section.

At first, it can be specified whether or not the Advanced.m code is used [line 15]. When this value is zero, the Basic-code continue as just one optimization problem, without any comparison calculations and plots. When this value equals zero, a number of design variables can be defined [line 16-23]. Some basic options for the calculation can be defined also [line 24-27]. The output options can be defined in [line 28-30], which can be used to gain some speed on the optimization process, as outputting and plotting can take a lot of time. The element properties can be defined [line 31-34]. The force and supports needs to be defined next [line 35-40].

From this line on, the user input is not necessary anymore, the elemental stiffness matrix is build up in [line 41-49], followed by the building of the nodes matrix [line 50-54]. To gain optimization speed a preparation scheme for the filter is made up [line 55-75]. After building up the load vector and some initialization [line 76-90] the main optimization loop starts at [line 91].

While the convergence of the optimization loop is above the minimum convergence, and the maximum number of iterations is not exceeded, the loop keeps running and assign a new loop number [line 92-93]. Each loop consists of a finite element analysis, where the stiffness matrix is built up and updated according to each node number, followed by an update of the element's displacements and associated compliance values [line 94-104]. Now, a sensitivity analysis is performed for each element and filtered accordingly. [line 105-107].

The design variables are updated using the Optimality Criteria method, where the Lagrangian multipliers for the volume constraint are calculated. Eventually, the design variable x is updated [line 108-122]. Each element value of x is stored in a massive matrix X , which contains each value of x for every iteration; the same holds for the compliance c [line 123-127]. The final results are displayed in the MATLAB command window, and the iterations and final result is plotted, if this is specified in the pre-amble [line 128-175]. For speed improvements, an additional option is made, to just optimize without iteration output and drawings [line 176-225]. The tic-toc commands displaying the total run time of the code [line 226].

```

1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %                                                                                                                                                                %
3  % Topology Optimization Using Matlab                                                                                                                                                                %
4  % BasicRep.m                                                                                                                                                                                        %
5  %                                                                                                                                                                %
6  % Delft University of Technology, Department PME                                                                                                                                                  %
7  % Master of Science Thesis Project                                                                                                                                                                %
8  %                                                                                                                                                                %
9  % Stefan Broxterman                                                                                                                                                                                %
10 %                                                                                                                                                                %
11 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
12 %
13 tic                                % start timer
14 %% DEFINE PARAMETERS
15 adv = 1;                            % use advanced function [0 = off, 1 = on]
16 if adv == 0                          % define parameters at behalf of the advanced
    function

```

```

17     nx = 90;           % number of elements horizontal
18     ny = 30;           % number of elements vertical
19     vol = 0.5;         % volume fraction [0-1]
20     pen = 3;           % penalty
21     rmin = 1.5;        % filter size
22     clc; clf; close all;
23 end
24 %% DEFINE CALCULATION
25 tol = 0.01;           % tolerance for convergence criterion [0.01]
26 move = 0.2;           % move limit for lagrange [0.2]
27 miter = 1000;        % maximum number of iterations [1000]
28 %% DEFINE OUTPUT
29 draw = 1;             % plot iterations [0 = off, 1 = on]
30 dis = 1;              % display iterations [0 = off, 1 = on]
31 %% DEFINE MATERIAL
32 E = 1;                % young's modulus of solid [1]
33 Emin = 1e-9;          % young's modulus of void [1e-9]
34 nu = 0.3;             % poisson ratio [0.3]
35 %% DEFINE FORCE
36 Fe = 2*(nx+1)*(ny+1); % element of force application [2*(nx+1)*(ny+1)]
37 Fn = 1;               % number of applied force locations [1]
38 Fv = -1;              % value of applied force [-1]
39 %% DEFINE SUPPORTS
40 fix = 1:2*(ny+1);     % fixed elements [1:2*(ny+1)]
41 %% PREPARE FINITE ELEMENT
42 N = 2*(nx+1)*(ny+1); % total element nodes
43 all = 1:2*(nx+1)*(ny+1); % all degrees of freedom
44 free = setdiff(all,fix); % free degrees of freedom
45 A11 = [12 3 -6 -3; 3 12 3 0; -6 3 12 -3; -3 0 -3 12]; % fem
46 A12 = [-6 -3 0 3; -3 -6 -3 -6; 0 -3 -6 3; 3 -6 3 -6]; % fem
47 B11 = [-4 3 -2 9; 3 -4 -9 4; -2 -9 -4 -3; 9 4 -3 -4]; % fem
48 B12 = [ 2 -3 4 -9; -3 2 9 -2; 4 9 2 3; -9 -2 3 2]; % fem
49 Ke = 1/(1-nu^2)/24*([A11 A12;A12' A11]+nu*[B11 B12;B12' B11]); % element
    stiffness matrix
50 nodes = reshape(1:(nx+1)*(ny+1),1+ny,1+nx); % create node numer matrix
51 dofvec = reshape(2*nodes(1:end-1,1:end-1)+1,nx*ny,1); % create dof vector
52 dofmat = repmat(dofvec,1,8)+repmat([0 1 2*ny+[2 3 0 1] -2 -1],nx*ny,1); %
    create dof matrix
53 iK = reshape(kron(dofmat,ones(8,1))',64*nx*ny,1); % build sparse i
54 jK = reshape(kron(dofmat,ones(1,8))',64*nx*ny,1); % build sparse j
55 %% PREPARE FILTER
56 iH = ones(nx*ny*(2*(ceil(rmin)-1)+1)^2,1); % build sparse i
57 jH = ones(size(iH)); % create sparse vector of ones
58 kH = zeros(size(iH)); % create sparse vector of zeros
59 m = 0; % index for filtering
60 for i = 1:nx % for each element calculate distance between ...
61     for j = 1:ny % elements' center for filtering
62         r1 = (i-1)*ny+j; % sparse value i
63         for k = max(i-(ceil(rmin)-1),1):min(i+(ceil(rmin)-1),nx) %
            center of element
64             for l = max(j-(ceil(rmin)-1),1):min(j+(ceil(rmin)-1),ny) %
                center of element
65                 r2 = (k-1)*ny+l; % sparse value 2

```

```

66         m = m+1; % update index for filtering
67         iH(m) = r1; % sparse vector for filtering
68         jH(m) = r2; % sparse vector for filtering
69         kH(m) = max(0,rmin-sqrt((i-k)^2+(j-l)^2)); % weight
           factor
70     end
71 end
72 end
73 end
74 H = sparse(iH,jH,kH); % build filter
75 Hs = sum(H,2); % summation of filter
76 %% DEFINE STRUCTURAL
77 x = repmat(vol,ny,nx); % initial material distribution
78 xF = x; % set filtered design variables
79 Fsiz = size(Fe,2); % size of load vector
80 F = sparse(Fe,Fn,Fv,N,Fsiz); % define load vector
81 %% PRE-ALLOCATE SPACE
82 npx = zeros(length(fix),1)'; % pre-allocate constraint dots
83 npy = zeros(length(fix),1)'; % pre-allocate constraint dots
84 npfx = zeros(length(Fe),1)'; % pre-allocate force dots
85 npfy = zeros(length(Fe),1)'; % pre-allocate force dots
86 U = zeros(size(F)); % pre-allocate space displacement
87 c = zeros(miter,1); % pre-allocate objective vector
88 %% INITIALIZE LOOP
89 iter = 0; % initialize loop
90 diff = 1; % initialize convergence criterion
91 %% START LOOP
92 while (diff > tol) && iter < miter % convergence criterion not met
93     iter = iter+1; % define iteration
94     p = pen; % set penalty
95     %% Finite element analysis
96     kK = reshape(Ke(:)*(Emin+xF(:)')^pen*(E-Emin),64*nx*ny,1); % create
           sparse vector k
97     K = sparse(iK,jK,kK); % combine sparse vectors
98     K = (K+K')/2; % build stiffness matrix
99     U(free,:) = K(free,free)\F(free,:); % displacement solving
100    c(iter) = 0; % set compliance to zero
101    Sens = 0; % set sensitivity to zero
102    %% Calculate compliance and sensitivity
103    c0 = reshape(sum((U(dofmat)*Ke).*U(dofmat),2),ny,nx); % initial
           compliance
104    c(iter) = c(iter) + sum(sum((Emin+xF.^p*(E-Emin)).*c0)); % calculate
           compliance
105    Sens = Sens -p*(E-Emin)*xF.^(p-1).*c0; % sensitivity
106    Senc = ones(ny,nx); % set constraint sensitivity
107    Sens(:) = H*(x(:).*Sens(:))./Hs./max(1e-3,x(:)); % update filtered
           sensitivity
108    %% Update design variables Optimality Criterion
109    l1 = 0; % initial lower bound for lagrangian mulitplier
110    l2 = 1e9; % initial upper bound for lagrangian multiplier
111    while (l2-l1)/(l1+l2) > 1e-3; % start loop
112        lag = 0.5*(l1+l2); % average of lagrangian interval

```

```

113     xnew = max(0,max(x-move,min(1,min(x+move,x.*sqrt(-Sens./Senc/lag)
114         )))); % update element densities
115     xF = xnew; % updated result
116     if sum(xF(:)) > vol*nx*ny; % check for optimum
117         l1 = lag; % update lower bound to average
118     else
119         l2 = lag; % update upper bound to average
120     end
121     diff = max(abs(xnew(:)-x(:))); % difference of maximum element change
122     x = xnew; % update design variable
123     %% Store results into database X
124     X(:, :, iter) = xF; % each element value x is stored for each
125         iteration
126     C(iter) = c(iter); % each compliance is stored for each iteration
127     assignin('base', 'X', X); % each iteration (3rd dimension)
128     assignin('base', 'C', C); % each iteration (3rd dimension)
129     %% Results
130     if dis == 1 % display iterations
131         disp([' Iter:' sprintf('%4i', iter) ' Obj:' sprintf('%10.4f', c(
132             iter)) ...
133             ' Vol:' sprintf('%6.3f', mean(xF(:))) ' Diff:' sprintf('%6.3f',
134                 diff)]);
135     end
136     if draw == 1 % plot iterations
137         figure(1)
138         subplot(2,1,1)
139         colormap(gray); imagesc(1-xF);
140         set(gca, 'XTick', [], 'YTick', [], 'XTicklabel', [], ...
141             'YTicklabel', [], 'xcolor', 'w', 'ycolor', 'w')
142         xlabel(sprintf('c = %.2f', c(iter)), 'Color', 'k')
143         drawnow;
144         hold on
145         if iter == 1
146             axis equal; axis tight;
147             % Plot coloured dots for constraints
148             for i = 1:length(fix)
149                 npx(i) = ceil(fix(i)/(2*(ny+1))) - 0.5;
150                 nplot = ceil(fix(i)/2);
151                 while nplot > (ny+1)
152                     nplot = nplot - (ny+1);
153                 end
154                 npy(i) = nplot - 0.5;
155             end
156             plot(npx, npy, 'r.', 'MarkerSize', 20)
157             % Plot coloured dots for force application
158             for i = 1:length(Fe)
159                 npfx(i) = ceil(Fe(i)/(2*(ny+1))) - 0.5;
160                 nplot = ceil(Fe(i)/2);
161                 while nplot > (ny+1)
162                     nplot = nplot - (ny+1);
163                 end
164                 npfy(i) = nplot - 0.5;

```

```

162         end
163         plot(npfx,npfy,'g.','MarkerSize',20)
164     end
165     % Plot compliance plot
166     figure(1)
167     subplot(2,1,2)
168     plot(c(1:iter))
169     xaxmax = c(iter);
170     yaxmax = max(c);
171     yaxmin = min(c(1:iter));
172     ylim([0.95*yaxmin yaxmax])
173     xlim([0 iter+10])
174 end
175 end
176 %% ONLY DISPLAY FINAL RESULT
177 if dis == 0 % display final result
178     disp([' Iter:' sprintf('%4i',iter) ' Obj:' sprintf('%10.4f',c) ...
179         ' Con:' sprintf('%6.3f',diff)]);
180 end
181 if draw == 0 % plot final result
182     figure(1)
183     subplot(2,1,1)
184     colormap(gray); imagesc(1-xF);
185     axis equal; axis tight;
186     set(gca,'XTick',[],'YTick',[],'XTicklabel',[],...
187         'YTicklabel',[],'xcolor','w','ycolor','w')
188     xlabel(sprintf('c = %.2f',c(iter)),'Color','k')
189     drawnow;
190     hold on
191     %% Plot coloured dots for constraints
192     for i = 1:length(fix)
193         npx(i) = ceil(fix(i)/(2*(ny+1)))-0.5;
194         nplot = ceil(fix(i)/2);
195         while nplot > (ny+1)
196             nplot = nplot-(ny+1);
197         end
198         npy(i) = nplot-0.5;
199     end
200     plot(npfx,npy,'r.','MarkerSize',20)
201     %% Plot coloured dots for force application
202     for i = 1:length(Fe)
203         npfx(i) = ceil(Fe(i)/(2*(ny+1)))-0.5;
204         nplot = ceil(Fe(i)/2);
205         while nplot > (ny+1)
206             nplot = nplot-(ny+1);
207         end
208         npfy(i) = nplot-0.5;
209     end
210     plot(npfx,npfy,'g.','MarkerSize',20)
211     %% Plot compliance plot
212     if adv == 0
213         figure(1)
214         subplot(2,1,2)

```

```
215     plot(c(1:iter))
216     xaxmax = c(iter);
217     yaxmax = max(c);
218     yaxmin = min(c(1:iter));
219     ylim([0.95*yaxmin yaxmax])
220     xlim([0 iter+10])
221 end
222 end
223 toc           % stop timer
```

B.2 ADVANCED.m

As an addition to (B.1), this advanced code can be used to plot progression pictures for multiple situations. In this Advanced code the optimization variables can be varied automatically. First, change the value of `adv` in `Basic.m` to one, in order to enable the program to vary the design variables. For upcoming add-ins the same Advanced function could be used at any time.

In the `Advanced.m` code the variation of the design variable needs to be chosen [line 13-14]. The next lines can be used to make vectors, which consist the values of the design variables, which are willing to be compared [line 15-21]. As up to now, it can only hold a maximum of four values per run. Only one variable can be varied per run, in order to hold the order variables constant, a default value can be defined in [line 22-28]. The program now write some values and pre-allocate spaces, user input is not needed from this line on [line 29-40]. The loop is starting, and makes a call to `Basic.m` for each design configuration, the programs determines whether or not the users made a design variation, or just want to plot an evolutionary scheme, as defined by `var = 6` [line 41-69]. The figures and progression plots are made in the coming lines [line 70-97]. Each calculation run time is collected and stored. After completion of each variation of the design, a matrix `Y` is displayed in the command windows. Which consist the number of run, the design variation vector, the number of loops needed for that configuration, and the associated objective and run time [line 98-124]. Compliance values are stored and plotted in one graph [line 125-211]. Next, values of the variations are stored into the workspace for further usage and finally the design problem is drawn [line 212-238].

```

1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %                                                                                                                                              %
3  % Topology Optimization Using Matlab                                                                 %
4  % ADVANCED.m                                                                                       %
5  %                                                                                                                                              %
6  % Delft University of Technology, Department PME                                                 %
7  % Master of Science Thesis Project                                                                %
8  %                                                                                                                                              %
9  % Stefan Broxterman                                                                               %
10 %                                                                                                                                              %
11 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
12 clc; clf; close all; clear X;
13 %% DEFINE OPTIMIZATION VARIABLES
14 var = 4;          % [1 = mesh, 2 = penalty, 3 = filter radius, 4 =
    volume fraction, 5 = filter method, 6 = evolution]
15 nxvec = [30,60,90,120]; % horizontal elements vector
16 nyvec = [10,20,30,40]; % vertical elements vector
17 volvec = [0.2 0.35 0.5 0.65]; % volume fraction vector
18 rminvec = [1,1.25,1.5,3]; % filter size vector
19 penvec = [1, 2, 3, 5]; % penalty vector
20 filvec = [0, 1, 2]; % filter vector
21 evolvec = [0.05, 0.25, 0.5, 1]; % evolution fraction vector
22 %% SET DEFAULT VALUES
23 nx = nxvec(3); % default number of horizontal elements
24 ny = nyvec(3); % default number of vertical elements
25 vol = volvec(3); % default number of volume fraction

```

```

26 pen = penvec(3);           % default penalty
27 rmin = rminvec(3);        % default filter radius
28 fil = filvec(1);          % default filter method
29 %% SET OPTIMIZATION VALUES
30 ex = [30,60,90,120];      % vector size for pre-allocating space
31 figend = 4;                % set total of varying values
32 label = ['a','b','c','d','e']; % graphic label
33 %% PRE-ALLOCATE SPACE
34 loops = zeros(1,size(ex,2)); % initial loops matrix
35 obj = zeros(1,size(ex,2)); % initial objective matrix
36 t = zeros(1,size(ex,2)); % initial time matrix
37 Y = zeros(size(ex,2),5); % initial results matrix
38 if var == 6                % for evolution scheme, BasicK.m only needs to
    ...
39     BASIC                  % run one time only
40 end
41 %% START LOOP
42 for fig = 1:figend        % start iteration loop
43     tic;                    % start timer
44     if var ~= 6            % for non-evolution scheme, run below
45         clear X; clear C; % clear results matrix for each run
46         if var == 1        % differentiation on number of elements
47             nx = nxvec(fig); % pick each horizontal value
48             ny = nyvec(fig); % pick each vertical value
49         elseif var == 2    % differentiation on penalty
50             pen = penvec(fig); % pick each penalty
51         elseif var == 3    % differentiation on filter radius
52             rmin = rminvec(fig); % pick each rmin
53         elseif var == 4    % differentiation on filter method
54             vol = volvec(fig); % pick each filter method
55         elseif var == 5    % differentiation on filter method
56             fil = filvec(fig); % pick each filter method
57         end
58         BASIC              % run Basic.m
59         loops(fig) = size(X,3); % number of iterations used
60         obj(fig) = c(iter); % store objective function
61         prog = X(:, :, loops(fig)); % store densities for progression
        drawing
62     elseif var == 6        % store compliance for evolution vector
63         loops = size(X,3); % for evolutionary scheme, calculate rounded
        ...
64         loop(1) = round(evolvec(1)*loops); % values of loops and store
        ...
65         loop(2) = round(evolvec(2)*loops); % this loop number
66         loop(3) = round(evolvec(3)*loops);
67         loop(4) = round(evolvec(4)*loops);
68         prog = X(:, :, loop); % progression picture for each evolution
        fraction
69     end
70     %% Set graphics
71     if draw == 1          % check for drawing
72         H = get(gcf, 'Position'); % get position of figure
73     else

```

```

74     H = [680,558,560,420]; % set size of figure(2) plot windows
75     end
76     H2 = figure(2); % plot window for progression pictures
77     set(H2, 'position', [H(1)+H(3) H(2) H(3) H(4)]); % place figure(2) next
78     to (1)
79     %% Draw progression plots
80     subplot(3,2,fig+2) % plot each differentiation
81     colormap(gray); % grayscale
82     if var == 6 % evolution needs different plotting
83         imagesc(1-prog(:, :, fig)); % plot progression picture
84         xlabel(sprintf('c = %.2f', C(loop(fig))), 'color', 'k')
85     else
86         imagesc(1-prog); % plot progression picture
87         xlabel(sprintf('c = %.2f', obj(fig)), 'color', 'k')
88     end
89     set(gca, 'XTick', [], 'YTick', [], 'XTicklabel', [], ...
90           'YTicklabel', [], 'xcolor', 'w', 'ycolor', 'w')
91     axis equal; axis tight; % set additional options
92     if var == 6 % evolution needs different plotting
93         xlabel(sprintf('c = %.2f', C(loop(fig))), 'color', 'k')
94     else
95         xlabel(sprintf('c = %.2f', obj(fig)), 'color', 'k')
96     end
97     ylabel(sprintf('%s', (label(fig+1))), ...
98           'rot', 0, 'color', 'k', 'FontSize', 11)
99     %% Store compliance
100    if var ~= 6 % store compliance for further plotting
101        if fig == 1
102            C1 = C;
103        elseif fig == 2
104            C2 = C;
105        elseif fig == 3
106            C3 = C;
107        elseif fig == 4
108            C4 = C;
109        end
110    end
111    %% Draw graphics
112    xbox = get(gca, 'XLim');
113    ybox = get(gca, 'YLim');
114    xwidth = xbox(2)-xbox(1);
115    ywidth = ybox(2)-ybox(1);
116    rectangle('Position', [xbox(1), ybox(1), xwidth, ywidth], ...
117            'EdgeColor', [0.5 0.5 0.5], 'LineStyle', ':'); drawnow;
118    t(fig) = toc;
119    %% Output
120    if var ~= 6 % output results for non-evolutionary schemes
121        Y(fig,:) = [fig ex(fig) loops(fig) obj(fig) t(fig)];
122        if fig == figend
123            Y
124        end;
125    end
126    %% Compliance graphs

```

```

126     if var ~= 6
127         H3 = figure(3);
128         set(H3, 'position', [H(1)-H(3) H(2) H(3) H(4)]); % place figure(2)
129         next to (1)
130     hold on
131     switch fig
132     case 1 % first variable
133         plot(1:length(C1), C1, 'b:', 'LineWidth', 2)
134         xaxmax = mean(length(C1));
135         yaxmax = max(max(C1));
136         yaxmin = min(C1);
137         if var == 1
138             legend(sprintf('mesh = %g x %g ', nxvec(1), nyvec(1)))
139         elseif var == 2
140             legend(sprintf('pen = %g', penvec(1)))
141         elseif var == 3
142             legend(sprintf('Rmin = %g', rminvec(1)))
143         elseif var == 4
144             legend(sprintf('vol = %g', volvec(1)))
145         elseif var == 5
146             legend(sprintf('filter = Sensitivity'))
147         end
148     case 2 % second variable
149         plot(1:length(C2), C2, 'r--', 'LineWidth', 2)
150         xaxmax = mean([length(C1) length(C2)]);
151         yaxmax = max([max(C1) max(C2)]);
152         yaxmin = min(min([C1 C2]));
153         if var == 1
154             legend(sprintf('mesh = %g x %g ', nxvec(1), nyvec(2)),
155                 sprintf('mesh = %g x %g ', nxvec(2), nyvec(2)))
156         elseif var == 2
157             legend(sprintf('pen = %g ', penvec(1)), sprintf('pen =
158                 %g', penvec(2)))
159         elseif var == 3
160             legend(sprintf('Rmin = %g ', rminvec(1)), sprintf('Rmin
161                 = %g', rminvec(2)))
162         elseif var == 4
163             legend(sprintf('vol = %g ', volvec(1)), sprintf('vol =
164                 %g', volvec(2)))
165         elseif var == 5
166             legend(sprintf('filter = Sensitivity'), sprintf('
167                 filter = Density'))
168         end
169     case 3 % third variable
170         plot(1:length(C3), C3, 'k', 'LineWidth', 2)
171         xaxmax = mean([length(C1) length(C2) length(C3)]);
172         yaxmax = max([max(C1) max(C2) max(C3)]);
173         yaxmin = min(min([C1 C2 C3]));
174         if var == 1
175             legend(sprintf('mesh = %g x %g ', nxvec(1), nyvec(2)),
176                 sprintf('mesh = %g x %g ', nxvec(2), nyvec(2)),
177                 sprintf('mesh = %g x %g ', nxvec(3), nyvec(3)))
178         elseif var == 2

```

```

171         legend(sprintf('pen = %g ',penvec(1)),sprintf('pen =
172                %g',penvec(2)),sprintf('pen = %g',penvec(3)))
173     elseif var == 3
174         legend(sprintf('Rmin = %g ',rminvec(1)),sprintf('Rmin
175                = %g',rminvec(2)),sprintf('Rmin = %g',rminvec(3))
176                )
177     elseif var == 4
178         legend(sprintf('vol = %g ',volvec(1)),sprintf('vol =
179                %g',volvec(2)),sprintf('vol = %g',volvec(3)))
180     elseif var == 5
181         legend(sprintf('filter = Sensitivity'),sprintf('
182                filter = Density'),sprintf('filter = Heaviside'))
183     end
184 case 4      % fourth variable
185 plot(1:length(C4),C4,'g-.','LineWidth',2)
186 xaxmax = mean([length(C1) length(C2) length(C3) length(C4
187                )]);
188 yaxmax = max([max(C1) max(C2) max(C3) max(C4)]);
189 yaxmin = min(min([C1 C2 C3 C4]));
190 if var == 1
191     legend(sprintf('mesh = %g x %g',nxvec(1),nyvec(2)),
192            sprintf('mesh = %g x %g',nxvec(2),nyvec(2)),
193            sprintf('mesh = %g x %g',nxvec(3),nyvec(3)),
194            sprintf('mesh = %g x %g',nxvec(4),nyvec(4)))
195 elseif var == 2
196     legend(sprintf('pen = %g ',penvec(1)),sprintf('pen =
197                %g',penvec(2)),sprintf('pen = %g',penvec(3)),
198            sprintf('pen = %g',penvec(4)))
199 elseif var == 3
200     legend(sprintf('Rmin = %g ',rminvec(1)),sprintf('Rmin
201                = %g',rminvec(2)),sprintf('Rmin = %g',rminvec(3))
202            ,sprintf('Rmin = %g',rminvec(4)))
203 elseif var == 4
204     legend(sprintf('vol = %g ',volvec(1)),sprintf('vol =
205                %g',volvec(2)),sprintf('vol = %g',volvec(3)),
206            sprintf('vol = %g',volvec(4)))
207 end
208 end
209 xlabel('Number of iterations')
210 ylabel('Compliance')
211 if exist('pcon','var') == 0
212     yaxmax = mean([yaxmin yaxmax]);
213 elseif pcon == 0
214     yaxmax = mean([yaxmin yaxmax]);
215 end
216 axis([0 xaxmax 0.95*yaxmin yaxmax])
217 elseif var == 6
218     H3 = figure(3);
219     set(H3,'position',[H(1)-H(3) H(2) H(3) H(4)]); % place figure(2)
220     next to (1)
221     hold on
222     plot(C)
223     xlabel('Number of iterations')

```

```
208         ylabel('Compliance')
209         axis([0 length(C) 0.9*min(C) max(C)])
210     end
211 end
212 %% STORE RESULTS
213 disp('Y = i, penalty, loops, objective, time')
214 if var == 1           % mesh refinement
215     Ymesh = Y;        % store result matrix
216     save('MeshRefinementY.mat','Y');
217 elseif var == 2      % penalty
218     Ypenal = Y;       % store result matrix
219     save('PenaltyY.mat','Y');
220 elseif var == 3      % filter radius
221     Yfilter = Y;     % store result matrix
222     save('FilterY.mat','Y');
223 elseif var == 4      % volume fraction
224     Yvolume = Y;     % store result matrix
225     save('VolumeY.mat','Y');
226 end
227 %% DRAW DESIGN PROBLEM
228 figure(2)
229 subplot(3,2,(1:2))   % plot the initial mechanical problem
230 rectangle('Position',[xbox(1),ybox(1),xwidth,ywidth],...
231         'FaceColor',[0.5 0.5 0.5])
232 axis equal; axis tight;
233 set(gca,'XTick',[],'YTick',[],'XTicklabel',[],...
234     'YTicklabel',[],'xcolor','w','ycolor','w')
235 ylabel(sprintf('%s',label(1)),'rot',0,'color','k','FontSize',11)
236 draw_arrow([xbox(2) ybox(1)],[xbox(2) -0.25*ywidth],1)
237 rectangle('Position',[-0.1*xwidth,ybox(1)-0.1*ywidth,...
238     0.1*xwidth,1.2*ywidth],'FaceColor',[0 0 0],'LineWidth',3)
```

B.3 BASIC.m

The final M-code, including all previous described functionality can be found here

```

1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %
3  % Topology Optimization Using Matlab
4  % BASIC.m
5  %
6  % Delft University of Technology, Department PME
7  % Master of Science Thesis Project
8  %
9  % Stefan Broxterman
10 %
11 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
12 %
13 tic % start timer
14 %% DEFINE PARAMETERS
15 adv = 1; % use advanced function [0 = off, 1 = on]
16 if adv == 0 % define parameters at behalf of the advanced
    function
17     nx = 90; % number of elements horizontal
18     ny = 30; % number of elements vertical
19     vol = 0.5; % volume fraction [0-1]
20     pen = 3; % penalty
21     rmin = 1.5; % filter size
22     fil = 0; % filter method [0 = sensitivity filtering, 1 =
        density filtering, 2 = heaviside filtering]
23     clc; clf; close all;
24 end
25 %% DEFINE SOLUTION METHOD
26 sol = 0; % solution method [0 = oc(sens), 1 = mma]
27 pcon = 0; % use continuation method [0 = off, 1 = on]
28 %% DEFINE CALCULATION
29 tol = 0.01; % tolerance for convergence criterion [0.01]
30 move = 0.2; % move limit for lagrange [0.2]
31 pcinc = 1.03; % penalty continuation increasing factor [1.03]
32 piter = 20; % number of iteration for starting penalty [20]
33 miter = 1000; % maximum number of iterations [1000]
34 %% DEFINE OUTPUT
35 draw = 1; % plot iterations [0 = off, 1 = on]
36 dis = 1; % display iterations [0 = off, 1 = on]
37 %% DEFINE MATERIAL
38 E = 1; % young's modulus of solid [1]
39 Emin = 1e-9; % young's modulus of void [1e-9]
40 nu = 0.3; % poisson ratio [0.3]
41 rho = 0e-3; % density [0e-3]
42 g = 9.81; % gravitational acceleration [9.81]
43 %% DEFINE FORCE
44 Fe = 2*(nx+1)*(ny+1); % element of force application [2*(nx+1)*(ny+1)]
45 Fn = 1; % number of applied force locations [1]
46 Fv = -1; % value of applied force [-1]
47 %% DEFINE SUPPORTS

```

```

48 fix = 1:2*(ny+1);           % fixed elements [1:2*(ny+1)]
49 %% DEFINE ELEMENT RESTRICTIONS
50 shap = 0;                   % [0 = no restrictions, 1 = circle, 2 = custom]
51 area = 0;                   % [0 = no material (passive), 1 = material (
    active)]
52 nodr = (round(ny/2)+(0:ny:(nx-1)*ny)); % custom restricted nodes
53 %% PREPARE FINITE ELEMENT
54 N = 2*(nx+1)*(ny+1);       % total element nodes
55 all = 1:2*(nx+1)*(ny+1);   % all degrees of freedom
56 free = setdiff(all,fix);    % free degrees of freedom
57 A11 = [12 3 -6 -3; 3 12 3 0; -6 3 12 -3; -3 0 -3 12]; % fem
58 A12 = [-6 -3 0 3; -3 -6 -3 -6; 0 -3 -6 3; 3 -6 3 -6]; % fem
59 B11 = [-4 3 -2 9; 3 -4 -9 4; -2 -9 -4 -3; 9 4 -3 -4]; % fem
60 B12 = [ 2 -3 4 -9; -3 2 9 -2; 4 9 2 3; -9 -2 3 2]; % fem
61 Ke = 1/(1-nu^2)/24*([A11 A12;A12' A11]+nu*[B11 B12;B12' B11]); % element
    stiffness matrix
62 nodes = reshape(1:(nx+1)*(ny+1),1+ny,1+nx); % create node numer matrix
63 dofvec = reshape(2*nodes(1:end-1,1:end-1)+1,nx*ny,1); % create dof vector
64 dofmat = repmat(dofvec,1,8)+repmat([0 1 2*ny+[2 3 0 1] -2 -1],nx*ny,1); %
    create dof matrix
65 iK = reshape(kron(dofmat,ones(8,1))',64*nx*ny,1); % build sparse i
66 jK = reshape(kron(dofmat,ones(1,8))',64*nx*ny,1); % build sparse j
67 %% PREPARE FILTER
68 iH = ones(nx*ny*(2*(ceil(rmin)-1)+1)^2,1); % build sparse i
69 jH = ones(size(iH)); % create sparse vector of ones
70 kH = zeros(size(iH)); % create sparse vector of zeros
71 m = 0; % index for filtering
72 for i = 1:nx % for each element calculate distance between ...
73     for j = 1:ny % elements' center for filtering
74         r1 = (i-1)*ny+j; % sparse value i
75         for k = max(i-(ceil(rmin)-1),1):min(i+(ceil(rmin)-1),nx) %
            center of element
76             for l = max(j-(ceil(rmin)-1),1):min(j+(ceil(rmin)-1),ny) %
                center of element
77                 r2 = (k-1)*ny+l; % sparse value 2
78                 m = m+1; % update index for filtering
79                 iH(m) = r1; % sparse vector for filtering
80                 jH(m) = r2; % sparse vector for filtering
81                 kH(m) = max(0,rmin-sqrt((i-k)^2+(j-l)^2)); % weight
                    factor
82             end
83         end
84     end
85 end
86 H = sparse(iH,jH,kH); % build filter
87 Hs = sum(H,2); % summation of filter
88 %% DEFINE ELEMENT RESTRICTIONS
89 x = repmat(vol,ny,nx); % initial material distribution
90 if shap == 0 % no restrictions
91     efree = (1:nx*ny)'; % all elements are free
92     eres = []; % no restricted elements
93 elseif shap == 1 % restrictions
94     rest = zeros(ny,nx); % pre-allocate space

```

```

95     for i = 1:nx           % start loop
96         for j = 1:ny       % for each element
97             if sqrt((j-ny/2)^2+(i-nx/4)^2) < ny/2.5 % circular
                    restriction
98                 rest(j,i) = 1; % write restriction
99                 if rest(j,i) == area % check for restriction
100                     x(j,i) = area; % store restrictions in material
                            distribution
101             end
102         end
103     end
104 end
105 efree = find(rest ~= 1); % set free elements
106 eres = find(rest == 1); % set restricted elements
107 end
108 if fil == 0 || fil == 1 % sensitivity, density filter
109     xF = x; % set filtered design variables
110 elseif fil == 2 % heaviside filter
111     beta = 1; % hs filter
112     xTilde = x; % hs filter
113     xF = 1-exp(-beta*xTilde)+xTilde*exp(-beta); % set filtered design
        space
114 end
115 xFree = xF(efree); % define free design matrix
116 %% DEFINE STRUCTURAL
117 Fsiz = size(Fe,2); % size of load vector
118 F = sparse(Fe,Fn,Fv,N,Fsiz); % define load vector
119 %% DEFINE MMA PARAMETERS
120 m = 1; % number of constraint functions
121 n = size(xFree(:,1)); % number of variables
122 xmin = zeros(n,1); % minimum values of x
123 xmax = ones(n,1); % maximum values of x
124 xold1 = zeros(n,1); % previous x, to monitor convergence
125 xold2 = xold1; % used by mma to monitor convergence
126 df0dx2 = zeros(n,1); % second derivative of the objective function
127 dfdx2 = zeros(1,n); % second derivative of the constraint function
128 low = xmin; % lower asymptotes from the previous iteration
129 upp = xmax; % upper asymptotes from the previous iteration
130 a0 = 1; % constant a_0 in mma formulation
131 a = zeros(m,1); % constant a_i in mma formulation
132 cmma = 1e3*ones(m,1); % constant c_i in mma formulation
133 d = zeros(m,1); % constant d_i in mma formulation
134 subs = 200; % maximum number of subsolv iterations
135 %% PRE-ALLOCATE SPACE
136 npx = zeros(length(fix),1)'; % pre-allocate constraint dots
137 npy = zeros(length(fix),1)'; % pre-allocate constraint dots
138 npfx = zeros(length(Fe),1)'; % pre-allocate force dots
139 npfy = zeros(length(Fe),1)'; % pre-allocate force dots
140 U = zeros(size(F)); % pre-allocate space displacement
141 c = zeros(miter,1); % pre-allocate objective vector
142 %% INITIALIZE LOOP
143 iter = 0; % initialize loop
144 diff = 1; % initialize convergence criterion

```

```

145 loopbeta = 1;           % initialize beta-loop
146 %% START LOOP
147 while ((diff > tol) || (iter < piter+1)) && iter < miter % convergence
    criterion not met
148     loopbeta = loopbeta + 1; % iteration loop for hs filter
149     iter = iter+1;          % define iteration
150     if pcon == 1           % use continuation method
151         if iter <= piter % first number of iterations...
152             p = 1;        %... set penalty 1
153         elseif iter > piter % after a number of iterations...
154             p = min(pen,pcinc*p); % ... set continuation penalty
155         end
156     elseif pcon == 0      % not using continuation method
157         p = pen;         % set penalty
158     end
159 %% Selfweight
160     if rho ~= 0          % gravity is involved
161         xP=zeros(ny,nx); % pre-allocate space
162         xP(xF>0.25) = xF(xF>0.25).^p; % normal penalization
163         xP(xF<=0.25) = xF(xF<=0.25).*(0.25^(p-1)); % below pseudo-density
164         Fsw = zeros(N,1); % pre-allocate self-weight
165         for i=1:nx*ny % for each element, set gravitational...
166             Fsw(dofmat(i,2:2:end))=Fsw(dofmat(i,2:2:end))-xF(i)*rho
                *9.81/4;
167         end % force to the attached nodes
168         Fsw=repmat(Fsw,1,size(F,2)); % set self-weight for load cases
169     elseif rho == 0      % no gravity
170         xP = xF.^p;      % penalized design variable
171         Fsw = 0;         % no selfweight
172     end
173     Ftot = F + Fsw;     % total force
174 %% Finite element analysis
175     kK = reshape(Ke(:)*(Emin+xP(:)')*(E-Emin)),64*nx*ny,1); % create
        sparse vector k
176     K = sparse(iK,jK,kK); % combine sparse vectors
177     K = (K+K')/2;        % build stiffness matrix
178     U(free,:) = K(free,free)\Ftot(free,:); % displacement solving
179     c(iter) = 0;        % set compliance to zero
180     Sens = 0;          % set sensitivity to zero
181 %% Calculate compliance and sensitivity
182     for i = 1:size(F,2) % for number of load cases
183         Ui = U(:,i);    % displacement per load case
184         c0 = reshape(sum((Ui(dofmat)*Ke).*Ui(dofmat),2),ny,nx); % initial
            compliance
185         c(iter) = c(iter) + sum(sum((Emin+xF.^p*(E-Emin)).*c0)); %
            calculate compliance
186         Sens = Sens + reshape(2*Ui(dofmat)*repmat([0;-9.81*rho/4],4,1),ny
            ,nx) -p*(E-Emin)*xF.^(p-1).*c0; % sensitivity
187     end
188     Senc = ones(ny,nx); % set constraint sensitivity
189     if fil == 0         % optimality criterion with sensitivity filter
190         Sens(:) = H*(x(:).*Sens(:))./Hs./max(1e-3,x(:)); % update
            filtered sensitivity

```

```

191     elseif fil == 1      % optimality criterion with density filter
192         Sens(:) = H*(Sens(:)./Hs); % update filtered sensitivity
193         Senc(:) = H*(Senc(:)./Hs); % update filtered sensitivity of
            constraint
194     elseif fil == 2      % optimality criterion with heaviside filter
195         dx = beta*exp(-beta*xTilde)+exp(-beta); % update hs parameter
196         Sens(:) = H*(Sens(:).*dx(:)./Hs); % update filtered sensitivity
197         Senc(:) = H*(Senc(:).*dx(:)./Hs); % update filtered sensitivity
            of constraint
198     end
199     %% Update design variables Optimality Criterion
200     if sol == 0          % use optimality criterion method
201         l1 = 0;          % initial lower bound for lagrangian mulitplier
202         l2 = 1e9;        % initial upper bound for lagrangian multiplier
203         while (l2-l1)/(l1+l2) > 1e-3; % start loop
204             lag = 0.5*(l1+l2); % average of lagrangian interval
205             xnew = max(0,max(x-move,min(1,min(x+move,x.*sqrt(-Sens./Senc/
                lag))))); % update element densities
206             if fil == 0 % sensitivity filter
207                 xF = xnew; % updated result
208             elseif fil == 1 % density filter
209                 xF(:) = (H*xnew(:))./Hs; % updated filtered density
                    result
210             elseif fil == 2 % heaviside filter
211                 xTilde(:) = (H*xnew(:))./Hs; % set filtered density
212                 xF(:) = 1-exp(-beta*xTilde)+xTilde*exp(-beta); % updated
                    result
213             end
214             if shap == 1 % restriction is on
215                 xF(rest==1) = area; % set restricted area
216             end
217             if sum(xF(:)) > vol*nx*ny; % check for optimum
218                 l1 = lag; % update lower bound to average
219             else
220                 l2 = lag; % update upper bound to average
221             end
222         end
223         %% Method of moving asymptotes
224     elseif sol == 1      % use mma solver
225         xval = xFree(:); % store current design variable for mma
226         if iter == 1     % for the first iteration...
227             cscale = 1/c(iter); % ...set scaling factor for mma solver
228         end
229         f0 = c(iter)*cscale; % objective at current design variable for
            mma
230         df0dx = Sens(efree)*cscale; % store sensitivity for mma
231         f = (sum(xF(:))/(vol*nx*ny)-1); % normalized constraint function
232         dfdx = Senc(efree)'/(vol*ny*nx); % derivative of the constraint
            function
233         [xmma,~,~,~,~,~,~,~,~,low,upp] = ...
234             mmasub(m,n,iter,xval,xmin,xmax,xold1,xold2, ...
235                 f0,df0dx,df0dx2,f,dfdx,dfdx2,low,upp,a0,a,cmma,d,subs); % mma
                solver

```

```

236     xold2 = xold1; % used by mma to monitor convergence
237     xold1 = xFree(:); % previous x, to monitor convergence
238     xnew = xF; % update result
239     xnew(efree) = xmma; % include restricted elements
240     xnew = reshape(xnew,ny,nx); % reshape xmma vector to original
        size
241     if fil == 0 % sensitivity filter
242         xF = xnew; % update design variables
243     elseif fil == 1 % density filter
244         xF(:) = (H*xnew(:))./Hs; % update filtered densities result
245     elseif fil == 2 % heaviside filter
246         xTilde(:) = (H*xnew(:))./Hs; % filtered result
247         xF(:) = 1 - exp(-beta*xTilde) + xTilde*exp(-beta); % update design
        variable
248     end
249     if shap == 1 % if restrictions enableed
250         xF(rest==1) = area; % set restricted area
251     end
252
253 end
254 xFree = xnew(efree); % set non-restricted area
255 diff = max(abs(xnew(:)-x(:))); % difference of maximum element change
256 x = xnew; % update design variable
257 if fil == 2 && beta < 512 && pen == p(end) && (loopbeta >= 50 || diff
    <= tol) % hs filter
258     beta = 2*beta; % increase beta-factor
259     fprintf('beta now is %3.0f\n',beta) % display increase of b-
        factor
260     loopbeta = 0; % set hs filter loop to zero
261     diff = 1; % set convergence to initial value
262 end
263 %% Store results into database X
264 X(:, :, iter) = xF; % each element value x is stored for each
    iteration
265 C(iter) = c(iter); % each compliance is stored for each iteration
266 assignin('base', 'X', X); % each iteration (3rd dimension)
267 assignin('base', 'C', C); % each iteration (3rd dimension)
268 %% Results
269 if dis == 1 % display iterations
270     disp([' Iter:' sprintf('%4i',iter) ' Obj:' sprintf('%10.4f',c(
        iter)) ...
271         ' Vol:' sprintf('%6.3f',mean(xF(:))) ' Diff:' sprintf('%6.3f',
        diff)]);
272 end
273 if draw == 1 % plot iterations
274     figure(1)
275     subplot(2,1,1)
276     colormap(gray); imagesc(1-xF);
277     set(gca, 'XTick', [], 'YTick', [], 'XTicklabel', [], ...
278         'YTicklabel', [], 'xcolor', 'w', 'ycolor', 'w')
279     xlabel(sprintf('c = %.2f',c(iter)), 'Color', 'k')
280     drawnow;
281     hold on

```

```

282     if iter == 1
283         axis equal; axis tight;
284         % Plot coloured dots for constraints
285         for i = 1:length(fix)
286             npx(i) = ceil(fix(i)/(2*(ny+1)))-0.5;
287             nplot = ceil(fix(i)/2);
288             while nplot > (ny+1)
289                 nplot = nplot-(ny+1);
290             end
291             npy(i) = nplot-0.5;
292         end
293         plot(npx,npy,'r.','MarkerSize',20)
294         % Plot coloured dots for force application
295         for i = 1:length(Fe)
296             npfx(i) = ceil(Fe(i)/(2*(ny+1)))-0.5;
297             nplot = ceil(Fe(i)/2);
298             while nplot > (ny+1)
299                 nplot = nplot-(ny+1);
300             end
301             npfy(i) = nplot-0.5;
302         end
303         plot(npfx,npfy,'g.','MarkerSize',20)
304     end
305     % Plot compliance plot
306     figure(1)
307     subplot(2,1,2)
308     plot(c(1:iter))
309     xaxmax = c(iter);
310     yaxmax = max(c);
311     yaxmin = min(c(1:iter));
312     if pcon == 0
313         yaxmax = mean([yaxmin yaxmax]);
314     end
315     ylim([0.95*yaxmin yaxmax])
316     xlim([0 iter+10])
317 end
318 end
319 %% ONLY DISPLAY FINAL RESULT
320 if dis == 0 % display final result
321     disp([' Iter:' sprintf('%4i',iter) ' Obj:' sprintf('%10.4f',c) ...
322         ' Con:' sprintf('%6.3f',diff)]);
323 end
324 if draw == 0 % plot final result
325     figure(1)
326     subplot(2,1,1)
327     colormap(gray); imagesc(1-xF);
328     axis equal; axis tight;
329     set(gca,'XTick',[],'YTick',[],'XTicklabel',[],...
330         'YTicklabel',[],'xcolor','w','ycolor','w')
331     xlabel(sprintf('c = %.2f',c(iter)),'Color','k')
332     drawnow;
333     hold on
334     %% Plot coloured dots for constraints

```

```
335     for i = 1:length(fix)
336         npx(i) = ceil(fix(i)/(2*(ny+1))) - 0.5;
337         nplot = ceil(fix(i)/2);
338         while nplot > (ny+1)
339             nplot = nplot - (ny+1);
340         end
341         npy(i) = nplot - 0.5;
342     end
343     plot(npx, npy, 'r.', 'MarkerSize', 20)
344     %% Plot coloured dots for force application
345     for i = 1:length(Fe)
346         npfx(i) = ceil(Fe(i)/(2*(ny+1))) - 0.5;
347         nplot = ceil(Fe(i)/2);
348         while nplot > (ny+1)
349             nplot = nplot - (ny+1);
350         end
351         npfy(i) = nplot - 0.5;
352     end
353     plot(npfx, npfy, 'g.', 'MarkerSize', 20)
354     %% Plot compliance plot
355     if adv == 0
356         figure(1)
357         subplot(2,1,2)
358         plot(c(1:iter))
359         xaxmax = c(iter);
360         yaxmax = max(c);
361         yaxmin = min(c(1:iter));
362         if pcon == 0
363             yaxmax = mean([yaxmin yaxmax]);
364         end
365         ylim([0.95*yaxmin yaxmax])
366         xlim([0 iter+10])
367     end
368 end
369 toc           % stop timer
```

B.4 BASIC 3D.m

In the previous section, an add-in is given to produce a simple 3D optimization. However, for certain cases, it could be helpful to have the same different options as described in the add-in sections. The following code includes the same functionality as the BASIC code (B.3), but now for three dimensions. This code is tested and working. A small reminder should be made regarding the restrictions option. In the 2D optimization code a simple circle could be made, to describe circular restricted area. In this 3D code however, this circular restricted area is replaced by two options. Shape option 1 describes a cylindrical restrictive regions, shape option 2 describes a spherical restrictive region.

```

1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %
3  % Topology Optimization Using Matlab
4  % BASIC3D.m
5  %
6  % Delft University of Technology, Department PME
7  % Master of Science Thesis Project
8  %
9  % Stefan Broxterman
10 %
11 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
12 %
13 tic % start timer
14 %% DEFINE PARAMETERS
15 adv = 1; % use advanced function [0 = off, 1 = on]
16 if adv == 0 % define parameters at behalf of the advanced
17     function
18     nx = 30; % number of elements horizontal
19     ny = 10; % number of elements vertical
20     nz = 4; % number of elements lateral
21     vol = 0.5; % volume fraction [0-1]
22     pen = 3; % penalty
23     rmin = 1.5; % filter size
24     fil = 0; % filter method [0 = sensitivity filtering, 1 =
25         density filtering, 2 = heaviside filtering]
26     clear X ;
27     clc; clf; close all;
28 end
29 %% DEFINE SOLUTION METHOD
30 sol = 1; % solution method [0 = oc(sens), 1 = mma]
31 pcon = 1; % use continuation method [0 = off, 1 = on]
32 %% DEFINE CALCULATION
33 tol = 0.01; % tolerance for convergence criterion [0.01]
34 move = 0.2; % move limit for lagrange [0.2]
35 pcinc = 1.03; % penalty continuation increasing factor [1.03]
36 piter = 20; % number of iteration for starting penalty [20]
37 miter = 1000; % maximum number of iterations [1000]
38 graysc = 1; % use gray-scale filter [0 = off, 1 = on]
39 q = 1; % gray-scale parameter
40 qmax = 2; % maximum gray-scale parameter

```

```

39 plotiter = 5;           % gap of iterations used to plot or draw
    iterations
40 %% DEFINE OUTPUT
41 draw = 1;               % plot iterations [0 = off, 1 = on, 2 = partial]
42 dis = 1;               % display iterations [0 = off, 1 = on, 2 =
    partial]
43 %% DEFINE MATERIAL
44 E = 1;                 % young's modulus of solid [1]
45 Emin = 1e-9;          % young's modulus of void [1e-9]
46 nu = 0.3;             % poisson ratio [0.3]
47 rho = 0e-3;           % density [0e-3]
48 g = 9.81;            % gravitational acceleration [9.81]
49 %% DEFINE FORCE
50 Fe = (3*(nx+1)*(ny+1)-1)+(3*(nx+1)*(ny+1))*(0:nz)'; % element of force
    application
51 Fn = 1;               % number of applied force locations [1]
52 Fv = -1;             % value of applied force [-1]
53 %% DEFINE SUPPORTS
54 fix = repmat((1:3*(ny+1))',1,nz+1)+repmat((0:nz)*3*(nx+1)*(ny+1),length
    ((1:3*(ny+1))),1); % fixed elements
55 fix = fix(:);
56 %% DEFINE ELEMENT RESTRICTIONS
57 shap = 2;            % [0 = no restrictions, 1 = cylinder, 2 = sphere]
58 area = 0;           % [0 = no material (passive), 1 = material (
    active)]
59 nodr = (round(ny/2)+(0:ny:(nx-1)*ny)); % custom restricted nodes
60 %% PREPARE FINITE ELEMENT
61 N = 3*(nx+1)*(ny+1)*(nz+1); % total elements nodes
62 all = 1:3*(nx+1)*(ny+1)*(nz+1); % all degrees of freedom
63 free = setdiff(all,fix); % free degrees of freedom
64 A = [32 6 -8 6 -6 4 3 -6 -10 3 -3 -3 -4 -8;
    -48 0 0 -24 24 0 0 0 12 -12 0 12 12 12]; % fem
65 k = 1/72*A'*[1; nu]; % simple stiffness matrix
66 %% GENERATE SIX SUB-MATRICES AND THEN GET KE MATRIX
67 K1 = [k(1) k(2) k(2) k(3) k(5) k(5);
    k(2) k(1) k(2) k(4) k(6) k(7);
    k(2) k(2) k(1) k(4) k(7) k(6);
    k(3) k(4) k(4) k(1) k(8) k(8);
    k(5) k(6) k(7) k(8) k(1) k(2);
    k(5) k(7) k(6) k(8) k(2) k(1)]; % stiffness matrix
68 K2 = [k(9) k(8) k(12) k(6) k(4) k(7);
    k(8) k(9) k(12) k(5) k(3) k(5);
    k(10) k(10) k(13) k(7) k(4) k(6);
    k(6) k(5) k(11) k(9) k(2) k(10);
    k(4) k(3) k(5) k(2) k(9) k(12)
    k(11) k(4) k(6) k(12) k(10) k(13)]; % stiffness matrix
69 K3 = [k(6) k(7) k(4) k(9) k(12) k(8);
    k(7) k(6) k(4) k(10) k(13) k(10);
    k(5) k(5) k(3) k(8) k(12) k(9);
    k(9) k(10) k(2) k(6) k(11) k(5);
    k(12) k(13) k(10) k(11) k(6) k(4);
    k(2) k(12) k(9) k(4) k(5) k(3)]; % stiffness matrix
70 K4 = [k(14) k(11) k(11) k(13) k(10) k(10);

```

```

87     k(11) k(14) k(11) k(12) k(9)  k(8);
88     k(11) k(11) k(14) k(12) k(8)  k(9);
89     k(13) k(12) k(12) k(14) k(7)  k(7);
90     k(10) k(9)  k(8)  k(7)  k(14) k(11);
91     k(10) k(8)  k(9)  k(7)  k(11) k(14)]; % stiffness matrix
92 K5 = [k(1) k(2)  k(8)  k(3) k(5)  k(4);
93       k(2) k(1)  k(8)  k(4) k(6)  k(11);
94       k(8) k(8)  k(1)  k(5) k(11) k(6);
95       k(3) k(4)  k(5)  k(1) k(8)  k(2);
96       k(5) k(6)  k(11) k(8) k(1)  k(8);
97       k(4) k(11) k(6)  k(2) k(8)  k(1)]; % stiffness matrix
98 K6 = [k(14) k(11) k(7)  k(13) k(10) k(12);
99       k(11) k(14) k(7)  k(12) k(9)  k(2);
100      k(7)  k(7)  k(14) k(10) k(2)  k(9);
101      k(13) k(12) k(10) k(14) k(7)  k(11);
102      k(10) k(9)  k(2)  k(7)  k(14) k(7);
103      k(12) k(2)  k(9)  k(11) k(7)  k(14)]; % stiffness matrix
104 Ke = 1/((nu+1)*(1-2*nu))*...
105     [ K1  K2  K3  K4;
106       K2' K5  K6  K3';
107       K3' K6  K5' K2';
108       K4  K3  K2  K1']; % element stiffness matrix
109 nodes = reshape(1:(nx+1)*(ny+1),1+ny,1+nx); % create node number matrix
110 nodes2 = reshape(nodes(1:end-1,1:end-1),ny*nx,1); % create node number
111         matrix
112 nodes3 = 0:(ny+1)*(nx+1):(nz-1)*(ny+1)*(nx+1); % create node number
113         matrix
114 nodes4 = repmat(nodes2, size(nodes3))+repmat(nodes3, size(nodes2)); %
115         create node number matrix
116 dofvec = 3*nodes4(:)+1; % create dof vector
117 dofmat = repmat(dofvec,1,24)+repmat([0 1 2 3*ny+[3 4 5 0 1 2] -3 -2 -1
118         3*(ny+1)*(nx+1) + [0 1 2 3*ny + [3 4 5 0 1 2] -3 -2 -1]],nx*ny*nz,1);
119         % create dof matrix
120 iK = kron(dofmat,ones(24,1))'; % build sparse i
121 jK = kron(dofmat,ones(1,24))'; % build sparse j
122 %% PREPARE FILTER
123 iH = ones(nx*ny*nz*(2*(ceil(rmin)-1)+1)^2,1); % build sparse i
124 jH = ones(size(iH)); % create sparse vector of ones
125 kH = zeros(size(iH)); % create sparse vector of zeros
126 m = 0; % index for filtering
127 for h = 1:nz % for each element calculate...
128     for i = 1:nx % distance between elements'...
129         for j = 1:ny % centre for filtering
130             r1 = (h-1)*nx*ny + (i-1)*ny+j; % sparse value 1
131             for k2 = max(h-(ceil(rmin)-1),1):min(h+(ceil(rmin)-1),nz) %
132                 centre of element
133                 for k = max(i-(ceil(rmin)-1),1):min(i+(ceil(rmin)-1),nx)
134                     % centre of element
135                     for l = max(j-(ceil(rmin)-1),1):min(j+(ceil(rmin)-1),
136                         ny) % centre of element
137                         r2 = (k2-1)*nx*ny + (k-1)*ny+l; % sparse value 2
138                         m = m+1; % update index for filtering
139                         iH(m) = r1; % sparse vector for filtering

```

```

132         jH(m) = r2; % sparse vector for filtering
133         kH(m) = max(0,rmin-sqrt((i-k)^2+(j-1)^2)+(h-k2)
                    ^2); % weight factor
134     end
135     end
136     end
137     end
138     end
139 end
140 H = sparse(iH,jH,kH); % build filter
141 Hs = sum(H,2); % summation of filter
142 %% DEFINE STRUCTURAL
143 x = repmat(vol,ny,nx,nz); % initial material distribution
144 if shap == 0 % no restrictions
145     efree = (1:nx*ny*nz)'; % all elements are free
146     eres= []; % no restricted elements
147 elseif shap == 1 || shap == 2 % restrictions
148     rest = zeros(ny,nx,nz); % pre-allocate space
149     for i = 1:nx % start loop
150         for j = 1:ny % for each element
151             for k = 1:nz % for lateral element
152                 if sqrt((j-ny/2)^2+(i-nx/3)^2) < ny/2.5 % circular
                    restriction
153                     if shap == 1 % cylindrical restriction
154                         rest(j,i,k) = 1; % write restriction
155                         if rest(j,i,k) == area % check for restriction
156                             x(j,i,k) = area; % store restrictions in
                                material distribution
157                     end
158                 elseif shap == 2 % spherical restriction
159                     if sqrt((j-ny/2)^2+(k-nz/2)^2) < nz/2.5 %
                        spherical restriction
160                         if sqrt((k-nz/2)^2+(i-nx/3)^2) < nz/2.5 %
                            spherical restriction
161                             rest(j,i,k) = 1; % write restriction
162                             if rest(j,i,k) == area % check for
                                restriction
163                                 x(j,i,k) = area; % store restrictions
                                    in material distribution
164                             end
165                         end
166                     end
167                 end
168             end
169         end
170     end
171 end
172 efree = find(rest ~= 1); % set free elements
173 eres = find(rest == 1); % set restricted elements
174 end
175 if fil == 0 || fil == 1 % sensitivity, density filter
176     xF = x; % set filtered design variables
177 elseif fil == 2 % heaviside filter

```

```

178     beta = 1;           % hs filter
179     xTilde = x;        % hs filter
180     xF = 1-exp(-beta*xTilde)+xTilde*exp(-beta); % set filtered design
        space
181 end
182 xFree = xF(efree);    % define free design matrix
183 %% DEFINE STRUCTURAL
184 Fsiz = size(Fe,2);    % size of load vector
185 F = sparse(Fe,Fn,Fv,N,Fsiz); % define load vector
186 %% DEFINE MMA PARAMETERS
187 m = 1;                % number of constraint functions
188 n = size(xFree(:,1)); % number of variables
189 xmin = zeros(n,1);    % minimum values of x
190 xmax = ones(n,1);    % maximum values of x
191 xold1 = zeros(n,1);  % previous x, to monitor convergence
192 xold2 = xold1;       % used by mma to monitor convergence
193 df0dx2 = zeros(n,1); % second derivative of the objective function
194 dfdx2 = zeros(1,n);  % second derivative of the constraint function
195 low = xmin;          % lower asymptotes from the previous iteration
196 upp = xmax;         % upper asymptotes from the previous iteration
197 a0 = 1;              % constant a_0 in mma formulation
198 a = zeros(m,1);      % constant a_i in mma formulation
199 cmma = 1e3*ones(m,1); % constant c_i in mma formulation
200 d = zeros(m,1);      % constant d_i in mma formulation
201 subs = 200;          % maximum number of subsolv iterations
202 %% PRE-ALLOCATE SPACE
203 npz = zeros(length(fix),1)'; % pre-allocate constraint dots
204 npy = zeros(length(fix),1)'; % pre-allocate constraint dots
205 npz = zeros ( length ( fix ) ,1)'; % pre-allocate constraint dots
206 npfx = zeros(length(Fe),1)'; % pre-allocate force dots
207 npfy = zeros(length(Fe),1)'; % pre-allocate force dots
208 npfz = zeros(length(Fe),1)'; % pre-allocate force dots
209 U = zeros(size(F));    % pre-allocate space displacement
210 c = zeros(miter,1);   % pre-allocate objective vector
211 %% INITIALIZE LOOP
212 iter = 0;              % initialize loop
213 diff = 1;             % initialize convergence criterion
214 loopbeta = 1;         % initialize beta-loop
215 %% START LOOP
216 while ((diff > tol) || (iter < piter+1)) && iter < miter % convergence
        criterion not met
217     loopbeta = loopbeta +1; % iteration loop for hs filter
218     iter = iter+1;         % define iteration
219     if pcon == 1          % use continuation method
220         if iter <= piter % first number of iterations...
221             p = 1;        %... set penalty 1
222         elseif iter > piter % after a number of iterations...
223             p = min(pen,pcinc*p); % ... set continuation penalty
224         end
225     elseif pcon == 0      % not using continuation method
226         p = pen;         % set penalty
227     end
228     if grayscale == 1    % if grayscale is enabled

```

```

229         if iter <= 15    % within 15 iterations
230             q = 1;        % don't use grayscale
231         else            % after 15 iterations
232             q = min(qmax,1.01*q); % use continuation method to pick a
                gray-scale factor
233         end
234     end
235 %% Selfweight
236     if rho ~= 0        % gravity is involved
237         xP=zeros(ny,nx,nz); % pre-allocate space
238         xP(xF>0.25) = xF(xF>0.25).^p; % normal penalization
239         xP(xF<=0.25) = xF(xF<=0.25).*(0.25^(p-1)); % below pseudo-density
240         Fsw = zeros(N,1); % pre-allocate self-weight
241         for i=1:nx*ny*nz % for each element, set gravitational...
242             Fsw(dofmat(i,2:3:end))=Fsw(dofmat(i,2:3:end))-xF(i)*rho
                *9.81/4;
243         end            % force to the attached nodes
244         Fsw= repmat(Fsw,1,size(F,2)); % set self-weight for load cases
245     elseif rho == 0    % no gravity
246         xP = xF.^p;    % penalized design variable
247         Fsw = 0;      % no selfweight
248     end
249     Ftot = F + Fsw;    % total force
250 %% Finite element analysis
251     kK = Ke(:)*(Emin+xP(:)')*(E-Emin)); % create sparse vector k
252     K = sparse(iK,jK,kK); % combine sparse vectors
253     K = (K+K')/2;      % build stiffness matrix
254     U(free,:) = K(free,free)\Ftot(free,:); % displacement solving
255     c(iter) = 0;      % set compliance to zero
256     Sens = 0;        % set sensitivity to zero
257 %% Calculate compliance and sensitivity
258     for i = 1:size(F,2) % for number of load cases
259         Ui = U(:,i); % displacement per load case
260         c0 = reshape(sum((Ui(dofmat)*Ke).*Ui(dofmat)),2),ny,nx,nz); %
                initial compliance
261         c(iter) = c(iter) + sum(sum(sum((Emin+xF.^p*(E-Emin)).*c0))); %
                calculate compliance
262         Sens = Sens + reshape(2*Ui(dofmat)*repmat([0;-9.81*rho/4; 0],8,1)
                ,ny,nx,nz) -p*(E-Emin)*xF.^(p-1).*c0; % sensitivity
263     end
264     Senc = ones(ny,nx,nz); % set constraint sensitivity
265     if fil == 0        % optimality criterion with sensitivity filter
266         Sens(:) = H*(x(:).*Sens(:))./Hs./max(1e-3,x(:)); % update
                filtered sensitivity
267     elseif fil == 1    % optimality criterion with density filter
268         Sens(:) = H*(Sens(:))./Hs; % update filtered sensitivity
269         Senc(:) = H*(Senc(:))./Hs; % update filtered sensitivity of
                constraint
270     elseif fil == 2    % optimality criterion with heaviside filter
271         dx = beta*exp(-beta*xTilde)+exp(-beta); % update hs parameter
272         Sens(:) = H*(Sens(:).*dx(:))./Hs; % update filtered sensitivity
273         Senc(:) = H*(Senc(:).*dx(:))./Hs; % update filtered sensitivity
                of constraint

```

```

274     end
275     %% Update design variables Optimality Criterion
276     if sol == 0           % use optimality criterion method
277         l1 = 0;           % initial lower bound for lagranian mulitplier
278         l2 = 1e9;        % initial upper bound for lagranian multiplier
279         while (l2-l1)/(l1+l2) > 1e-3; % start loop
280             lag = 0.5*(l1+l2); % average of lagranian interval
281             if grayscale == 0 % don't use grayscale
282                 xnew = max(0,max(x-move,min(1,min(x+move,x.*sqrt(-Sens./
                Senc/lag))))); % update element densities
283             elseif grayscale == 1 % use grayscale
284                 xnew = max(0,max(x-move,min(1,min(x+move,x.*sqrt(-Sens./
                Senc/lag)).^q))); % update element densities
285             end
286             if fil == 0 % sensitivity filter
287                 xF = xnew; % updated result
288             elseif fil == 1 % density filter
289                 xF(:) = (H*xnew(:))./Hs; % updated filtered density
                result
290             elseif fil == 2 % heaviside filter
291                 xTilde(:) = (H*xnew(:))./Hs; % set filtered density
292                 xF(:) = 1-exp(-beta*xTilde)+xTilde*exp(-beta); % updated
                result
293             end
294             if shap == 1 || shap == 2 % restriction is on
295                 xF(rest==1) = area; % set restricted area
296             end
297             if sum(xF(:)) > vol*nx*ny*nz; % check for optimum
298                 l1 = lag; % update lower bound to average
299             else
300                 l2 = lag; % update upper bound to average
301             end
302         end
303         %% Method of moving asymptotes
304     elseif sol == 1 % use mma solver
305         xval = xFree(:); % store current design variable for mma
306         if iter == 1 % for the first iteration...
307             cscale = 1/c(iter); % ...set scaling factor for mma solver
308         end
309         f0 = c(iter)*cscale; % objective at current design variable for
            mma
310         df0dx = Sens(efree)*cscale; % store sensitivity for mma
311         f = (sum(xF(:))/(vol*nx*ny*nz)-1); % normalized constraint
            function
312         dfdx = Senc(efree)'/ (vol*ny*nx*nz); % derivative of the
            constraint function
313         [xmma,~,~,~,~,~,~,~,~,low,upp] = ...
314             mmasub(m,n,iter,xval,xmin,xmax,xold1,xold2, ...
315                 f0,df0dx,df0dx2,f,dfdx,dfdx2,low,upp,a0,a,cmma,d,subs); % mma
            solver
316         xold2 = xold1; % used by mma to monitor convergence
317         xold1 = xFree(:); % previous x, to monitor convergence
318         xnew = xF; % update result

```

```

319     xnew(efree) = xmma; % include restricted elements
320     xnew = reshape(xnew,ny,nx,nz); % reshape xmma vector to original
        size
321     if fil == 0      % sensitivity filter
322         xF = xnew; % update design variables
323     elseif fil == 1 % density filter
324         xF(:) = (H*xnew(:))./Hs; % update filtered densities result
325     elseif fil == 2 % heaviside filter
326         xTilde(:) = (H*xnew(:))./Hs; % filtered result
327         xF(:) = 1 - exp(-beta*xTilde) + xTilde * exp(-beta); % update design
        variable
328     end
329     if shap == 1 || shap == 2 % if restrictions enableed
330         xF(rest==1) = area; % set restricted area
331     end
332
333     end
334     xFree = xnew(efree); % set non-restricted area
335     diff = max(abs(xnew(:)-x(:))); % difference of maximum element change
336     x = xnew; % update design variable
337     if fil == 2 && beta < 512 && pen == p(end) && (loopbeta >= 50 || diff
        <= tol) % hs filter
338         beta = 2*beta; % increase beta-factor
339         fprintf('beta now is %3.0f\n',beta) % display increase of b-
        factor
340         loopbeta = 0; % set hs filter loop to zero
341         diff = 1; % set convergence to initial value
342     end
343     %% Store results into database X
344     X(:, :, :, iter) = xF; % each element value x is stored for each
        iteration
345     C(iter) = c(iter); % each compliance is stored for each iteration
346     assignin('base', 'X', X); % each iteration (3rd dimension)
347     assignin('base', 'C', C); % each iteration (3rd dimension)
348     %% Results
349     if dis == 1 % display iterations
350         disp([' Iter:' sprintf('%4i',iter) ' Obj:' sprintf('%10.4f',c(
        iter)) ...
351             ' Vol:' sprintf('%6.3f',mean(xF(:))) ' Diff:' sprintf('%6.3f',
        diff)]);
352     elseif dis == 2 % display parts of iterations
353         if iter == 1 || iter == disiter
354             if iter == 1
355                 disiter = plotiter;
356             elseif iter == disiter
357                 disiter = disiter + plotiter;
358             end
359             disp([' Iter:' sprintf('%4i',iter) ' Obj:' sprintf('%10.4f',c
        (iter)) ...
360                 ' Vol:' sprintf('%6.3f',mean(xF(:))) ' Diff:' sprintf('%6.3f',
        diff)]);
361         end
362     end

```

```

363     end
364     if draw == 1           % plot iterations
365         figure(1)
366         subplot(2,1,1)
367         [nely,nelx,nelz] = size(xF);
368         hx = 1; hy = 1; hz = 1;           % User-defined unit element
369         size
370         face = [1 2 3 4; 2 6 7 3; 4 3 7 8; 1 5 8 4; 1 2 6 5; 5 6 7 8];
371         for k = 1:nelz
372             z = (k-1)*hz;
373             for i = 1:nelx
374                 xplot = (i-1)*hx;
375                 for j = 1:nely
376                     y = nely*hy - (j-1)*hy;
377                     if (xF(j,i,k) > 0.5) % User-defined display density
378                         threshold
379                         vert = [xplot y z; xplot y-hx z; xplot+hx y-hx z;
380                                 xplot+hx y z; xplot y z+hx;xplot y-hx z+hx;
381                                 xplot+hx y-hx z+hx;xplot+hx y z+hx];
382                         vert(:,[2 3]) = vert(:,[3 2]); vert(:,2,:) = -
383                             vert(:,2,:);
384                         patch('Faces',face,'Vertices',vert,'FaceColor'
385                               ,[0.2+0.8*(1-xF(j,i,k)),0.2+0.8*(1-xF(j,i,k))
386                               ,0.2+0.8*(1-xF(j,i,k))]);
387                         hold on;
388                     end
389                 end
390             end
391         end
392     end
393     axis equal; axis tight;
394     set(gca,'XTick',[],'YTick',[],'ZTick',[],'XTicklabel',[],...
395           'YTicklabel',[],'ZTicklabel',[],'xcolor','w','ycolor','w','
396           zcolor','w')
397     view([30,30]);
398     xlabel(sprintf('c = %.2f',c(iter)),'Color','k')
399     drawnow;
400     hold on
401     if iter == 1
402         % Plot coloured dots for constraints
403         for i = 1:length(fix)
404             nplotx = ceil(fix(i)/(3*(ny+1)));
405             while nplotx > (nx+1)
406                 nplotx = nplotx -(nx+1);
407             end
408             npx(i) = nplotx-1;
409             nplot = ceil(fix(i)/3);
410             while nplot > (ny+1)
411                 nplot = nplot-(ny+1);
412             end
413             npy(i) = nplot-1;
414             npz(i) = 1-ceil(fix(i)/(3*(nx+1)*(ny+1)));
415         end
416         plot3(npx,npz,npy,'r.','MarkerSize',20)

```

```

408         % Plot coloured dots for force application
409         for i = 1:length(Fe)
410             nplotx = ceil(Fe(i)/(3*(ny+1)));
411             while nplotx > (nx+1)
412                 nplotx = nplotx -(nx+1);
413             end
414             npfx(i) = nplotx-1;
415             nplot = ceil(Fe(i)/3);
416             while nplot > (ny)
417                 nplot = nplot-(ny+1);
418             end
419             npfy(i) = nplot;
420             npfz(i) = 1-ceil(Fe(i)/(3*(nx+1)*(ny+1)));
421         end
422         plot3(npfx,npfz,npfy,'g.','MarkerSize',20)
423         drawnow;
424     end
425     % Plot compliance plot
426     figure(1)
427     subplot(2,1,2)
428     plot(c(1:iter))
429     xaxmax = c(iter);
430     yaxmax = max(c);
431     yaxmin = min(c(1:iter));
432     ylim([0.95*yaxmin yaxmax])
433     xlim([0 iter+10])
434     elseif draw == 2         % plot parts of iterations
435         if iter == 1 || iter == drawiter
436             if iter == 1
437                 drawiter = plotiter;
438             elseif iter == drawiter
439                 drawiter = drawiter + plotiter;
440             end
441             figure(1)
442             subplot(2,1,1)
443             [nely,nelx,nelz] = size(xF);
444             hx = 1; hy = 1; hz = 1;           % User-defined unit element
445             size
446             face = [1 2 3 4; 2 6 7 3; 4 3 7 8; 1 5 8 4; 1 2 6 5; 5 6 7 8];
447             for k = 1:nelz
448                 z = (k-1)*hz;
449                 for i = 1:nelx
450                     xplot = (i-1)*hx;
451                     for j = 1:nely
452                         y = nely*hy - (j-1)*hy;
453                         if (xF(j,i,k) > 0.5) % User-defined display density
454                             threshold
455                             vert = [xplot y z; xplot y-hx z; xplot+hx y-hx z;
456                                     xplot+hx y z; xplot y z+hx;xplot y-hx z+hx;
457                                     xplot+hx y-hx z+hx;xplot+hx y z+hx];
458                             vert(:,[2 3]) = vert(:,[3 2]); vert(:,2,:) = -
459                                 vert(:,2,:);

```

```

455         patch('Faces',face,'Vertices',vert,'FaceColor'
                ,[0.2+0.8*(1-xF(j,i,k)),0.2+0.8*(1-xF(j,i,k))
                ,0.2+0.8*(1-xF(j,i,k))]);
456         hold on;
457     end
458 end
459 end
460 end
461 axis equal; axis tight;
462 set(gca,'XTick',[],'YTick',[],'ZTick',[],'XTicklabel',[],...
463     'YTicklabel',[],'ZTicklabel',[],'xcolor','w','ycolor','w','
        zcolor','w');
464 view([30,30]);
465 xlabel(sprintf('c = %.2f',c(iter)),'Color','k');
466 drawnow;
467 hold on
468 if iter == 1
469     % Plot coloured dots for constraints
470     for i = 1:length(fix)
471         nplotx = ceil(fix(i)/(3*(ny+1)));
472         while nplotx > (nx+1)
473             nplotx = nplotx -(nx+1);
474         end
475         npx(i) = nplotx-1;
476         nplot = ceil(fix(i)/3);
477         while nplot > (ny+1)
478             nplot = nplot-(ny+1);
479         end
480         npy(i) = nplot-1;
481         npz(i) = 1-ceil(fix(i)/(3*(nx+1)*(ny+1)));
482     end
483     plot3(npx,npz,npy,'r.','MarkerSize',20)
484     % Plot coloured dots for force application
485     for i = 1:length(Fe)
486         nplotx = ceil(Fe(i)/(3*(ny+1)));
487         while nplotx > (nx+1)
488             nplotx = nplotx -(nx+1);
489         end
490         npfx(i) = nplotx-1;
491         nplot = ceil(Fe(i)/3);
492         while nplot > (ny)
493             nplot = nplot-(ny+1);
494         end
495         npfy(i) = nplot;
496         npfz(i) = 1-ceil(Fe(i)/(3*(nx+1)*(ny+1)));
497     end
498     plot3(npfx,npfz,npfy,'g.','MarkerSize',20)
499     drawnow;
500 end
501 % Plot compliance plot
502 figure(1)
503 subplot(2,1,2)
504 plot(c(1:iter))

```

```

505     xaxmax = c(iter);
506     yaxmax = max(c);
507     yaxmin = min(c(1:iter));
508     ylim([0.95*yaxmin yaxmax])
509     xlim([0 iter+10])
510     end
511 end
512 end
513 %% ONLY DISPLAY FINAL RESULT
514 if dis == 0 || dis == 2 % display final result
515     disp([' Iter:' sprintf('%4i',iter) ' Obj:' sprintf('%10.4f',c(iter))
516         ...
517         ' Vol:' sprintf('%6.3f',mean(xF(:))) ' Diff:' sprintf('%6.3f',
518             ,diff)]]);
519 end
520 if draw == 0 || draw == 2 % plot final result
521     figure(1)
522     subplot(2,1,1)
523     [nely,nelx,nelz] = size(xF);
524     hx = 1; hy = 1; hz = 1; % User-defined unit element size
525     face = [1 2 3 4; 2 6 7 3; 4 3 7 8; 1 5 8 4; 1 2 6 5; 5 6 7 8];
526     for k = 1:nelz
527         z = (k-1)*hz;
528         for i = 1:nelx
529             xplot = (i-1)*hx;
530             for j = 1:nely
531                 y = nely*hy - (j-1)*hy;
532                 if (xF(j,i,k) > 0.5) % User-defined display density
533                     threshold
534                     vert = [xplot y z; xplot y-hx z; xplot+hx y-hx z;
535                         xplot+hx y z; xplot y z+hx;xplot y-hx z+hx; xplot+
536                             hx y-hx z+hx;xplot+hx y z+hx];
537                     vert(:,[2 3]) = vert(:,[3 2]); vert(:,2,:) = -vert
538                         (:,2,:);
539                     patch('Faces',face,'Vertices',vert,'FaceColor',
540                         ,[0.2+0.8*(1-xF(j,i,k)),0.2+0.8*(1-xF(j,i,k))
541                             ,0.2+0.8*(1-xF(j,i,k))]);
542                     hold on;
543                 end
544             end
545         end
546     end
547 end
548 axis equal; axis tight;
549 set(gca,'XTick',[],'YTick',[],'ZTick',[],'XTicklabel',[],...
550     'YTicklabel',[],'ZTicklabel',[],'xcolor','w','ycolor','w','zcolor',
551     ',','w')
552 view([30,30]);
553 xlabel(sprintf('c = %.2f',c(iter)),'Color','k')
554 drawnow;
555 hold on
556 % Plot coloured dots for constraints
557 for i = 1:length(fix)
558     nplotx = ceil(fix(i)/(3*(ny+1)));

```

```

549     while nplotx > (nx+1)
550         nplotx = nplotx -(nx+1);
551     end
552     npx(i) = nplotx-1;
553     nplot = ceil(fix(i)/3);
554     while nplot > (ny+1)
555         nplot = nplot-(ny+1);
556     end
557     npy(i) = nplot-1;
558     npz(i) = 1-ceil(fix(i)/(3*(nx+1)*(ny+1)));
559 end
560 plot3(npx,npz,npy,'r.','MarkerSize',20)
561 % Plot coloured dots for force application
562 for i = 1:length(Fe)
563     nplotx = ceil(Fe(i)/(3*(ny+1)));
564     while nplotx > (nx+1)
565         nplotx = nplotx -(nx+1);
566     end
567     npfx(i) = nplotx-1;
568     nplot = ceil(Fe(i)/3);
569     while nplot > (ny)
570         nplot = nplot-(ny+1);
571     end
572     npfy(i) = nplot;
573     npfz(i) = 1-ceil(Fe(i)/(3*(nx+1)*(ny+1)));
574 end
575 plot3(npfx,npfz,npfy,'g.','MarkerSize',20)
576 % Plot compliance plot
577 figure(1)
578 subplot(2,1,2)
579 plot(c(1:iter))
580 xaxmax = c(iter);
581 yaxmax = max(c);
582 yaxmin = min(c(1:iter));
583 ylim([0.95*yaxmin yaxmax])
584 xlim([0 iter+10])
585 end
586 toc % stop timer

```

B.5 ADVANCED 3D.m

By the inspiration of the ADVANCED (B.2) for 2D-problems, an 3D-adapted code is made available. The changes are quite big, so it's recommended to just run this new file, instead of writing an add-in code.

By the introduction of this code, it can be very interesting to vary the number of discretization of the lateral elements and see how it behaves.

```

1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %                                                                                                                                              %
3  % Topology Optimization Using Matlab                                                                 %
4  % ADVANCED3D.m                                                                                     %
5  %                                                                                                                                              %
6  % Delft University of Technology, Department PME                                                 %
7  % Master of Science Thesis Project                                                                %
8  %                                                                                                                                              %
9  % Stefan Broxterman                                                                                %
10 %                                                                                                                                              %
11 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
12 clc; clf; close all; clear X; clear prog;
13 %% DEFINE OPTIMIZATION VARIABLES
14 var = 6;          % [1 = mesh, 2 = penalty, 3 = filter radius, 4 =
    volume fraction, 5 = filter method, 6 = evolution]
15 nxvec = [30,60,90,120]; % horizontal elements vector
16 nyvec = [10,20,30,40]; % vertical elements vector
17 nzvec = [1,2,3,5];     % lateral elements vector
18 volvec = [0.2 0.35 0.5 0.65]; % volume fraction vector
19 rminvec = [1,1.25,1.5,3]; % filter size vector
20 penvec = [1, 2, 3, 5]; % penalty vector
21 filvec = [0, 1, 2];   % filter vector
22 evolvec = [0.05, 0.25, 0.5, 1]; % evolution fraction vector
23 %% SET DEFAULT VALUES
24 nx = nxvec(1);      % default number of horizontal elements
25 ny = nyvec(1);      % default number of vertical elements
26 nz = nzvec(3);      % default number of lateral elements
27 vol = volvec(3);    % default number of volume fraction
28 pen = penvec(3);    % default penalty
29 rmin = rminvec(3);  % default filter radius
30 fil = filvec(1);    % default filter method
31 q = 1;              % gray-scale parameter
32 qmax = 2;           % maximum gray-scale parameter
33 %% SET OPTIMIZATION VALUES
34 ex = [30,60,90,120]; % vector size for pre-allocating space
35 figend = 4;          % set total of varying values
36 label = ['a','b','c','d','e']; % graphic label
37 %% PRE-ALLOCATE SPACE
38 loops = zeros(1,size(ex,2)); % initial loops matrix
39 obj = zeros(1,size(ex,2)); % initial objective matrix
40 t = zeros(1,size(ex,2)); % initial time matrix
41 Y = zeros(size(ex,2),5); % initial results matrix
42 if var == 6          % for evolution scheme, BasicK.m only needs to
    ...

```

```

43     BASIC3D           % run one time only
44 end
45 %% START LOOP
46 for fig = 1:figend   % start iteration loop
47     tic;             % start timer
48     if var ~= 6      % for non-evolution scheme, run below
49         clear X; clear C; % clear results matrix for each run
50         if var == 1  % differentiation on number of elements
51             nx = nxvec(fig); % pick each horizontal value
52             ny = nyvec(fig); % pick each vertical value
53             nz = nzvec(fig);
54         elseif var == 2 % differentiation on penalty
55             pen = penvec(fig); % pick each penalty
56         elseif var == 3 % differentiation on filter radius
57             rmin = rminvec(fig); % pick each rmin
58         elseif var == 4 % differentiation on filter method
59             vol = volvec(fig); % pick each filter method
60         elseif var == 5 % differentiation on filter method
61             fil = filvec(fig); % pick each filter method
62         end
63         BASIC3D           % run Basic.m
64         loops(fig) = size(X,4); % number of iterations used
65         obj(fig) = c(iter); % store objective function
66         prog = X(:, :, :, loops(fig)); % store densities for progression
67         drawing
68     elseif var == 6    % store compliance for evolution vector
69         loops = size(X,4); % for evolutionary scheme, calculate rounded
70         ...
71         loop(1) = round(evolvec(1)*loops); % values of loops and store
72         ...
73         loop(2) = round(evolvec(2)*loops); % this loop number
74         loop(3) = round(evolvec(3)*loops);
75         loop(4) = round(evolvec(4)*loops);
76         prog = X(:, :, :, loop); % progression picture for each evolution
77         fraction
78     end
79 %% Set graphics
80 if draw == 1 || draw == 2 % check for drawing
81     H = get(gcf, 'Position'); % get position of figure
82 else
83     H = [680,558,560,420]; % set size of figure(2) plot windows
84 end
85 H2 = figure(2); % plot window for progression pictures
86 set(H2, 'position', [H(1)+H(3) H(2) H(3) H(4)]); % place figure(2) next
87 to (1)
88 %% Draw progression plots
89 subplot(3,2,fig+2) % plot each differentiation
90 if var == 6 % evolution needs different plotting
91     [nely,nelx,nelz] = size(prog(:, :, :, fig));
92     hx = 1; hy = 1; hz = 1; % User-defined unit element
93     size
94     face = [1 2 3 4; 2 6 7 3; 4 3 7 8; 1 5 8 4; 1 2 6 5; 5 6 7 8];
95     for k = 1:nelz

```

```

90         z = (k-1)*hz;
91         for i = 1:nelx
92             xplot = (i-1)*hx;
93             for j = 1:nely
94                 y = nely*hy - (j-1)*hy;
95                 if (prog(j,i,k,fig) > 0.5) % User-defined display
96                     density threshold
97                     vert = [xplot y z; xplot y-hx z; xplot+hx y-hx z;
98                             xplot+hx y z; xplot y z+hx;xplot y-hx z+hx;
99                             xplot+hx y-hx z+hx;xplot+hx y z+hx];
100                    vert(:,[2 3]) = vert(:,[3 2]); vert(:,2,:) = -
101                        vert(:,2,:);
102                    patch('Faces',face,'Vertices',vert,'FaceColor'
103                        ,[0.2+0.8*(1-prog(j,i,k,fig)),0.2+0.8*(1-prog(
104                            j,i,k,fig)),0.2+0.8*(1-prog(j,i,k,fig))]);
105                    hold on;
106                end
107            end
108        end
109    else % plot advanced graphs
110        [nely,nelx,nelz] = size(prog);
111        hx = 1; hy = 1; hz = 1; % User-defined unit element size
112        face = [1 2 3 4; 2 6 7 3; 4 3 7 8; 1 5 8 4; 1 2 6 5; 5 6 7 8];
113        for k = 1:nelz
114            z = (k-1)*hz;
115            for i = 1:nelx
116                xplot = (i-1)*hx;
117                for j = 1:nely
118                    y = nely*hy - (j-1)*hy;
119                    if (prog(j,i,k) > 0.5) % User-defined display
120                        density threshold
121                        vert = [xplot y z; xplot y-hx z; xplot+hx y-hx z;
122                                xplot+hx y z; xplot y z+hx;xplot y-hx z+hx;
123                                xplot+hx y-hx z+hx;xplot+hx y z+hx];
124                        vert(:,[2 3]) = vert(:,[3 2]); vert(:,2,:) = -
125                            vert(:,2,:);
126                        patch('Faces',face,'Vertices',vert,'FaceColor'
127                            ,[0.2+0.8*(1-prog(j,i,k)),0.2+0.8*(1-prog(j,i,
128                                k)),0.2+0.8*(1-prog(j,i,k))]);
129                        hold on;
130                    end
131                end
132            end
133        end
134    end
135    axis equal; axis tight;
136    set(gca,'XTick',[],'YTick',[],'ZTick',[],'XTicklabel',[],...
137        'YTicklabel',[],'ZTicklabel',[],'xcolor','w','ycolor','w','zcolor'
138        ',','w');
139    view([30,30]);
140    xlabel(sprintf('c = %.2f',c(iter)),'Color','k');
141    drawnow;

```

```

130 hold on
131 if var == 6 % evolution needs different plotting
132 xlabel(sprintf('c = %.2f',C(loop(fig))), 'color', 'k')
133 else
134 xlabel(sprintf('c = %.2f',obj(fig)), 'color', 'k')
135 end
136 ylabel(sprintf('%s', (label(fig+1))), ...
137 'rot',0, 'color', 'k', 'FontSize', 11)
138 %% Store compliance
139 if var ~= 6 % store compliance for further plotting
140 if fig == 1
141 C1 = C;
142 elseif fig == 2
143 C2 = C;
144 elseif fig == 3
145 C3 = C;
146 elseif fig == 4
147 C4 = C;
148 end
149 end
150 %% Draw graphics
151 xbox = [0.5 nx+0.5];
152 ybox = [0.5 ny+0.5];
153 xwidth = xbox(2)-xbox(1);
154 ywidth = ybox(2)-ybox(1);
155 t(fig) = toc;
156 %% Output
157 if var ~= 6 % output results for non-evolutionary schemes
158 Y(fig,:) = [fig ex(fig) loops(fig) obj(fig) t(fig)];
159 if fig == figend
160 Y
161 end;
162 end
163 %% Compliance graphs
164 if var ~= 6
165 H3 = figure(3);
166 set(H3, 'position', [H(1)-H(3) H(2) H(3) H(4)]); % place figure(2)
167 next to (1)
168 hold on
169 switch fig
170 case 1 % first variable
171 plot(1:length(C1),C1, 'b:', 'LineWidth', 2)
172 xaxmax = mean(length(C1));
173 yaxmax = max(max(C1));
174 yaxmin = min(C1);
175 if var == 1
176 legend(sprintf('mesh = %g x %g ',nxvec(1),nyvec(1)))
177 elseif var == 2
178 legend(sprintf('pen = %g',penvec(1)))
179 elseif var == 3
180 legend(sprintf('Rmin = %g',rminvec(1)))
181 elseif var == 4
182 legend(sprintf('vol = %g',volvec(1)))

```

```

182         elseif var == 5
183             legend(sprintf('filter = Sensitivity'))
184         end
185     case 2         % second variable
186         plot(1:length(C2),C2,'r--','LineWidth',2)
187         xaxmax = mean([length(C1) length(C2)]);
188         yaxmax = max([max(C1) max(C2)]);
189         yaxmin = min(min([C1 C2]));
190         if var == 1
191             legend(sprintf('mesh = %g x %g',nxvec(1),nyvec(2)),
192                    sprintf('mesh = %g x %g',nxvec(2),nyvec(2)))
193         elseif var == 2
194             legend(sprintf('pen = %g ',penvec(1)),sprintf('pen =
195                 %g',penvec(2)))
196         elseif var == 3
197             legend(sprintf('Rmin = %g ',rminvec(1)),sprintf('Rmin
198                 = %g',rminvec(2)))
199         elseif var == 4
200             legend(sprintf('vol = %g ',volvec(1)),sprintf('vol =
201                 %g',volvec(2)))
202         elseif var == 5
203             legend(sprintf('filter = Sensitivity'),sprintf('
204                 filter = Density'))
205         end
206     case 3         % third variable
207         plot(1:length(C3),C3,'k','LineWidth',2)
208         xaxmax = mean([length(C1) length(C2) length(C3)]);
209         yaxmax = max([max(C1) max(C2) max(C3)]);
210         yaxmin = min(min([C1 C2 C3]));
211         if var == 1
212             legend(sprintf('mesh = %g x %g',nxvec(1),nyvec(2)),
213                    sprintf('mesh = %g x %g',nxvec(2),nyvec(2)),
214                    sprintf('mesh = %g x %g',nxvec(3),nyvec(3)))
215         elseif var == 2
216             legend(sprintf('pen = %g ',penvec(1)),sprintf('pen =
217                 %g',penvec(2)),sprintf('pen = %g',penvec(3)))
218         elseif var == 3
219             legend(sprintf('Rmin = %g ',rminvec(1)),sprintf('Rmin
220                 = %g',rminvec(2)),sprintf('Rmin = %g',rminvec(3))
221             )
222         elseif var == 4
223             legend(sprintf('vol = %g ',volvec(1)),sprintf('vol =
224                 %g',volvec(2)),sprintf('vol = %g',volvec(3)))
225         elseif var == 5
226             legend(sprintf('filter = Sensitivity'),sprintf('
227                 filter = Density'),sprintf('filter = Heaviside'))
228         end
229     case 4         % fourth variable
230         plot(1:length(C4),C4,'g-.','LineWidth',2)
231         xaxmax = mean([length(C1) length(C2) length(C3) length(C4)
232             )]);
233         yaxmax = max([max(C1) max(C2) max(C3) max(C4)]);
234         yaxmin = min(min([C1 C2 C3 C4]));

```

```

222         if var == 1
223             legend(sprintf('mesh = %g x %g',nxvec(1),nyvec(2)),
                    sprintf('mesh = %g x %g',nxvec(2),nyvec(2)),
                    sprintf('mesh = %g x %g',nxvec(3),nyvec(3)),
                    sprintf('mesh = %g x %g',nxvec(4),nyvec(4)))
224         elseif var == 2
225             legend(sprintf('pen = %g ',penvec(1)),sprintf('pen =
                    %g',penvec(2)),sprintf('pen = %g',penvec(3)),
                    sprintf('pen = %g',penvec(4)))
226         elseif var == 3
227             legend(sprintf('Rmin = %g ',rminvec(1)),sprintf('Rmin
                    = %g',rminvec(2)),sprintf('Rmin = %g',rminvec(3))
                    ,sprintf('Rmin = %g',rminvec(4)))
228         elseif var == 4
229             legend(sprintf('vol = %g ',volvec(1)),sprintf('vol =
                    %g',volvec(2)),sprintf('vol = %g',volvec(3)),
                    sprintf('vol = %g',volvec(4)))
230         end
231     end
232     xlabel('Number of iterations')
233     ylabel('Compliance')
234     if exist('pcon','var') == 0,
235         yaxmax = mean([yaxmin yaxmax]);
236     elseif pcon == 0
237         yaxmax = mean([yaxmin yaxmax]);
238     end
239     axis([0 xaxmax 0.95*yaxmin yaxmax])
240     elseif var == 6
241         H3 = figure(3);
242         set(H3,'position',[H(1)-H(3) H(2) H(3) H(4)]); % place figure(2)
                next to (1)
243         hold on
244         plot(C)
245         xlabel('Number of iterations')
246         ylabel('Compliance')
247         axis([0 length(C) 0.9*min(C) max(C)])
248     end
249 end
250 %% STORE RESULTS
251 disp('Y = i, penalty, loops, objective, time')
252 if var == 1 % mesh refinement
253     Ymesh = Y; % store result matrix
254     save('MeshRefinementY.mat','Y');
255 elseif var == 2 % penalty
256     Ypenal = Y; % store result matrix
257     save('PenaltyY.mat','Y');
258 elseif var == 3 % filter radius
259     Yfilter = Y; % store result matrix
260     save('FilterY.mat','Y');
261 elseif var == 4 % volume fraction
262     Yvolume = Y; % store result matrix
263     save('VolumeY.mat','Y');
264 end

```

```
265 %% DRAW DESIGN PROBLEM
266 figure(2)
267 subplot(3,2,(1:2)) % plot the initial mechanical problem
268 rectangle('Position',[xbox(1),ybox(1),xwidth,ywidth],...
269          'FaceColor',[0.5 0.5 0.5])
270 axis equal; axis tight;
271 set(gca,'XTick',[],'YTick',[],'XTicklabel',[],...
272       'YTicklabel',[],'xcolor','w','ycolor','w')
273 ylabel(sprintf('%s',(label(1))), 'rot',0,'color','k','FontSize',11)
274 draw_arrow([xbox(2) ybox(1)],[xbox(2) -0.25*ywidth],1)
275 rectangle('Position',[-0.1*xwidth,ybox(1)-0.1*ywidth,...
276          0.1*xwidth,1.2*ywidth],'FaceColor',[0 0 0],'LineWidth',3)
```

B.6 BASIC COMPLIANT MECHANISMS.m

In this section, the complete code of producing a micro-gripper is made available. Using the predefined discretization, a void region is declared as restrictive region, to allow a gripper mechanism. Using this boundary condition, the design problem of Figure 3-12a can be calculated. Displacement field is plotted on the go.

```

1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %                                                                                                                                                                %
3  % Topology Optimization Using Matlab                                                                                                                                 %
4  % BASIC_COMPLIANCE.m                                                                                                                                              %
5  %                                                                                                                                                                %
6  % Delft University of Technology, Department PME                                                                                                                                                            %
7  % Master of Science Thesis Project                                                                                                                                                                       %
8  %                                                                                                                                                                %
9  % Stefan Broxterman                                                                                                                                                                                       %
10 %                                                                                                                                                                %
11 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
12 %
13 tic                                                                                                                                                                % start timer
14 %% DEFINE PARAMETERS
15 adv = 0;                                                                                                                                                            % use advanced function [0 = off, 1 = on]
16 if adv == 0                                                                                                                                                            % define parameters at behalf of the advanced
    function
17     nx = 120;                                                                                                                                                            % number of elements horizontal
18     ny = 60;                                                                                                                                                            % number of elements vertical
19     vol = 0.2;                                                                                                                                                            % volume fraction [0-1]
20     pen = 4;                                                                                                                                                            % penalty
21     rmin = 1.4;                                                                                                                                                            % filter size
22     fil = 1;                                                                                                                                                            % filter method [0 = sensitivity filtering, 1 =
        density filtering, 2 = heaviside filtering]
23     clc; clf; close all; clear X;
24 end
25 %% DEFINE SOLUTION METHOD
26 sol = 0;                                                                                                                                                            % solution method [0 = oc(sens), 1 = mma]
27 pcon = 1;                                                                                                                                                            % use continuation method [0 = off, 1 = on]
28 %% DEFINE CALCULATION
29 tol = 0.01;                                                                                                                                                            % tolerance for convergence criterion [0.01]
30 move = 0.1;                                                                                                                                                            % move limit for lagrange [0.2]
31 pcinc = 1.03;                                                                                                                                                            % penalty continuation increasing factor [1.03]
32 piter = 20;                                                                                                                                                            % number of iteration for starting penalty [20]
33 miter = 1000;                                                                                                                                                            % maximum number of iterations [1000]
34 sym = 2;                                                                                                                                                            % symmetry [0 = off, 1 = x-axis, 2 = y-axis]
35 def = 1;                                                                                                                                                            % plot deformations [0 = off, 1 = on]
36 %% DEFINE OUTPUT
37 draw = 1;                                                                                                                                                            % plot iterations [0 = off, 1 = on]
38 dis = 1;                                                                                                                                                            % display iterations [0 = off, 1 = on]
39 %% DEFINE MATERIAL
40 E = 1;                                                                                                                                                            % young's modulus of solid [1]
41 Emin = 1e-9;                                                                                                                                                            % young's modulus of void [1e-9]
42 nu = 0.3;                                                                                                                                                            % poisson ratio [0.3]

```

```

43 rho = 0e-3;           % density [0e-3]
44 g = 9.81;           % gravitational acceleration [9.81]
45 Kin = 0.01 ;        % spring stiffness at input force [5e-4]
46 Kout = 0.01;        % spring stiffness at output force [5e-4]
47 %% DEFINE FORCE
48 Uin = 2*(ny+1)-1;    % input force node
49 Uout = 2*(nx+1)*(ny+1)-round((2/6)*ny)-2; % output force
50 Fe = [Uin Uout];    % element of force application [Uin Uout]
51 Fn = [1 2];         % number of applied force locations [1 2]
52 Fv = [1 -1];        % value of applied force [1 -1]
53 %% DEFINE SUPPORTS
54 fix = [1:4 (Uin+1):2*(ny+1):round((5/6)*(Uout+1))]; % create symmetry
55 %% DEFINE ELEMENT RESTRICTIONS
56 shap = 1;           % [0 = no restrictions, 1 = circle, 2 = custom]
57 area = 0;           % [0 = no material (passive), 1 = material (
    active)]
58 nodr = (round(ny/2)+(0:ny:(nx-1)*ny)); % custom restricted nodes
59 %% PREPARE FINITE ELEMENT
60 N = 2*(nx+1)*(ny+1); % total element nodes
61 all = 1:2*(nx+1)*(ny+1); % all degrees of freedom
62 free = setdiff(all,fix); % free degrees of freedom
63 A11 = [12 3 -6 -3; 3 12 3 0; -6 3 12 -3; -3 0 -3 12]; % fem
64 A12 = [-6 -3 0 3; -3 -6 -3 -6; 0 -3 -6 3; 3 -6 3 -6]; % fem
65 B11 = [-4 3 -2 9; 3 -4 -9 4; -2 -9 -4 -3; 9 4 -3 -4]; % fem
66 B12 = [ 2 -3 4 -9; -3 2 9 -2; 4 9 2 3; -9 -2 3 2]; % fem
67 Ke = 1/(1-nu^2)/24*([A11 A12;A12' A11]+nu*[B11 B12;B12' B11]); % element
    stiffness matrix
68 nodes = reshape(1:(nx+1)*(ny+1),1+ny,1+nx); % create node numer matrix
69 dofvec = reshape(2*nodes(1:end-1,1:end-1)+1,nx*ny,1); % create dof vector
70 dofmat = repmat(dofvec,1,8)+repmat([0 1 2*ny+[2 3 0 1] -2 -1],nx*ny,1); %
    create dof matrix
71 iK = reshape(kron(dofmat,ones(8,1))',64*nx*ny,1); % build sparse i
72 jK = reshape(kron(dofmat,ones(1,8))',64*nx*ny,1); % build sparse j
73 %% PREPARE FILTER
74 iH = ones(nx*ny*(2*(ceil(rmin)-1)+1)^2,1); % build sparse i
75 jH = ones(size(iH)); % create sparse vector of ones
76 kH = zeros(size(iH)); % create sparse vector of zeros
77 m = 0; % index for filtering
78 for i = 1:nx % for each element calculate distance between ...
79     for j = 1:ny % elements' center for filtering
80         r1 = (i-1)*ny+j; % sparse value i
81         for k = max(i-(ceil(rmin)-1),1):min(i+(ceil(rmin)-1),nx) %
            center of element
82             for l = max(j-(ceil(rmin)-1),1):min(j+(ceil(rmin)-1),ny) %
                center of element
83                 r2 = (k-1)*ny+l; % sparse value 2
84                 m = m+1; % update index for filtering
85                 iH(m) = r1; % sparse vector for filtering
86                 jH(m) = r2; % sparse vector for filtering
87                 kH(m) = max(0,rmin-sqrt((i-k)^2+(j-l)^2)); % weight
                    factor
88             end
89         end

```

```

90     end
91 end
92 H = sparse(iH,jH,kH); % build filter
93 Hs = sum(H,2); % summation of filter
94 %% DEFINE ELEMENT RESTRICTIONS
95 x = repmat(vol,ny,nx); % initial material distribution
96 if shap == 0 % no restrictions
97     efree = (1:nx*ny)'; % all elements are free
98     eres= []; % no restricted elements
99 elseif shap == 1 % restrictions
100    rest = zeros(ny,nx); % pre-allocate space
101    % for i = 1:nx % start loop
102    % for j = 1:ny % for each element
103    % if sqrt((j-ny/2)^2+(i-nx/4)^2) < ny/2.5 % circular
      restriction
104    % rest(j,i) = 1; % write restriction
105    % if rest(j,i) == area % check for restriction
106    % x(j,i) = area; % store restrictions in material
      distribution
107    % end
108    % end
109    % end
110    % end
111    for i = round((5/6)*nx):nx
112        for j = round((5/6)*ny):ny
113            rest(j,i) = area;
114            x(j,i) = area;
115        end
116    end
117    efree = find(rest ~= 1); % set free elements
118    eres = find(rest == 1); % set restricted elements
119 end
120 if fil == 0 || fil == 1 % sensitivity, density filter
121    xF = x; % set filtered design variables
122 elseif fil == 2 % heaviside filter
123    beta = 1; % hs filter
124    xTilde = x; % hs filter
125    xF = 1-exp(-beta*xTilde)+xTilde*exp(-beta); % set filtered design
      space
126 end
127 xFree = xF(efree); % define free design matrix
128 %% DEFINE STRUCTURAL
129 Fsiz = size(Fe,2); % size of load vector
130 F = sparse(Fe,Fn,Fv,N,Fsiz); % define load vector
131 %% DEFINE MMA PARAMETERS
132 m = 1; % number of constraint functions
133 n = size(xFree(:,1)); % number of variables
134 xmin = zeros(n,1); % minimum values of x
135 xmax = ones(n,1); % maximum values of x
136 xold1 = zeros(n,1); % previous x, to monitor convergence
137 xold2 = xold1; % used by mma to monitor convergence
138 df0dx2 = zeros(n,1); % second derivative of the objective function
139 dfdx2 = zeros(1,n); % second derivative of the constraint function

```

```

140 low = xmin; % lower asymptotes from the previous iteration
141 upp = xmax; % upper asymptotes from the previous iteration
142 a0 = 1; % constant a_0 in mma formulation
143 a = zeros(m,1); % constant a_i in mma formulation
144 cmma = 1e3*ones(m,1); % constant c_i in mma formulation
145 d = zeros(m,1); % constant d_i in mma formulation
146 subs = 200; % maximum number of subsolv iterations
147 %% PRE-ALLOCATE SPACE
148 npx = zeros(length(fix),1)'; % pre-allocate constraint dots
149 npy = zeros(length(fix),1)'; % pre-allocate constraint dots
150 npfx = zeros(length(Fe),1)'; % pre-allocate force dots
151 npfy = zeros(length(Fe),1)'; % pre-allocate force dots
152 U = zeros(size(F)); % pre-allocate space displacement
153 c = zeros(miter,1); % pre-allocate objective vector
154 %% INITIALIZE LOOP
155 iter = 0; % initialize loop
156 diff = 1; % initialize convergence criterion
157 loopbeta = 1; % initialize beta-loop
158 %% START LOOP
159 while ((diff > tol) || (iter < piter+1)) && iter < miter % convergence
    criterion not met
160     loopbeta = loopbeta + 1; % iteration loop for hs filter
161     iter = iter+1; % define iteration
162     if pcon == 1 % use continuation method
163         if iter <= piter % first number of iterations...
164             p = 1; %... set penalty 1
165         elseif iter > piter % after a number of iterations...
166             p = min(pen,pcinc*p); % ... set continuation penalty
167         end
168     elseif pcon == 0 % not using continuation method
169         p = pen; % set penalty
170     end
171     %% Selfweight
172     if rho ~= 0 % gravity is involved
173         xP=zeros(ny,nx); % pre-allocate space
174         xP(xF>0.25) = xF(xF>0.25).^p; % normal penalization
175         xP(xF<=0.25) = xF(xF<=0.25).*(0.25^(p-1)); % below pseudo-density
176         Fsw = zeros(N,1); % pre-allocate self-weight
177         for i=1:nx*ny % for each element, set gravitational...
178             Fsw(dofmat(i,2:2:end))=Fsw(dofmat(i,2:2:end))-xF(i)*rho
                *9.81/4;
179         end % force to the attached nodes
180         Fsw=repmat(Fsw,1,size(F,2)); % set self-weight for load cases
181     elseif rho == 0 % no gravity
182         xP = xF.^p; % penalized design variable
183         Fsw = 0; % no selfweight
184     end
185     Ftot = F + Fsw; % total force
186     %% Finite element analysis
187     kK = reshape(Ke(:)*(Emin+xP(:))* (E-Emin),64*nx*ny,1); % create
        sparse vector k
188     K = sparse(iK,jK,kK); % combine sparse vectors
189     K = (K+K')/2; % build stiffness matrix

```

```

190 K(Uin,Uin) = K(Uin,Uin) + Kin; % add input spring stiffness
191 K(Uout,Uout) = K(Uout,Uout) + Kout; % add output spring stiffness
192 U(free,:) = K(free,free)\Ftot(free,:); % displacement solving
193 c(iter) = 0; % set compliance to zero
194 %% Calculate compliance and sensitivity
195 U1 = U(:,1); U2 = U(:,2);
196 c0 = reshape(sum((U1(dofmat)*Ke).*U2(dofmat),2),ny,nx);
197 c(iter) = U(Uout,1);
198 Sens = p*(E-Emin)*xF.^(p-1).*c0;
199 Senc = ones(ny,nx); % set constraint sensitivity
200 if fil == 0 % optimality criterion with sensitivity filter
201     Sens(:) = H*(x(:).*Sens(:))./Hs./max(1e-3,x(:)); % update
        filtered sensitivity
202 elseif fil == 1 % optimality criterion with density filter
203     Sens(:) = H*(Sens(:))./Hs; % update filtered sensitivity
204     Senc(:) = H*(Senc(:))./Hs; % update filtered sensitivity of
        constraint
205 elseif fil == 2 % optimality criterion with heaviside filter
206     dx = beta*exp(-beta*xTilde)+exp(-beta); % update hs parameter
207     Sens(:) = H*(Sens(:).*dx(:))./Hs; % update filtered sensitivity
208     Senc(:) = H*(Senc(:).*dx(:))./Hs; % update filtered sensitivity
        of constraint
209 end
210 %% Update design variables Optimality Criterion
211 if sol == 0 % use optimality criterion method
212     l1 = 0; % initial lower bound for lagrangian mulitplier
213     l2 = 1e9; % initial upper bound for lagrangian multiplier
214     while (l2-l1)/(l1+l2) > 1e-4 && l2 > 1e-40; % start loop
215         lag = 0.5*(l1+l2); % average of lagrangian interval
216         xnew = max(0,max(x-move,min(1,min(x+move,x.*(max(1e-10,-Sens
                ./lag)).^0.3))))); % update element densities
217         if fil == 0 % sensitivity filter
218             xF = xnew; % updated result
219         elseif fil == 1 % density filter
220             xF(:) = (H*xnew(:))./Hs; % updated filtered density
                result
221         elseif fil == 2 % heaviside filter
222             xTilde(:) = (H*xnew(:))./Hs; % set filtered density
223             xF(:) = 1-exp(-beta*xTilde)+xTilde*exp(-beta); % updated
                result
224         end
225         if shap == 1 % restriction is on
226             xF(rest==1) = area; % set restricted area
227         end
228         if sum(xF(:)) > vol*nx*ny; % check for optimum
229             l1 = lag; % update lower bound to average
230         else
231             l2 = lag; % update upper bound to average
232         end
233     end
234     %% Method of moving asymptotes
235 elseif sol == 1 % use mma solver
236     xval = xFree(:); % store current design variable for mma

```

```

237     if iter == 1      % for the first iteration...
238         cscale = 1/c(iter); % ...set scaling factor for mma solver
239     end
240     f0 = c(iter)*cscale; % objective at current design variable for
        mma
241     df0dx = Sens(efree)*cscale; % store sensitivity for mma
242     f = (sum(xF(:))/(vol*nx*ny)-1); % normalized constraint function
243     dfdx = Senc(efree)'/(vol*ny*nx); % derivative of the constraint
        function
244     [xmma,~,~,~,~,~,~,~,~,low,upp] = ...
245         mmasub(m,n,iter,xval,xmin,xmax,xold1,xold2, ...
246             f0,df0dx,df0dx2,f,dfdx,dfdx2,low,upp,a0,a,cmma,d,subs); % mma
        solver
247     xold2 = xold1; % used by mma to monitor convergence
248     xold1 = xFree(:); % previous x, to monitor convergence
249     xnew = xF; % update result
250     xnew(efree) = xmma; % include restricted elements
251     xnew = reshape(xnew,ny,nx); % reshape xmma vector to original
        size
252     if fil == 0      % sensitivity filter
253         xF = xnew; % update design variables
254     elseif fil == 1 % density filter
255         xF(:) = (H*xnew(:))./Hs; % update filtered densities result
256     elseif fil == 2 % heaviside filter
257         xTilde(:) = (H*xnew(:))./Hs; % filtered result
258         xF(:) = 1-exp(-beta*xTilde)+xTilde*exp(-beta); % update design
        variable
259     end
260     if shap == 1    % if restrictions enableed
261         xF(rest==1) = area; % set restricted area
262     end
263
264 end
265 xFree = xnew(efree); % set non-restricted area
266 diff = max(abs(xnew(:)-x(:))); % difference of maximum element change
267 x = xnew; % update design variable
268 if fil == 2 && beta < 512 && pen == p(end) && (loopbeta >= 50 || diff
    <= tol) % hs filter
269     beta = 2*beta; % increase beta-factor
270     fprintf('beta now is %3.0f\n',beta) % display increase of b-
        factor
271     loopbeta = 0; % set hs filter loop to zero
272     diff = 1; % set convergence to initial value
273 end
274 %% Store results into database X
275 X(:, :, iter) = xF; % each element value x is stored for each
    iteration
276 C(iter) = c(iter); % each compliance is stored for each iteration
277 assignin('base', 'X', X); % each iteration (3rd dimension)
278 assignin('base', 'C', C); % each iteration (3rd dimension)
279 %% Results
280 if dis == 1 % display iterations

```

```

281     disp([' Iter:' sprintf('%4i',iter) ' Uin:' sprintf('%6.2f',U(Uin)
282         ) ...
        ' Uout:' sprintf('%6.2f',c(iter)) ' Con:' sprintf('%6.2f',
        diff) ' Vol:' sprintf('%6.2f',mean(xF(:))) ' Diff:'
        sprintf('%6.3f',diff)]);
283 end
284 if draw == 1           % plot iterations
285     figure(1)
286     subplot(2,1,1)
287     colormap(gray); imagesc(1-xF);
288     set(gca, 'XTick', [], 'YTick', [], 'XTicklabel', [], ...
289         'YTicklabel', [], 'xcolor', 'w', 'ycolor', 'w')
290     xlabel(sprintf('c = %.2f',c(iter)), 'Color', 'k')
291     drawnow;
292     hold on
293     if iter == 1
294         axis equal; axis tight;
295         % Plot coloured dots for constraints
296         for i = 1:length(fix)
297             npx(i) = ceil(fix(i)/(2*(ny+1))) - 0.5;
298             nplot = ceil(fix(i)/2);
299             while nplot > (ny+1)
300                 nplot = nplot - (ny+1);
301             end
302             npy(i) = nplot - 0.5;
303         end
304         plot(npx, npy, 'r.', 'MarkerSize', 20)
305         % Plot coloured dots for force application
306         for i = 1:length(Fe)
307             npfx(i) = ceil(Fe(i)/(2*(ny+1))) - 0.5;
308             nplot = ceil(Fe(i)/2);
309             while nplot > (ny+1)
310                 nplot = nplot - (ny+1);
311             end
312             npfy(i) = nplot - 0.5;
313         end
314         plot(npfx, npfy, 'g.', 'MarkerSize', 20)
315     end
316     % Plot compliance plot
317     figure(1)
318     subplot(2,1,2)
319     plot(c(1:iter))
320     xaxmax = c(iter);
321     yaxmax = max(c);
322     yaxmin = min(c(1:iter));
323     if pcon == 0
324         yaxmax = mean([yaxmin yaxmax]);
325     end
326     ylim([0.95*yaxmin yaxmax])
327     xlim([0 iter+10])
328     figure(2)
329     if sym ~= 0           % apply symmetry
330         if sym == 1       % symmetry around x-axis

```

```

331         xFlip = fliplr(xF);
332         xFlipplot = [xFlip xF];
333     end
334     if sym == 2      % symmetry around y-axis
335         xFlip = flip(xF);
336         xFlipplot = [xF; xFlip];
337     end
338     colormap gray
339     imagesc(1-xFlipplot)
340     axis equal
341     axis off
342 end
343 end
344 end
345 %% ONLY DISPLAY FINAL RESULT
346 if dis == 0      % display final result
347     disp([' Iter:' sprintf('%4i',iter) ' Uin:' sprintf('%6.2f',U(Uin))
348         ...
349         ' Uout:' sprintf('%6.2f',c(iter)) ' Con:' sprintf('%6.2f',diff) '
350         Vol:' sprintf('%6.2f',mean(xF(:))) ' Diff:' sprintf('%6.3f',
351         diff)]);
352 end
353 if draw == 0      % plot final result
354     figure(1)
355     subplot(2,1,1)
356     colormap(gray); imagesc(1-xF);
357     axis equal; axis tight;
358     set(gca, 'XTick', [], 'YTick', [], 'XTicklabel', [], ...
359         'YTicklabel', [], 'xcolor', 'w', 'ycolor', 'w')
360     xlabel(sprintf('c = %.2f',c(iter)), 'Color', 'k')
361     drawnow;
362     hold on
363     %% Plot coloured dots for constraints
364     for i = 1:length(fix)
365         npx(i) = ceil(fix(i)/(2*(ny+1))) - 0.5;
366         nplot = ceil(fix(i)/2);
367         while nplot > (ny+1)
368             nplot = nplot - (ny+1);
369         end
370         npy(i) = nplot - 0.5;
371     end
372     plot(npx, npy, 'r.', 'MarkerSize', 20)
373     %% Plot coloured dots for force application
374     for i = 1:length(Fe)
375         npfx(i) = ceil(Fe(i)/(2*(ny+1))) - 0.5;
376         nplot = ceil(Fe(i)/2);
377         while nplot > (ny+1)
378             nplot = nplot - (ny+1);
379         end
380         npfy(i) = nplot - 0.5;
381     end
382     plot(npfx, npfy, 'g.', 'MarkerSize', 20)
383     %% Plot compliance plot

```

```

381     if adv == 0
382         figure(1)
383         subplot(2,1,2)
384         plot(c(1:iter))
385         xaxmax = c(iter);
386         yaxmax = max(c);
387         yaxmin = min(c(1:iter));
388         if pcon == 0
389             yaxmax = mean([yaxmin yaxmax]);
390         end
391         ylim([0.95*yaxmin yaxmax])
392         xlim([0 iter+10])
393     end
394     figure(2)
395     if sym ~= 0           % apply symmetry
396         if sym == 1      % symmetry around x-axis
397             xFlip = fliplr(xF);
398             xFliplot = [xFlip xF];
399         end
400         if sym == 2      % symmetry around y-axis
401             xFlip = flip(xF);
402             xFliplot = [xF; xFlip];
403         end
404         colormap gray
405         imagesc(1-xFliplot)
406         axis equal
407         axis off
408     end
409 end
410 %% PLOTTING DISPLACEMENT (COMPLIANT MECHANISMS)
411 if def == 1
412     figure(2)
413     xaxis = get(gca, 'XLim');
414     yaxis = get(gca, 'YLim');
415     figure(3)
416     clear mov
417     colormap(gray);
418     Umov = 1;           % Start movie counter
419     Umax = 0.05;       % Define maximum displacement
420     for Udisp = linspace(0,Umax,1); % Vary input displacement
421         clf
422         for ely = 1:ny % plot displacements...
423             for elx = 1:nx % for each element...
424                 if xF(ely,elx) > 0 % exclude white regions for plotting
425                     purposes
426                         n1 = (ny+1)*(elx-1)+ely;
427                         n2 = (ny+1)* elx +ely;
428                         Ue = -Udisp*U([2*n1-1;2*n1; 2*n2-1;2*n2; 2*n2+1;2*n2
429                             +2; 2*n1+1;2*n1+2],1);
430                         ly = ely-1; lx = elx-1;
431                         xx = [Ue(1,1)+lx Ue(3,1)+lx+1 Ue(5,1)+lx+1 Ue(7,1)+lx
432                             ]';

```

```
430         yy = [-Ue(2,1)-ly -Ue(4,1)-ly -Ue(6,1)-ly-1 -Ue(8,1)-  
431             ly-1]';  
         patch([xx xx],[yy+ny -yy-ny],[-xF(ely,elx) -xF(ely,  
432             elx)], 'LineStyle', 'none');  
     end  
433     end  
434     end  
435     xlim(xaxis)  
436     ylim(yaxis-ny)  
437     drawnow  
438     mov(Umov) = getframe(3); % movie  
439     Umov = Umov +1; % update counter  
440     end  
441  
442     movlip = flip(mov); % create symmetry  
443     movull = [mov movlip]; % create symmetry  
444     FileName = ['Compliant_',datestr(now, 'ddmm_HHMMSS'),'avi']; %  
         dynamic filename  
445     movie2avi(movull, FileName, 'compression', 'None', 'FPS', 10); % save  
         video  
446     end  
447     toc % stop timer
```

B.7 Design of Supports.m

In this section, the complete code of producing bridge examples is available. A distributed vertical force at the top, and a user-friendly configuration interface can be used to calculate design of support, including a pre-defined cost distribution. The produced picture in Figure 4-4 can be made immediately by running this code.

```

1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %                                                                 %
3  % Topology Optimization Using Matlab %
4  % BRIDGE.m %
5  % %
6  % Delft University of Technology, Department PME %
7  % Master of Science Thesis Project %
8  % %
9  % Stefan Broxterman %
10 % %
11 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
12 %
13 tic % start timer
14 %% DEFINE PARAMETERS
15 adv = 0; % use advanced function [0 = off, 1 = on]
16 if adv == 0 % define parameters at behalf of the advanced
    function
17     nx = 80; % number of elements horizontal
18     ny = 40; % number of elements vertical
19     vol = 0.2; % volume fraction [0-1]
20     pen = 3; % penalty
21     rmin = 1.5; % filter size
22     fil = 1; % filter method [0 = sensitivity filtering, 1 =
        density filtering, 2 = heaviside filtering]
23     clc; clf; close all; clear X; clear Z; % clear workspace
24 end
25 %% DEFINE SOLUTION METHOD
26 sol = 1; % solution method [0 = oc(sens), 1 = mma]
27 pcon = 0; % use continuation method [0 = off, 1 = on]
28 %% DEFINE CALCULATION
29 tol = 0.01; % tolerance for convergence criterion [0.01]
30 move = 0.2; % move limit for lagrange [0.2]
31 pcinc = 1.03; % penalty continuation increasing factor [1.03]
32 piter = 20; % number of iteration for starting penalty [20]
33 miter = 1000; % maximum number of iterations [1000]
34 plotiter = 5; % gap of iterations used to plot or draw
    iterations [5]
35 def = 0; % plot deformations [0 = off, 1 = on]
36 zplot = 0.99; % define treshold plotting supports [0.99]
37 %% DEFINE OUTPUT
38 draw = 2; % plot iterations [0 = off, 1 = on, 2 = partial]
39 dis = 2; % display iterations [0 = off, 1 = on, 2 =
    partial]
40 %% DEFINE MATERIAL

```

```

41 E = 1; % young's modulus of solid [1]
42 Emin = 1e-9; % young's modulus of void [1e-9]
43 nu = 0.3; % poisson ratio [0.3]
44 rho = 0e-3; % density [0e-3]
45 g = 9.81; % gravitational acceleration [9.81]
46 %% DEFINE FORCE
47 Fe = 2:2*(ny+1):2*(ny+1)*(nx+1); % element of force application [2:2*(ny
+1):2*(ny+1)*(nx+1)]
48 Fn = 1; % number of applied force locations [1]
49 Fv = -1; % value of applied force [-1]
50 %% DEFINE SUPPORTS
51 fix = [1:2 2*(ny+1)*nx+(1:2)]; % define fixed locations [1:2 2*(ny+1)*nx
+(1:2)]
52 %% DEFINE DESIGN OF SUPPORTS
53 supp = [1:ny (1:ny)+(nx-1)*ny ny:ny:nx*ny]; % support area [1:ny (1:ny)+(
nx-1)*ny ny:ny:nx*ny]
54 supp = unique(supp); % create unique support area
55 zvol = 0.2; % maximum support area [0.2]
56 cost = 1; % set maximum cost of supports [1]
57 k0 = 0.01; % spring stiffness for support stiffness [0.01]
58 q = 5; % penalty for support design [3]
59 zmin = 1e-4; % minimum support design variable [1e-4]
60 dist = 2; % cost distribution [0 = off, 1 = x-distributed,
2 = y-distribution]
61 %% DEFINE ELEMENT RESTRICTIONS
62 shap = 2; % [0 = no restrictions, 1 = circle, 2 = custom]
63 area = 1; % [0 = no material (passive), 1 = material (
active)]
64 nodr = [1:ny:nx*ny 2:ny:nx*ny]; % custom restricted nodes [1:ny:nx*ny]
65 %% PREPARE FINITE ELEMENT
66 N = 2*(nx+1)*(ny+1); % total element nodes
67 all = 1:2*(nx+1)*(ny+1); % all degrees of freedom
68 free = setdiff(all,fix); % free degrees of freedom
69 A11 = [12 3 -6 -3; 3 12 3 0; -6 3 12 -3; -3 0 -3 12]; % fem
70 A12 = [-6 -3 0 3; -3 -6 -3 -6; 0 -3 -6 3; 3 -6 3 -6]; % fem
71 B11 = [-4 3 -2 9; 3 -4 -9 4; -2 -9 -4 -3; 9 4 -3 -4]; % fem
72 B12 = [ 2 -3 4 -9; -3 2 9 -2; 4 9 2 3; -9 -2 3 2]; % fem
73 Ke = 1/(1-nu^2)/24*([A11 A12;A12' A11]+nu*[B11 B12;B12' B11]); % element
stiffness matrix
74 nodes = reshape(1:(nx+1)*(ny+1),1+ny,1+nx); % create node numer matrix
75 dofvec = reshape(2*nodes(1:end-1,1:end-1)+1,nx*ny,1); % create dof vector
76 dofmat = repmat(dofvec,1,8)+repmat([0 1 2*ny+[2 3 0 1] -2 -1],nx*ny,1); %
create dof matrix
77 iK = reshape(kron(dofmat,ones(8,1))',64*nx*ny,1); % build sparse i
78 jK = reshape(kron(dofmat,ones(1,8))',64*nx*ny,1); % build sparse j
79 %% PREPARE FILTER
80 iH = ones(nx*ny*(2*(ceil(rmin)-1)+1)^2,1); % build sparse i
81 jH = ones(size(iH)); % create sparse vector of ones
82 kH = zeros(size(iH)); % create sparse vector of zeros
83 m = 0; % index for filtering
84 for i = 1:nx % for each element calculate distance between ...
85     for j = 1:ny % elements' center for filtering
86         r1 = (i-1)*ny+j; % sparse value i

```

```

87     for k = max(i-(ceil(rmin)-1),1):min(i+(ceil(rmin)-1),nx) %
        center of element
88     for l = max(j-(ceil(rmin)-1),1):min(j+(ceil(rmin)-1),ny) %
        center of element
89         r2 = (k-1)*ny+1; % sparse value 2
90         m = m+1; % update index for filtering
91         iH(m) = r1; % sparse vector for filtering
92         jH(m) = r2; % sparse vector for filtering
93         kH(m) = max(0,rmin-sqrt((i-k)^2+(j-l)^2)); % weight
            factor
94     end
95 end
96 end
97 end
98 H = sparse(iH,jH,kH); % build filter
99 Hs = sum(H,2); % summation of filter
100 %% DEFINE ELEMENT RESTRICTIONS
101 x = repmat(vol,ny,nx); % initial material distribution
102 if shap == 0 % no restrictions
103     efree = (1:nx*ny)'; % all elements are free
104     eres= []; % no restricted elements
105 elseif shap == 1 % circular restrictions
106     rest = zeros(ny,nx); % pre-allocate space
107     for i = 1:nx % start loop
108         for j = 1:ny % for each element
109             if sqrt((j-ny/2)^2+(i-nx/4)^2) < ny/2.5 % circular
                restriction
110                 rest(j,i) = 1; % write restriction
111                 if rest(j,i) == area % check for restriction
112                     x(j,i) = area; % store restrictions in material
                        distribution
113             end
114         end
115     end
116 end
117 elseif shap == 2 % custom restrictions
118     rest = zeros(ny*nx,1); % pre-allocate space
119     for i = 1:length(nodr) % write restriction
120         resti = nodr(i); % write restriction
121         rest(resti) = 1; % write restriction
122     end
123     rest = reshape(rest,ny,nx);
124     for i = 1:nx % start loop
125         for j = 1:ny % for each element
126             if rest(j,i) == area % check for restriction
127                 x(j,i) = area; % store restrictions in material
                        distribution
128             end
129         end
130     end
131     efree = find(rest ~= 1); % set free elements
132     eres = find(rest == 1); % set restricted elements
133 end

```

```

134 if fil == 0 || fil == 1 % sensitivity, density filter
135     xF = x; % set filtered design variables
136 elseif fil == 2 % heaviside filter
137     beta = 1; % hs filter
138     xTilde = x; % hs filter
139     xF = 1-exp(-beta*xTilde)+xTilde*exp(-beta); % set filtered design
        space
140 end
141 xFree = xF(efree); % define free design matrix
142 %% DEFINE STRUCTURAL
143 Fsiz = size(Fe,2); % size of load vector
144 F = sparse(Fe,Fn,Fv,N,Fsiz); % define load vector
145 %% DESIGN OF SUPPORT DISTRIBUTION
146 xsiz = size(xFree(:,1)); % size of design variables
147 zsiz = size(supp,2); % size of support design variables
148 xzer = zeros(xsiz,1); % empty row of zeros for mma usage
149 zzer = zeros(zsiz,1); % empty row of zeros for mma usage
150 z = zeros(ny,nx); % create design of support domain
151 z(supp) = 1; %zvol; % plugin initial support design variables
152 zval = z'; % create vector of design variables
153 Si = 1; % counter
154 if dist == 1 % x-axis cost distribution
155     Scos = [linspace(1,cost,nx/2) linspace(cost,1,nx/2)]; % x-axis cost
        distribution
156     Scost = zeros(nx,nx); % create multiplication matrix
157     for i = 1:nx % create weighted cost matrix
158         Scost(Si,i) = Scos(i); % plug-in cost values
159         Si = Si+1; % update counter
160     end
161 elseif dist == 2 % y-axis cost distribution
162     Scos = [linspace(cost,1,ny/2) linspace(1,cost,ny/2)]; % y-axis cost
        distribution
163     Scost = zeros(ny,ny); % create multiplication matrix
164     for i = 1:ny % create weighted cost matrix
165         Scost(Si,i) = Scos(i); % plug-in cost values
166         Si = Si+1; % update counter
167     end
168 end
169 Adofsup = dofmat(supp,:); % degrees of freedom for support locations
170 Asup = unique(Adofsup(:)); % unique support locations
171 zF = z; % set design of support
172 zval = zval(zval ~= 0); % create configurable design of support vector
173 k1 = k0*eye(8); % reshape scalar to diagonal matrix
174 %% DEFINE MMA PARAMETERS
175 m = 2; % number of constraint functions
176 n = xsiz+zsiz; % number of variables
177 xmin = [1e-4*ones(xsiz,1);zmin*ones(zsiz,1)]; % minimum values of x
178 xmax = ones(n,1); % maximum values of x
179 xold1 = zeros(n,1); % previous x, to monitor convergence
180 xold2 = xold1; % used by mma to monitor convergence
181 df0dx2 = zeros(n,1); % second derivative of the objective function
182 dfdx2 = zeros(m,n); % second derivative of the constraint function
183 low = xmin; % lower asymptotes from the previous iteration

```

```

184 upp = xmax; % upper asymptotes from the previous iteration
185 a0 = 1; % constant a_0 in mma formulation [1]
186 a = zeros(m,1); % constant a_i in mma formulation
187 cmma = 1e3*ones(m,1); % constant c_i in mma formulation
188 d = zeros(m,1); % constant d_i in mma formulation
189 subs = 200; % maximum number of subsolv iterations [200]
190 %% PRE-ALLOCATE SPACE
191 npx = zeros(length(fix),1)'; % pre-allocate constraint dots
192 npy = zeros(length(fix),1)'; % pre-allocate constraint dots
193 npfx = zeros(length(Fe),1)'; % pre-allocate force dots
194 npfy = zeros(length(Fe),1)'; % pre-allocate force dots
195 npdx = zeros(length(nodes),1)'; % pre-allocate force dots
196 npdy = zeros(length(nodes),1)'; % pre-allocate force dots
197 U = zeros(size(F)); % pre-allocate space displacement
198 c = zeros(miter,1); % pre-allocate objective vector
199 %% INITIALIZE LOOP
200 iter = 0; % initialize loop
201 diff = 1; % initialize convergence criterion
202 loopbeta = 1; % initialize beta-loop
203 %% START LOOP
204 while ((diff > tol) || (iter < piter+1)) && iter < miter % convergence
    criterion not met
205     loopbeta = loopbeta + 1; % iteration loop for hs filter
206     iter = iter+1; % define iteration
207     if pcon == 1 % use continuation method
208         if iter <= piter % first number of iterations...
209             p = 1; %... set penalty 1
210         elseif iter > piter % after a number of iterations...
211             p = min(pen,pcinc*p); % ... set continuation penalty
212         end
213     elseif pcon == 0 % not using continuation method
214         p = pen; % set penalty
215     end
216     %% Selfweight
217     if rho ~= 0 % gravity is involved
218         xP=zeros(ny,nx); % pre-allocate space
219         xP(xF>0.25) = xF(xF>0.25).^p; % normal penalization
220         xP(xF<=0.25) = xF(xF<=0.25).*(0.25^(p-1)); % below pseudo-density
221         Fsw = zeros(N,1); % pre-allocate self-weight
222         for i=1:nx*ny % for each element, set gravitational...
223             Fsw(dofmat(i,2:2:end))=Fsw(dofmat(i,2:2:end))-xF(i)*rho
                *9.81/4;
224         end % force to the attached nodes
225         Fsw=repmat(Fsw,1,size(F,2)); % set self-weight for load cases
226     elseif rho == 0 % no gravity
227         xP = xF.^p; % penalized design variable
228         Fsw = 0; % no selfweight
229     end
230     Ftot = F + Fsw; % total force
231     %% Finite element analysis
232     kK = reshape(Ke(:)*(Emin+xP(:))*(E-Emin),64*nx*ny,1); % create
        sparse vector k
233     K = sparse(iK,jK,kK); % combine sparse vectors

```

```

234 K = (K+K')/2; % build stiffness matrix
235 Kfvec = zeros(2*(ny+1)*(nx+1),1); % build zeros support vector
236 for i = 1:length(supp) % for each support element...
237     dofsup = dofmat(supp(i),:); %...find the corresponding dof
238     for j = 1:length(dofsup) % calculate new stiffness vector
239         Kfvec(dofsup(j)) = Kfvec(dofsup(j))+(zF(supp(i))^q)*k0;
240     end
241 end
242 Kf = spdiags(Kfvec,0,2*(ny+1)*(nx+1),2*(ny+1)*(nx+1)); % create
    diagonal Kf
243 Kt = K+Kf; % update total force
244 U(free,:) = Kt(free,free)\Ftot(free,:); % displacement solving
245 c(iter) = 0; % set compliance to zero
246 comp(iter) = 0;%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%TEMP%%
247 Sens = 0; % set sensitivity to zero
248 Sensz = 0; % set constraint sensitivity to zero
249 %% Calculate compliance and sensitivity
250 for i = 1:size(Fn,2) % for number of load cases
251     Ui = U(:,i); % displacement per load case
252     c0 = reshape(sum((Ui(dofmat)*Ke).*Ui(dofmat),2),ny,nx); % initial
        compliance
253     cz0 = reshape(sum((Ui(dofmat)*k1).*Ui(dofmat),2),ny,nx); %
        initial support compliance
254     c(iter) = c(iter) + sum(sum((Emin+xF.^p*(E-Emin)).*c0)) + sum(sum
        ((zF.^q).*cz0)); % calculate compliance
255     comp(iter) = comp(iter) + sum(sum((Emin+xF.^p*(E-Emin)).*c0));%
        %%%TEMP%%
256     Sens = Sens + reshape(2*Ui(dofmat)*repmat([0;-9.81*rho/4],4,1),ny
        ,nx) -p*(E-Emin)*xF.^(p-1).*c0; % sensitivity
257     Sensz = Sensz + -q*zF.^(q-1).*cz0; % calculate sensitivity to
        support variable
258 end
259 Senc = ones(ny,nx); % set constraint sensitivity
260 if dist == 0
261     Sencz = ones(ny,nx);
262 elseif dist == 1
263     Sencz =ones(ny,nx)*Scost; % set weighted cost constraint
        sensitivity
264 elseif dist == 2
265     Sencz = Scost*ones(ny,nx); % set weighted cost constraint
        sensitivity
266     %Sencz = ones(ny,nx); % set weighted cost constraint sensitivity
267 end
268 if fil == 0 % optimality criterion with sensitivity filter
269     Sens(:) = H*(x(:).*Sens(:))./Hs./max(1e-3,x(:)); % update
        filtered sensitivity
270 elseif fil == 1 % optimality criterion with density filter
271     Sens(:) = H*(Sens(:))./Hs; % update filtered sensitivity
272     Senc(:) = H*(Senc(:))./Hs; % update filtered sensitivity of
        constraint
273 elseif fil == 2 % optimality criterion with heaviside filter
274     dx = beta*exp(-beta*xTilde)+exp(-beta); % update hs parameter

```

```

275     Sens(:) = H*(Sens(:).*dx(:)./Hs); % update filtered sensitivity
276     Senc(:) = H*(Senc(:).*dx(:)./Hs); % update filtered sensitivity
           of constraint
277     end
278     %% Update design variables Optimality Criterion
279     if sol == 0           % use optimality criterion method
280         l1 = 0;           % initial lower bound for lagranian mulitplier
281         l2 = 1e9;        % initial upper bound for lagranian multiplier
282         while (l2-l1)/(l1+l2) > 1e-3; % start loop
283             lag = 0.5*(l1+l2); % average of lagranian interval
284             xnew = max(0,max(x-move,min(1,min(x+move,x.*sqrt(-Sens./Senc/
                lag))))); % update element densities
285             if fil == 0 % sensitivity filter
286                 xF = xnew; % updated result
287             elseif fil == 1 % density filter
288                 xF(:) = (H*xnew(:))./Hs; % updated filtered density
                result
289             elseif fil == 2 % heaviside filter
290                 xTilde(:) = (H*xnew(:))./Hs; % set filtered density
291                 xF(:) = 1-exp(-beta*xTilde)+xTilde*exp(-beta); % updated
                result
292             end
293             if shap == 1 % restriction is on
294                 xF(rest==1) = area; % set restricted area
295             end
296             if sum(xF(:)) > vol*nx*ny; % check for optimum
297                 l1 = lag; % update lower bound to average
298             else
299                 l2 = lag; % update upper bound to average
300             end
301         end
302         %% Method of moving asymptotes
303     elseif sol == 1      % use mma solver
304         xval = [xFree(:);zval(:)]; % store current design variable for
                mma
305         if iter == 1     % for the first iteration...
306             cscale = 1/c(iter); % ...set scaling factor for mma solver
307             cscale = 5.0131e-6;
308         end
309         f0 = c(iter)*cscale; % objective at current design variable for
                mma
310         df0dx = [Sens(efree)*cscale; Senz(supp)']*cscale; % store
                sensitivity for mma
311         if dist == 0     % no cost distribution
312             Scosts = zF; % cost-funcion no influence
313         elseif dist == 1 % x-axis cost distribution
314             Scosts = zF*Scost; % update weighted constraint function
315         elseif dist == 2 % y-axis cost distribution
316             Scosts = Scost*zF; % update weighted constraint function
317         end
318         f = [(sum(xF(:))/(vol*nx*ny)-1);(sum(Scosts(supp))/(zvol*size(
                supp,2))-1)]; % normalized constraint function

```

```

319     dfdx = [Senc(efree)']/(vol*ny*nx) zzer'; xzer' Sencz(supp)/(zvol*
        size(supp,2)); % derivative of the constraint function
320     [xmma,~,~,~,~,~,~,~,~,low,upp] = ...
321         mmasub(m,n,iter,xval,xmin,xmax,xold1,xold2, ...
322             f0,df0dx,df0dx2,f,dfdx,dfdx2,low,upp,a0,a,comma,d,subs); % mma
        solver
323     xold2 = xold1; % used by mma to monitor convergence
324     xold1 = [xFree(:);zval(:)]; % previous x, to monitor convergence
325     xnew = xF; % update result
326     xnew(efree) = xmma(1:xsiz); % include restricted elements
327     znew = zF; % update design result
328     znew(supp) = xmma(xsiz+1:end); % include mma solved supports
329     xnew = reshape(xnew,ny,nx); % reshape xmma vector to original
        size
330     znew = reshape(znew,ny,nx); % reshape support vector to original
        size
331     if fil == 0 % sensitivity filter
332         xF = xnew; % update design variables
333     elseif fil == 1 % density filter
334         xF(:) = (H*xnew(:))./Hs; % update filtered densities result
335     elseif fil == 2 % heaviside filter
336         xTilde(:) = (H*xnew(:))./Hs; % filtered result
337         xF(:) = 1 - exp(-beta*xTilde) + xTilde*exp(-beta); % update design
        variable
338     end
339     if shap == 1 || shap == 2 % if restrictions enableed
340         xF(rest==1) = area; % set restricted area
341     end
342     zF(:) = znew(:); % update support variables
343     zval = znew(supp); % update support variables
344     end
345     xFree = xnew(efree); % set non-restricted area
346     diff = max(abs(xnew(:)-x(:))); % difference of maximum element change
347     x = xnew; % update design variable
348     z = znew; % update support design variable
349     if fil == 2 && beta < 512 && pen == p(end) && (loopbeta >= 50 || diff
        <= tol) % hs filter
350         beta = 2*beta; % increase beta-factor
351         fprintf('beta now is %3.0f\n',beta) % display increase of b-
        factor
352         loopbeta = 0; % set hs filter loop to zero
353         diff = 1; % set convergence to initial value
354     end
355     %% Store results into database X
356     X(:, :, iter) = xF; % each element value x is stored for each
        iteration
357     C(iter) = c(iter); % each compliance is stored for each iteration
358     Z(:, :, iter) = zF; % each support variable is stored for each
        iteration
359     assignin('base', 'X', X); % each iteration (3rd dimension)
360     assignin('base', 'C', C); % each iteration (3rd dimension)
361     assignin('base', 'Z', Z); % each iteration (3rd dimension).
362     %% Results

```

```

363     if dis == 1           % display iterations
364         disp([' Iter:' sprintf('%4i',iter) ' Obj:' sprintf('%10.4f',c(
            iter)) ...
365             ' Vol:' sprintf('%6.3f',mean(xF(:))) ' Diff:' sprintf('%6.3f',
            ,diff) ' ZVol:' sprintf('%6.3f',mean(Scosts(supp)))]);
366     elseif dis == 2     % display parts of iterations
367         if iter == 1 || iter == disiter
368             if iter == 1
369                 disiter = plotiter;
370             elseif iter == disiter
371                 disiter = disiter + plotiter;
372             end
373         disp([' Iter:' sprintf('%4i',iter) ' Obj:' sprintf('%10.4f',c
            (iter)) ...
374             ' Vol:' sprintf('%6.3f',mean(xF(:))) ' Diff:' sprintf('
            %6.3f',diff) ' ZVol:' sprintf('%6.3f',mean(Scosts(supp)
            ))]);
375     end
376 end
377 if draw == 1           % plot iterations
378     figure(1)
379     subplot(2,1,1)
380     colormap(gray); imagesc(1-xF);
381     set(gca, 'XTick', [], 'YTick', [], 'XTicklabel', [], ...
382         'YTicklabel', [], 'xcolor', '[0.7 0.7 0.7]', 'ycolor', '[0.7 0.7
            0.7]');
383     xlabel(sprintf('c = %.2f',c(iter)), 'Color', 'k')
384     axis equal; axis tight
385     drawnow;
386     hold on
387     if iter == 1
388         % Plot coloured dots for force application
389         for i = 1:length(Fe)
390             npfx(i) = ceil(Fe(i)/(2*(ny+1))) - 0.5;
391             nplot = ceil(Fe(i)/2);
392             while nplot > (ny+1)
393                 nplot = nplot - (ny+1);
394             end
395             npfy(i) = nplot - 0.5;
396         end
397         plot(npfx, npfy, 'g.', 'MarkerSize', 20)
398         % Plot coloured dots for constraints
399         for i = 1:length(fix)
400             npx(i) = ceil(fix(i)/(2*(ny+1))) - 0.5;
401             nplot = ceil(fix(i)/2);
402             while nplot > (ny+1)
403                 nplot = nplot - (ny+1);
404             end
405             npy(i) = nplot - 0.5;
406         end
407         plot(npx, npy, 'r.', 'MarkerSize', 20)
408     end
409     % Plot coloured dots for design of supports

```

```

410     for i = 1:nx*ny
411         if zF(i) > zplot % treshold for plotting supports
412             if ceil(i/ny) == nx
413                 npdx(i) = ceil(i/ny) + 0.5;
414             elseif ceil(i/ny) == 1
415                 npdx(i) = ceil(i/ny) - 0.5;
416             else
417                 npdx(i) = ceil(i/ny);
418             end
419             nplot = i;
420             while nplot > ny
421                 nplot = nplot-ny;
422             end
423             if nplot == ny
424                 npdy(i) = nplot+0.5;
425             elseif nplot == 1
426                 npdy(i) = nplot-0.5;
427             else
428                 npdy(i) = nplot;
429             end
430         end
431     end
432     if iter > 1
433         delete(Dos)
434     end
435     if exist('npdx') %#ok<EXIST>
436         Dos = plot(nonzeros(npdx),nonzeros(npdy),'b.','MarkerSize',
437                 ,20);
438         clear npdx; clear npdy;
439         uistack(Dos,'bottom')
440     end
441     % Plot compliance plot
442     figure(1)
443     subplot(2,1,2)
444     plot(c(1:iter))
445     set(gca,'YTick',[],'YTicklabel',[])
446     xlabel('Iterations')
447     ylabel('Compliance')
448     xaxmax = c(iter);
449     yaxmax = max(c);
450     yaxmin = min(c(1:iter));
451     if pcon == 0
452         yaxmax = mean([yaxmin yaxmax]);
453     end
454     ylim([0.95*yaxmin yaxmax])
455     xlim([1 min(iter+10,miter)])
456     elseif draw == 2 % plot parts of iterations
457         if iter == 1 || iter == drawiter
458             if iter == 1
459                 drawiter = plotiter;
460             elseif iter == drawiter
461                 drawiter = drawiter + plotiter;
462             end
463         end
464     end

```

```

462     figure(1)
463     subplot(2,1,1)
464     colormap(gray); imagesc(1-xF);
465     set(gca, 'XTick', [], 'YTick', [], 'XTicklabel', [], ...
466           'YTicklabel', [], 'xcolor', '[0.7 0.7 0.7]', 'ycolor', '[0.7
           0.7 0.7]');
467     xlabel(sprintf('c = %.2f', c(iter)), 'Color', 'k')
468     axis equal; axis tight
469     drawnow;
470     hold on
471     if iter == 1
472         % Plot coloured dots for force application
473         for i = 1:length(Fe)
474             npfx(i) = ceil(Fe(i)/(2*(ny+1))) - 0.5;
475             nplot = ceil(Fe(i)/2);
476             while nplot > (ny+1)
477                 nplot = nplot - (ny+1);
478             end
479             npfy(i) = nplot - 0.5;
480         end
481         plot(npfx, npfy, 'g.', 'MarkerSize', 20)
482         % Plot coloured dots for constraints
483         for i = 1:length(fix)
484             npx(i) = ceil(fix(i)/(2*(ny+1))) - 0.5;
485             nplot = ceil(fix(i)/2);
486             while nplot > (ny+1)
487                 nplot = nplot - (ny+1);
488             end
489             npy(i) = nplot - 0.5;
490         end
491         plot(npx, npy, 'r.', 'MarkerSize', 20)
492     end
493     % Plot coloured dots for design of supports
494     for i = 1:nx*ny
495         if zF(i) > zplot % treshold for plotting supports
496             if ceil(i/ny) == nx
497                 npdx(i) = ceil(i/ny) + 0.5;
498             elseif ceil(i/ny) == 1
499                 npdx(i) = ceil(i/ny) - 0.5;
500             else
501                 npdx(i) = ceil(i/ny);
502             end
503             nplot = i;
504             while nplot > ny
505                 nplot = nplot - ny;
506             end
507             if nplot == ny
508                 npdy(i) = nplot + 0.5;
509             elseif nplot == 1
510                 npdy(i) = nplot - 0.5;
511             else
512                 npdy(i) = nplot;
513             end

```

```

514         end
515     end
516     if iter > 1
517         delete(Dos)
518     end
519     if exist('npdx') %#ok<EXIST>
520         Dos = plot(nonzeros(npdx),nonzeros(npdy),'b.','MarkerSize
521                 ',20);
522         clear npdx; clear npdy;
523         uistack(Dos,'bottom')
524     end
525     % Plot compliance plot
526     figure(1)
527     subplot(2,1,2)
528     plot(c(1:iter))
529     set(gca,'YTick',[],'YTicklabel',[])
530     xlabel('Iterations')
531     ylabel('Compliance')
532     xaxmax = c(iter);
533     yaxmax = max(c);
534     yaxmin = min(c(1:iter));
535     if pcon == 0
536         yaxmax = mean([yaxmin yaxmax]);
537     end
538     ylim([0.95*yaxmin yaxmax])
539     xlim([1 min(iter+10,miter)])
540 end
541 end
542 %% ONLY DISPLAY FINAL RESULT
543 if dis == 0 || dis == 2 % display final result
544     disp([' Iter:' sprintf('%4i',iter) ' Obj:' sprintf('%10.4f',c(iter))
545         ...
546         ' Vol:' sprintf('%6.3f',mean(xF(:))) ' Diff:' sprintf('%6.3f',
547         diff) ' ZVol:' sprintf('%6.3f',mean(Scosts(supp))]);
548 end
549 if draw == 0 || draw == 2 % plot final result
550     figure(1)
551     subplot(2,1,1)
552     colormap(gray); imagesc(1-xF);
553     axis equal; axis tight;
554     set(gca,'XTick',[],'YTick',[],'XTicklabel',[],...
555         'YTicklabel',[],'xcolor','[0.7 0.7 0.7]','ycolor','[0.7 0.7 0.7]')
556     xlabel(sprintf('c = %.2f',c(iter)),'Color','k')
557     drawnow;
558     hold on
559     % Plot coloured dots for force application
560     for i = 1:length(Fe)
561         npfx(i) = ceil(Fe(i)/(2*(ny+1)))-0.5;
562         nplot = ceil(Fe(i)/2);
563         while nplot > (ny+1)
564             nplot = nplot-(ny+1);

```

```

563         end
564         npfy(i) = nplot - 0.5;
565     end
566     For = plot(npfx, npfy, 'g.', 'MarkerSize', 20);
567     uistack(For, 'bottom')
568     % Plot coloured dots for constraints
569     for i = 1:length(fix)
570         npx(i) = ceil(fix(i)/(2*(ny+1))) - 0.5;
571         nplot = ceil(fix(i)/2);
572         while nplot > (ny+1)
573             nplot = nplot - (ny+1);
574         end
575         npy(i) = nplot - 0.5;
576     end
577     plot(npx, npy, 'r.', 'MarkerSize', 20)
578     % Plot coloured dots for design of supports
579     for i = 1:nx*ny
580         if zF(i) > zplot % treshold for plotting supports
581             if ceil(i/ny) == nx
582                 npdx(i) = ceil(i/ny) + 0.5;
583             elseif ceil(i/ny) == 1
584                 npdx(i) = ceil(i/ny) - 0.5;
585             else
586                 npdx(i) = ceil(i/ny);
587             end
588             nplot = i;
589             while nplot > ny
590                 nplot = nplot - ny;
591             end
592             if nplot == ny
593                 npdy(i) = nplot + 0.5;
594             elseif nplot == 1
595                 npdy(i) = nplot - 0.5;
596             else
597                 npdy(i) = nplot;
598             end
599         end
600     end
601     if exist('Dos(1)') %#ok<EXIST>
602         delete(Dos(1))
603     end
604     if exist('npdx') %#ok<EXIST>
605         Dos = plot(nonzeros(npdx), nonzeros(npdy), 'b.', 'MarkerSize', 20);
606         clear npdx; clear npdy;
607         uistack(Dos, 'bottom')
608     end
609     % Plot compliance plot
610     if adv == 0
611         figure(1)
612         subplot(2,1,2)
613         plot(c(1:iter))
614         set(gca, 'YTick', [], 'YTicklabel', [])
615         xlabel('Iterations')

```

```

616     ylabel('Compliance')
617     xaxmax = c(iter);
618     yaxmax = max(c);
619     yaxmin = min(c(1:iter));
620     if pcon == 0
621         yaxmax = mean([yaxmin yaxmax]);
622     end
623     ylim([0.95*yaxmin yaxmax])
624     xlim([1 min(iter+10,miter)])
625 end
626 end
627 %% PLOTTING DISPLACEMENT (COMPLIANT MECHANISMS)
628 if def == 1
629     figure(1)
630     subplot(2,1,1)
631     xaxis = get(gca,'XLim');
632     yaxis = get(gca,'YLim');
633     figure(3)
634     clear mov
635     colormap(gray);
636     Umov = 1;           % start movie counter
637     Umax = -0.005;     % define maximum displacement
638     for Udisp = linspace(0,Umax,10); % vary input displacement
639         clf
640         for ely = 1:ny % plot displacements...
641             for elx = 1:nx % for each element...
642                 if xF(ely,elx) > 0 % exclude white regions for plotting
643                     purposes
644                         n1 = (ny+1)*(elx-1)+ely;
645                         n2 = (ny+1)* elx +ely;
646                         Ue = Udisp*U([2*n1-1;2*n1; 2*n2-1;2*n2; 2*n2+1;2*n2
647                             +2; 2*n1+1;2*n1+2],1);
648                         ly = ely-1; lx = elx-1;
649                         xx = [Ue(1,1)+lx Ue(3,1)+lx+1 Ue(5,1)+lx+1 Ue(7,1)+lx
650                             ]';
651                         yy = [-Ue(2,1)-ly -Ue(4,1)-ly -Ue(6,1)-ly-1 -Ue(8,1)-
652                             ly-1]';
653                         subplot(2,1,1)
654                         patch([xx xx],[yy yy],[-xF(ely,elx) -xF(ely,elx)], '
655                            LineStyle','none');
656
657                 end
658             end
659         end
660         xlim(xaxis)
661         ylim([-yaxis(2) yaxis(1)])
662         axis equal; axis tight;
663         set(gca,'xcolor','[0.7 0.7 0.7]','ycolor','[0.7 0.7 0.7]')
664         drawnow
665         mov(Umov) = getframe(3); % movie
666         Umov = Umov +1; % update counter
667     end
668     movlip = flip(mov); % create symmetry

```

```
664     movull = [mov movlip]; % create symmetry
665     FileName = ['Compliant_',datestr(now, 'ddmm_HHMMSS'),'.avi']; %
        dynamic filename
666     movie2avi(movull, FileName, 'compression', 'None', 'FPS', 10); % save
        video
667 end
668 toc           % stop timer
669     max(K(:))
670     max(Kf(:))
```

B.8 ADVANCED DOS.m

By the inspiration of the ADVANCED (B.2) and the Design of Supports plug-in (C.9) a complete advanced and enhanced code is made. This code includes displaying support design and can be used to easily vary in cost distribution functions, in order to produce the figures as depicted in 4.3. The changes are quite big, so it's recommended to just run this new file, instead of writing an add-in code.

```

1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %                                                                                                                                                                %
3  % Topology Optimization Using Matlab                                                                                   %
4  % ADVANCED_DOS.m                                                                                                     %
5  %                                                                                                                                                                %
6  % Delft University of Technology, Department PME                                                                     %
7  % Master of Science Thesis Project                                                                                   %
8  %                                                                                                                                                                %
9  % Stefan Broxterman                                                                                                 %
10 %                                                                                                                                                                %
11 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
12 clc; clf; close all; clear X;
13 %% DEFINE OPTIMIZATION VARIABLES
14 var = 5; % [1 = mesh, 2 = penalty, 3 = filter radius, 4 =
    volume fraction, 5 = support cost, 6 = evolution]
15 nxvec = [30,60,80,120]; % horizontal elements vector
16 nyvec = [10,20,40,40]; % vertical elements vector
17 volvec = [0.2 0.35 0.5 0.65]; % volume fraction vector
18 rminvec = [1,1.25,1.5,3]; % filter size vector
19 penvec = [1, 2, 3, 5]; % penalty vector
20 filvec = [0, 1, 2]; % filter vector
21 costvec = [1, 5, 10, 50]; % cost vector
22 evolvec = [0.05, 0.25, 0.5, 1]; % evolution fraction vector
23 %% SET DEFAULT VALUES
24 nx = nxvec(3); % default number of horizontal elements
25 ny = nyvec(3); % default number of vertical elements
26 vol = volvec(1); % default number of volume fraction
27 pen = penvec(3); % default penalty
28 rmin = rminvec(3); % default filter radius
29 fil= filvec(2); % default filter method
30 cost = costvec; % default cost distribution
31 %% SET OPTIMIZATION VALUES
32 ex = [30,60,90,120]; % vector size for pre-allocating space
33 figend = 4; % set total of varying values
34 label = ['a','b','c','d','e']; % graphic label
35 %% PRE-ALLOCATE SPACE
36 loops = zeros(1,size(ex,2)); % initial loops matrix
37 obj = zeros(1,size(ex,2)); % initial objective matrix
38 t = zeros(1,size(ex,2)); % initial time matrix
39 Y = zeros(size(ex,2),5); % initial results matrix
40 if var == 6 % for evolution scheme, BasicK.m only needs to
    ...
41 BRIDGE % run one time only

```

```

42 end
43 %% START LOOP
44 for fig = 1:figend % start iteration loop
45     tic; % start timer
46     if var ~= 6 % for non-evolution scheme, run below
47         clear X; clear C; % clear results matrix for each run
48         if var == 1 % differentiation on number of elements
49             nx = nxvec(fig); % pick each horizontal value
50             ny = nyvec(fig); % pick each vertical value
51         elseif var == 2 % differentiation on penalty
52             pen = penvec(fig); % pick each penalty
53         elseif var == 3 % differentiation on filter radius
54             rmin = rminvec(fig); % pick each rmin
55         elseif var == 4 % differentiation on filter method
56             vol = volvec(fig); % pick each filter method
57         elseif var == 5 % differentiation on support cost
58             cost = costvec(fig); % pick each support cost
59         end
60         figure(1)
61         clf
62         BRIDGE % run Basic.m
63         loops(fig) = size(X,3); % number of iterations used
64         obj(fig) = c(iter); % store objective function
65         prog = X(:, :, loops(fig)); % store densities for progression
66         drawing
67     elseif var == 6 % store compliance for evolution vector
68         loops = size(X,3); % for evolutionary scheme, calculate rounded
69         ...
70         loop(1) = round(evolvec(1)*loops); % values of loops and store
71         ...
72         loop(2) = round(evolvec(2)*loops); % this loop number
73         loop(3) = round(evolvec(3)*loops);
74         loop(4) = round(evolvec(4)*loops);
75         prog = X(:, :, loop); % progression picture for each evolution
76         fraction
77     end
78 %% Set graphics
79 if draw == 1 % check for drawing
80     H = get(gcf, 'Position'); % get position of figure
81 else
82     H = [680,558,560,420]; % set size of figure(2) plot windows
83 end
84 H2 = figure(2); % plot window for progression pictures
85 set(H2, 'position', [H(1)+H(3) H(2) H(3) H(4)]); % place figure(2) next
86 to (1)
87 %% Draw progression plots
88 subplot(3,2,fig+2) % plot each differentiation
89 colormap(gray); % grayscale
90 if var == 6 % evolution needs different plotting
91     imagesc(1-prog(:, :, fig)); % plot progression picture
92     xlabel(sprintf('c = %.2f', C(loop(fig))), 'color', 'k')
93 else
94     imagesc(1-prog); % plot progression picture

```

```

90     xlabel(sprintf('c = %.2f',obj(fig)),'color','k')
91 end
92 set(gca,'XTick',[],'YTick',[],'XTicklabel',[],...
93     'YTicklabel',[],'xcolor','[0.7 0.7 0.7]','ycolor','[0.7 0.7 0.7]'
94 )
95 axis equal; axis tight; % set additional options
96 if var == 6 % evolution needs different plotting
97     xlabel(sprintf('c = %.2f',C(loop(fig))),'color','k')
98 else
99     xlabel(sprintf('c = %.2f',obj(fig)),'color','k')
100 end
101 ylabel(sprintf('%s      ',(label(fig+1))),...
102     'rot',0,'color','k','FontSize',11)
103 hold on
104 % Plot coloured dots for design of supports
105 for i = 1:nx*ny
106     if zF(i) > zplot % treshold for plotting supports
107         if ceil(i/ny) == nx
108             npdx(i) = ceil(i/ny) + 0.5;
109         elseif ceil(i/ny) == 1
110             npdx(i) = ceil(i/ny) - 0.5;
111         else
112             npdx(i) = ceil(i/ny);
113         end
114         nplot = i;
115         while nplot > ny
116             nplot = nplot-ny;
117         end
118         if nplot == ny
119             npdy(i) = nplot+0.5;
120         elseif nplot == 1
121             npdy(i) = nplot-0.5;
122         else
123             npdy(i) = nplot;
124         end
125     end
126 end
127 if exist('npdx') %#ok<EXIST>
128     Dos = plot(nonzeros(npdx),nonzeros(npdy),'b.','MarkerSize',20);
129     clear npdx; clear npdy;
130     uistack(Dos,'bottom')
131 end
132 %% Store compliance
133 if var ~= 6 % store compliance for further plotting
134     if fig == 1
135         C1 = C;
136     elseif fig == 2
137         C2 = C;
138     elseif fig == 3
139         C3 = C;
140     elseif fig == 4
141         C4 = C;
142     end

```

```

142     end
143     %% Draw graphics
144     xbox = get(gca, 'XLim');
145     ybox = get(gca, 'YLim');
146     xwidth = xbox(2)-xbox(1);
147     ywidth = ybox(2)-ybox(1);
148     rectangle('Position',[xbox(1),ybox(1),xwidth,ywidth],...
149             'EdgeColor',[0.5 0.5 0.5],'LineStyle',':'); drawnow;
150     t(fig) = toc;
151
152     %% Output
153     if var ~= 6 % output results for non-evolutionary schemes
154         Y(fig,:) = [fig ex(fig) loops(fig) obj(fig) t(fig)];
155         if fig == figend
156             Y
157         end;
158     end
159     %% Compliance graphs
160     if var ~= 6
161         H3 = figure(3);
162         set(H3, 'position', [H(1)-H(3) H(2) H(3) H(4)]); % place figure(2)
163             next to (1)
164         hold on
165         switch fig
166             case 1 % first variable
167                 plot(1:length(C1),C1, 'b:', 'LineWidth', 2)
168                 xaxmax = mean(length(C1));
169                 yaxmax = max(max(C1));
170                 yaxmin = min(C1);
171                 if var == 1
172                     legend(sprintf('mesh = %g x %g ',nxvec(1),nyvec(1)))
173                 elseif var == 2
174                     legend(sprintf('pen = %g',penvec(1)))
175                 elseif var == 3
176                     legend(sprintf('Rmin = %g',rminvec(1)))
177                 elseif var == 4
178                     legend(sprintf('vol = %g',volvec(1)))
179                 elseif var == 5
180                     legend(sprintf('cost = %g',costvec(1)))
181                 end
182             case 2 % second variable
183                 plot(1:length(C2),C2, 'r--', 'LineWidth', 2)
184                 xaxmax = mean([length(C1) length(C2)]);
185                 yaxmax = max([max(C1) max(C2)]);
186                 yaxmin = min(min([C1 C2]));
187                 if var == 1
188                     legend(sprintf('mesh = %g x %g',nxvec(1),nyvec(2)),
189                             sprintf('mesh = %g x %g',nxvec(2),nyvec(2)))
190                 elseif var == 2
191                     legend(sprintf('pen = %g ',penvec(1)),sprintf('pen =
192                             %g',penvec(2)))
193                 elseif var == 3

```

```

191         legend(sprintf('Rmin = %g ',rminvec(1)),sprintf('Rmin
           = %g',rminvec(2)))
192     elseif var == 4
193         legend(sprintf('vol = %g ',volvec(1)),sprintf('vol =
           %g',volvec(2)))
194     elseif var == 5
195         legend(sprintf('cost = %g ',costvec(1)),sprintf('cost
           = %g',costvec(2)))
196     end
197 case 3     % third variable
198     plot(1:length(C3),C3,'k','LineWidth',2)
199     xaxmax = mean([length(C1) length(C2) length(C3)]);
200     yaxmax = max([max(C1) max(C2) max(C3)]);
201     yaxmin = min(min([C1 C2 C3]));
202     if var == 1
203         legend(sprintf('mesh = %g x %g',nxvec(1),nyvec(2)),
           sprintf('mesh = %g x %g',nxvec(2),nyvec(2)),
           sprintf('mesh = %g x %g',nxvec(3),nyvec(3)))
204     elseif var == 2
205         legend(sprintf('pen = %g ',penvec(1)),sprintf('pen =
           %g',penvec(2)),sprintf('pen = %g',penvec(3)))
206     elseif var == 3
207         legend(sprintf('Rmin = %g ',rminvec(1)),sprintf('Rmin
           = %g',rminvec(2)),sprintf('Rmin = %g',rminvec(3))
           )
208     elseif var == 4
209         legend(sprintf('vol = %g ',volvec(1)),sprintf('vol =
           %g',volvec(2)),sprintf('vol = %g',volvec(3)))
210     elseif var == 5
211         legend(sprintf('cost = %g ',costvec(1)),sprintf('cost
           = %g',costvec(2)),sprintf('cost = %g',costvec(3))
           )
212     end
213 case 4     % fourth variable
214     plot(1:length(C4),C4,'g-.','LineWidth',2)
215     xaxmax = mean([length(C1) length(C2) length(C3) length(C4)
           ]]);
216     yaxmax = max([max(C1) max(C2) max(C3) max(C4)]);
217     yaxmin = min(min([C1 C2 C3 C4]));
218     if var == 1
219         legend(sprintf('mesh = %g x %g',nxvec(1),nyvec(2)),
           sprintf('mesh = %g x %g',nxvec(2),nyvec(2)),
           sprintf('mesh = %g x %g',nxvec(3),nyvec(3)),
           sprintf('mesh = %g x %g',nxvec(4),nyvec(4)))
220     elseif var == 2
221         legend(sprintf('pen = %g ',penvec(1)),sprintf('pen =
           %g',penvec(2)),sprintf('pen = %g',penvec(3)),
           sprintf('pen = %g',penvec(4)))
222     elseif var == 3
223         legend(sprintf('Rmin = %g ',rminvec(1)),sprintf('Rmin
           = %g',rminvec(2)),sprintf('Rmin = %g',rminvec(3))
           ,sprintf('Rmin = %g',rminvec(4)))
224     elseif var == 4

```

```

225         legend(sprintf('vol = %g ',volvec(1)),sprintf('vol =
                %g',volvec(2)),sprintf('vol = %g',volvec(3)),
                sprintf('vol = %g',volvec(4)))
226     elseif var == 5
227         legend(sprintf('cost = %g ',costvec(1)),sprintf('cost
                = %g',costvec(2)),sprintf('cost = %g',costvec(3))
                ,sprintf('cost = %g',costvec(4)))
228     end
229 end
230 xlabel('Number of iterations')
231 ylabel('Compliance')
232 if exist('pcon','var') == 0
233     yaxmax = mean([yaxmin yaxmax]);
234 elseif pcon == 0
235     yaxmax = mean([yaxmin yaxmax]);
236 end
237 axis([0 xaxmax 0.95*yaxmin yaxmax])
238 elseif var == 6
239     H3 = figure(3);
240     set(H3, 'position', [H(1)-H(3) H(2) H(3) H(4)]); % place figure(2)
                next to (1)
241     hold on
242     plot(C)
243     xlabel('Number of iterations')
244     ylabel('Compliance')
245     axis([0 length(C) 0.9*min(C) max(C)])
246 end
247 end
248 %% STORE RESULTS
249 disp('Y = i, penalty, loops, objective, time')
250 if var == 1 % mesh refinement
251     Ymesh = Y; % store result matrix
252     save('MeshRefinementY.mat','Y');
253 elseif var == 2 % penalty
254     Ypenal = Y; % store result matrix
255     save('PenaltyY.mat','Y');
256 elseif var == 3 % filter radius
257     Yfilter = Y; % store result matrix
258     save('FilterY.mat','Y');
259 elseif var == 4 % filter radius
260     Yvolume = Y; % store result matrix
261     save('VolumeY.mat','Y');
262 end
263 %% DRAW DESIGN PROBLEM
264 figure(2)
265 subplot(3,2,(1:2)) % plot the initial mechanical problem
266 rectangle('Position',[xbox(1),ybox(1),xwidth,ywidth],...
267 'FaceColor',[0.5 0.5 0.5])
268 axis equal; axis tight;
269 set(gca, 'XTick', [], 'YTick', [], 'XTicklabel', [], ...
270 'YTicklabel', [], 'xcolor', 'w', 'ycolor', 'w')
271 ylabel(sprintf('%s', (label(1))), 'rot', 0, 'color', 'k', 'FontSize', 11)
272 draw_arrow([xbox(2) ybox(1)], [xbox(2) -0.25*ywidth], 1)

```

```
273 rectangle('Position',[-0.1*xwidth,ybox(1)-0.1*ywidth,...
274           0.1*xwidth,1.2*ywidth],'FaceColor',[0 0 0],'LineWidth',3)
```

B.9 Design of Actuator Placement.m

In this section, the complete code of designing optimal actuator placement is available. Here, topology is not yet involved and remains fixed. By running this code, the produced picture in Figure 5-2 can be made immediately.

```

1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %                                                                                                                                                                %
3  % Topology Optimization Using Matlab                                                                                                                                 %
4  % Design of Actuator Placement                                                                                                                                      %
5  %                                                                                                                                                                %
6  % Delft University of Technology, Department PME                                                                                                                                                            %
7  % Master of Science Thesis Project                                                                                                                                   %
8  %                                                                                                                                                                %
9  % Stefan Broxterman                                                                                                                                                 %
10 %                                                                                                                                                                %
11 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
12 %
13 tic                                                                                                                                                                % start timer
14 %% DEFINE PARAMETERS
15 adv = 0;                                                                                                                                                            % use advanced function [0 = off, 1 = on]
16 if adv == 0                                                                                                                                                          % define parameters at behalf of the advanced
    function
17     nx = 90;                                                                                                                                                          % number of elements horizontal
18     ny = 30;                                                                                                                                                          % number of elements vertical
19     vol = 1;                                                                                                                                                          % volume fraction [0-1]
20     pen = 3;                                                                                                                                                          % penalty
21     rmin = 1.5;                                                                                                                                                       % filter size
22     fil = 1;                                                                                                                                                          % filter method [0 = sensitivity filtering, 1 =
        density filtering, 2 = heaviside filtering]
23     clc; clf; close all; clear X; clear W; % clear workspace
24 end
25 %% DEFINE SOLUTION METHOD
26 sol = 1;                                                                                                                                                            % solution method [0 = oc(sens), 1 = mma]
27 pcon = 1;                                                                                                                                                            % use continuation method [0 = off, 1 = on]
28 fincheck = 1;                                                                                                                                                       % finite difference check [0 = off, 1 = on, 2 =
    break]
29 %% DEFINE CALCULATION
30 tol = 0.001;                                                                                                                                                        % tolerance for convergence criterion [0.01]
31 move = 0.2;                                                                                                                                                          % move limit for lagrange [0.2]
32 pcinc = 1.03;                                                                                                                                                       % penalty continuation increasing factor [1.03]
33 piter = 20;                                                                                                                                                          % number of iteration for starting penalty [20]
34 miter = 1000;                                                                                                                                                       % maximum number of iterations [1000]
35 plotiter = 5;                                                                                                                                                       % gap of iterations used to plot or draw
    iterations [5]
36 def = 0;                                                                                                                                                            % plot deformations [0 = off, 1 = on, 2 = play
    video]
37 wplot = 0.20;                                                                                                                                                       % define treshold factor of Fmax for force plot
    [0.20]
38 h = 1e-6;                                                                                                                                                            % perturbation value for finite difference method
    [1e-6]
39 %% DEFINE OUTPUT

```

```

40 draw = 0; % plot iterations [0 = off, 1 = on, 2 = partial]
41 dis = 0; % display iterations [0 = off, 1 = on, 2 =
    partial]
42 %% DEFINE MATERIAL
43 E = 1; % young's modulus of solid [1]
44 Emin = 1e-9; % young's modulus of void [1e-9]
45 nu = 0.3; % poisson ratio [0.3]
46 rho = 0e-3; % density [0e-3]
47 g = 9.81; % gravitational acceleration [9.81]
48 %% DEFINE FORCE
49 Fe = 2*(ny+1)+45*2*(ny+1):2*(ny+1):2*(ny+1)*(nx+1); % element of force
    application [2*(ny+1)+45*2*(ny+1):2*(ny+1):2*(ny+1)*(nx+1)]
50 Fn = 1; % number of applied force locations [1]
51 Fv = -1/length(Fe); % value of applied force [-1]
52 %% DEFINE SUPPORTS
53 fix = 1:2*(ny+1); % fixed degrees of freedom [1:2*(ny+1)]
54 %% DEFINE DESIGN OF ACTUATOR
55 Fmaxnode = 1; % define max force per node [1]
56 Fmin = -1; % minimal force constraint [1]
57 sen = 5; % penalty for actuator design [5]
58 if abs(Fmaxnode) > abs(Fmin) % check for force model
59     Fmma = -Fmin; % use Fmin as maximum xmma value
60 else
61     Fmin = Fmin/Fmaxnode; % use fraction for constraint function
62     Fmma = Fmaxnode; % use maximum force per node as maximum xmma
        value
63 end
64 Uarray = 1:2*(nx+1)*(ny+1); % define objective area
65 %% DEFINE ELEMENT RESTRICTIONS
66 shap = 0; % [0 = no restrictions, 1 = circle, 2 = custom]
67 area = 1; % [0 = no material (passive), 1 = material (
    active)]
68 nodr = [1:ny:nx*ny 2:ny:nx*ny]; % custom restricted nodes [1:ny:nx*ny]
69 %% PREPARE FINITE ELEMENT
70 N = 2*(nx+1)*(ny+1); % total element nodes
71 all = 1:2*(nx+1)*(ny+1); % all degrees of freedom
72 free = setdiff(all,fix); % free degrees of freedom
73 A11 = [12 3 -6 -3; 3 12 3 0; -6 3 12 -3; -3 0 -3 12]; % fem
74 A12 = [-6 -3 0 3; -3 -6 -3 -6; 0 -3 -6 3; 3 -6 3 -6]; % fem
75 B11 = [-4 3 -2 9; 3 -4 -9 4; -2 -9 -4 -3; 9 4 -3 -4]; % fem
76 B12 = [ 2 -3 4 -9; -3 2 9 -2; 4 9 2 3; -9 -2 3 2]; % fem
77 Ke = 1/(1-nu^2)/24*([A11 A12;A12' A11]+nu*[B11 B12;B12' B11]); % element
    stiffness matrix
78 nodes = reshape(1:(nx+1)*(ny+1),1+ny,1+nx); % create node numer matrix
79 dofvec = reshape(2*nodes(1:end-1,1:end-1)+1,nx*ny,1); % create dof vector
80 dofmat = repmat(dofvec,1,8)+repmat([0 1 2*ny+[2 3 0 1] -2 -1],nx*ny,1); %
    create dof matrix
81 iK = reshape(kron(dofmat,ones(8,1))',64*nx*ny,1); % build sparse i
82 jK = reshape(kron(dofmat,ones(1,8))',64*nx*ny,1); % build sparse j
83 %% PREPARE FILTER
84 iH = ones(nx*ny*(2*(ceil(rmin)-1)+1)^2,1); % build sparse i
85 jH = ones(size(iH)); % create sparse vector of ones
86 kH = zeros(size(iH)); % create sparse vector of zeros

```

```

87 m = 0; % index for filtering
88 for i = 1:nx % for each element calculate distance between ...
89     for j = 1:ny % elements' center for filtering
90         r1 = (i-1)*ny+j; % sparse value i
91         for k = max(i-(ceil(rmin)-1),1):min(i+(ceil(rmin)-1),nx) %
           center of element
92             for l = max(j-(ceil(rmin)-1),1):min(j+(ceil(rmin)-1),ny) %
               center of element
93                 r2 = (k-1)*ny+l; % sparse value 2
94                 m = m+1; % update index for filtering
95                 iH(m) = r1; % sparse vector for filtering
96                 jH(m) = r2; % sparse vector for filtering
97                 kH(m) = max(0,rmin-sqrt((i-k)^2+(j-l)^2)); % weight
                   factor
98             end
99         end
100     end
101 end
102 H = sparse(iH,jH,kH); % build filter
103 Hs = sum(H,2); % summation of filter
104 %% DEFINE ELEMENT RESTRICTIONS
105 x = vol*ones(ny,nx); % initial material distribution
106 if shap == 0 % no restrictions
107     efree = (1:nx*ny)'; % all elements are free
108     eres= []; % no restricted elements
109 elseif shap == 1 % circular restrictions
110     rest = zeros(ny,nx); % pre-allocate space
111     for i = 1:nx % start loop
112         for j = 1:ny % for each element
113             if sqrt((j-ny/2)^2+(i-nx/4)^2) < ny/2.5 % circular
               restriction
114                 rest(j,i) = 1; % write restriction
115                 if rest(j,i) == area % check for restriction
116                     x(j,i) = area; % store restrictions in material
                       distribution
117             end
118         end
119     end
120 end
121 elseif shap == 2 % custom restrictions
122     rest = zeros(ny*nx,1); % pre-allocate space
123     for i = 1:length(nodr) % write restriction
124         resti = nodr(i); % write restriction
125         rest(resti) = 1; % write restriction
126     end
127     rest = reshape(rest,ny,nx);
128     for i = 1:nx % start loop
129         for j = 1:ny % for each element
130             if rest(j,i) == area % check for restriction
131                 x(j,i) = area; % store restrictions in material
                       distribution
132             end
133         end

```

```

134     end
135     efree = find(rest ~= 1); % set free elements
136     eres = find(rest == 1); % set restricted elements
137 end
138 if fil == 0 || fil == 1 % sensitivity, density filter
139     xF = x; % set filtered design variables
140 elseif fil == 2 % heaviside filter
141     beta = 1; % hs filter
142     xTilde = x; % hs filter
143     xF = 1-exp(-beta*xTilde)+xTilde*exp(-beta); % set filtered design
        space
144 end
145 xFree = xF(efree); % define free design matrix
146 %% DEFINE STRUCTURAL
147 Fsiz = size(Fe,1); % size of load vector
148 F = sparse(Fe,Fn,Fv,N,Fsiz); % define load vector
149 %% DESIGN OF ACTUATOR DISTRIBUTION
150 wsiz = size(Fe,2); % size of actuator variables
151 wzer = zeros(wsiz,1); % empty row of zeros for mma usage
152 wF = F; % plugin initial force distribution
153 wval = F(Fe); % create vector of design variables
154 %% DEFINE MMA PARAMETERS
155 m = 1; % number of constraint functions
156 n = wsiz; % number of variables
157 xmin = -1*ones(n,1); % minimum values of x
158 xmax = -(1e-9/Fmma)*ones(wsiz,1); % maximum values of x
159 xold1 = zeros(n,1); % previous x, to monitor convergence
160 xold2 = xold1; % used by mma to monitor convergence
161 df0dx2 = zeros(n,1); % second derivative of the objective function
162 dfdx2 = zeros(m,n); % second derivative of the constraint function
163 low = xmin; % lower asymptotes from the previous iteration
164 upp = xmax; % upper asymptotes from the previous iteration
165 a0 = 1; % constant a_0 in mma formulation [1]
166 a = zeros(m,1); % constant a_i in mma formulation
167 cmma = 1e3*ones(m,1); % constant c_i in mma formulation
168 d = zeros(m,1); % constant d_i in mma formulation
169 subs = 200; % maximum number of subsolv iterations [200]
170 %% PRE-ALLOCATE SPACE
171 npx = zeros(length(fix),1)'; % pre-allocate constraint dots
172 npy = zeros(length(fix),1)'; % pre-allocate constraint dots
173 npfx = zeros(length(Fe),1)'; % pre-allocate force dots
174 npfy = zeros(length(Fe),1)'; % pre-allocate force dots
175 npdx = zeros(length(nodes),1)'; % pre-allocate force dots
176 npdy = zeros(length(nodes),1)'; % pre-allocate force dots
177 U = zeros(size(F)); % pre-allocate space displacement
178 c = zeros(miter,1); % pre-allocate objective vector
179 L = zeros(N,1); % pre-allocate selection tensor
180 labda = zeros(N,1); % pre-allocate lagrange multiplier
181 Fi = zeros(1,N); % pre-allocate force selection vector
182 Cons = zeros(miter,1); % pre-allocate constraint vector
183 %% DEFINE SELECTION TENSOR
184 for j = Uarray % for each iteration..
185     if mod(j,2) == 0 % ...check for horizontal or vertical

```

```

186         L(j) = 1;           % vertical selection value
187     else
188         L(j) = 1;           % horizontal selection value
189     end
190 end
191 %% INITIALIZE LOOP
192 iter = 0;                   % initialize loop
193 diff = 1;                   % initialize convergence criterion
194 loopbeta = 1;               % initialize beta-loop
195 %% START LOOP
196 while ((diff > tol) || (iter < piter+1)) && iter < miter % convergence
    criterion not met
197     loopbeta = loopbeta + 1; % iteration loop for hs filter
198     iter = iter+1;           % define iteration
199     if pcon == 1             % use continuation method
200         if iter <= piter % first number of iterations...
201             p = 1;           %... set penalty 1
202             s = 0.5;         %... set penalty 0.5 for actuator design
203         elseif iter > piter % after a number of iterations...
204             p = min(pen,pcinc*p); % ... set continuation penalty
205             s = min(sen,1.06*s); % ... set continuation penalty actuator
                design
206         end
207     elseif pcon == 0         % not using continuation method
208         p = pen;             % set penalty
209         s = sen;             % set penalty actuator design
210     end
211 %% Selfweight
212 if rho ~= 0                 % gravity is involved
213     xP=zeros(ny,nx); % pre-allocate space
214     xP(xF>0.25) = xF(xF>0.25).^p; % normal penalization
215     xP(xF<=0.25) = xF(xF<=0.25).*(0.25^(p-1)); % below pseudo-density
216     Fsw = zeros(N,1); % pre-allocate self-weight
217     for i=1:nx*ny % for each element, set gravitational...
218         Fsw(dofmat(i,2:2:end))=Fsw(dofmat(i,2:2:end))-xF(i)*rho
                *9.81/4;
219     end % force to the attached nodes
220     Fsw=repmat(Fsw,1,size(F,2)); % set self-weight for load cases
221 elseif rho == 0            % no gravity
222     xP = xF.^p;             % penalized design variable
223     Fsw = 0;                % no selfweight
224 end
225 wP = atan(s*wF)/atan(s); % penalized actuator variable
226 Ftot = Fmma*(wP) + Fsw; % total force
227 %% Finite element analysis
228 kK = reshape(Ke(:)*(Emin+xP(:)')*(E-Emin)),64*nx*ny,1); % create
    sparse vector k
229 K = sparse(iK,jK,kK); % combine sparse vectors
230 K = (K+K')/2; % build stiffness matrix
231 U(free,:) = K(free,free)\Ftot(free,:); % displacement solving
232 c(iter) = 0; % set compliance to zero
233 Sens = 0; % set sensitivity to zero
234 Senw = 0; % set constraint sensitivity to zero

```

```

235     Cons(iter) = 0;           % set constraint to zero
236     Senc = ones(1,N);       % set constraint sensitivity
237     %% Calculate compliance and sensitivity
238     for i = 1:size(Fn,2) % for number of load cases
239         Ui = U(:,i);        % displacement per load case
240         c0 = reshape(sum((Ui(dofmat)*Ke).*Ui(dofmat),2),ny,nx); % initial
                compliance
241         c(iter) = c(iter) - sum(sum(Ui)); % objective
242         labda(free) = -K(free,free)\L(free); % calculate lagrange
                multiplie
243         Fi(Fe) = (Fmma*s./((s^2*wF(Fe).^2+1)*(atan(s))));% force
                selection vector
244         FFi = spdiags(Fi',0, N,N); % force selection vector
245         Sens = Sens + FFi(Fe,Fe)*labda(Fe); % calculate sensitivity
246         Cons(iter) = Cons(iter) + Fmma*(Fmin/sum(sum(wF)))-1; % calculate
                constraint
247         dCdf = Senc(Fe)'*Fmma*full(Fmin)/-(sum(sum(full(wF))))^2; %
                constraint sensitivity
248         if iter == 2 % finite difference method
249             wF1 = wF; % store first force vector
250             [~,S1] = max(abs(Sens(:))); % calculate maximum sensitivity
                value
251             Sens1 = Sens(S1); % store maximum sensitivity value
252             [~,S2] = max(abs(dCdf(:))); % calculate maximum sensitivity
                value
253             Sens2 = dCdf(S2); % store maximum sensitivity value
254         end
255     end
256
257     if fil == 0 % optimality criterion with sensitivity filter
258         Sens(:) = Sens; % update filtered sensitivity
259         Sencw(:) = Senc; % update filtered sensitivity
260     elseif fil == 1 % optimality criterion with density filter
261         Sens(:) = Sens; % update filtered sensitivity of constraint
262         Sencw(:) = Senc; % update filtered sensitivity of constraint
263     elseif fil == 2 % optimality criterion with heaviside filter
264         dx = beta*exp(-beta*xTilde)+exp(-beta); % update hs parameter
265         Sens(:) = H*(Sens(:).*dx(:)./Hs); % update filtered sensitivity
266         Sencw(:) = Senc; % update filtered sensitivity of constraint
267     end
268     %% Update design variables Optimality Criterion
269     if sol == 0 % use optimality criterion method
270         l1 = 0; % initial lower bound for lagranian mulitplier
271         l2 = 1e9; % initial upper bound for lagranian multiplier
272         while (l2-l1)/(l1+l2) > 1e-3 % start loop
273             lag = 0.5*(l1+l2); % average of lagranian interval
274             xnew = max(0,max(x-move,min(1,min(x+move,x.*sqrt(-Sens./Senc/
                lag))))); % update element densities
275             if fil == 0 % sensitivity filter
276                 xF = xnew; % updated result
277             elseif fil == 1 % density filter
278                 xF(:) = (H*xnew(:))./Hs; % updated filtered density
                result

```

```

279         elseif fil == 2 % heaviside filter
280             xTilde(:) = (H*xnew(:))./Hs; % set filtered density
281             xF(:) = 1-exp(-beta*xTilde)+xTilde*exp(-beta); % updated
                result
282         end
283         if shap == 1 % restriction is on
284             xF(rest==1) = area; % set restricted area
285         end
286         if sum(xF(:)) > vol*nx*ny % check for optimum
287             l1 = lag; % update lower bound to average
288         else
289             l2 = lag; % update upper bound to average
290         end
291     end
292     %% Method of moving asymptotes
293     elseif sol == 1 % use mma solver
294         xval = wval(:); % store current design variable for mma
295         if iter == 1 % for the first iteration...
296             cscale = 1/c(iter); % ...set scaling factor for mma solver
297         end
298         f0 = c(iter)*cscale; % objective at current design variable for
                mma
299         df0dx = Sens*cscale; % store sensitivity for mma
300         f = Cons(iter); % normalized constraint function
301         dfdx = dCdf; % derivative constraint function
302         [xmma,~,~,~,~,~,~,~,~,low,upp] = ...
303             mmasub(m,n,iter,xval,xmin,xmax,xold1,xold2, ...
304                 f0,df0dx,df0dx2,f,dfdx,dfdx2,low,upp,a0,a,cmma,d,subs); % mma
                solver
305         xold2 = xold1; % used by mma to monitor convergence
306         xold1 = wval(:); % previous x, to monitor convergence
307         xnew = xF; % update density result
308         wnew = wF; % update force result
309         wnew(Fe) = xmma(1:end); % include mma result
310         if fil == 0 % sensitivity filter
311             xF = xnew; % update design variables
312         elseif fil == 1 % density filter
313             xF(:) = (H*xnew(:))./Hs; % update filtered densities result
314         elseif fil == 2 % heaviside filter
315             xTilde(:) = (H*xnew(:))./Hs; % filtered result
316             xF(:) = 1-exp(-beta*xTilde)+xTilde*exp(-beta); % update design
                variable
317         end
318         if shap == 1 || shap == 2 % if restrictions enableed
319             xF(rest==1) = area; % set restricted area
320         end
321         wF(:) = wnew(:); % update support variables
322         wval = wnew(Fe); % update support variables
323     end
324     diff = max(abs(full(Fmma*wnew(:))-full(F(:)))); % difference of
                maximum element change
325     F = Fmma*wnew; % update design variable

```

```

326     if fil == 2 && beta < 512 && pen == p(end) && (loopbeta >= 50 || diff
327         <= tol) % hs filter
328         beta = 2*beta; % increase beta-factor
329         fprintf('beta now is %3.0f\n',beta) % display increase of b-
330             factor
331         loopbeta = 0; % set hs filter loop to zero
332         diff = 1; % set convergence to initial value
333     end
334     %% Finite difference method
335     if (fincheck == 1 || fincheck == 2) % check for finite difference
336         method
337         if iter == 2 % on first findif iteration
338             wF = wF1; % store first findif result...
339             wF(Fe(S1)) = wF1(Fe(S1))+h; %...and add a small pertubation
340         elseif iter == 3 % on second findif iteration
341             findif = (c(3)-c(2))/h; % calculate finite difference method
342             Sensdif = abs(max((findif-Sens1)/Sens1,(Sens1-findif)/findif)
343                 ); % maximum difference
344             if Sensdif > 0.01 % when difference between sensitivity and
345                 findif is too much display
346                 disp(['Warning: Sensitivity needs to be checked, max
347                     difference:' sprintf('%10.2f',Sensdif)])
348                 if fincheck == 2 % when fincheck is not accomplished...
349                     break %... break the loop and stop the code
350                 end
351             end
352             wF = wF1; % store first findif result...
353             wF(Fe(S2)) = wF1(Fe(S2))+h; %...and add a small pertubation
354         elseif iter == 4 % on third findif iteration
355             findif2 = (Cons(4)-Cons(2))/h; % calculate finite difference
356             method
357             Sensdif2 = abs(max((findif2-Sens2)/Sens2,(Sens2-findif2)/
358                 findif2)); % maximum difference
359             if Sensdif2 > 0.01 % when difference between sensitivity and
360                 findif is too much display
361                 disp(['Warning: Sensitivity needs to be checked, max
362                     difference:' sprintf('%10.2f',Sensdif2)])
363                 if fincheck == 2 % when fincheck is not accomplished...
364                     break %... break the loop and stop the code
365                 end
366             end
367         end
368     end
369     %% Store results into database X
370     X(:, :, iter) = xF; % each element value x is stored for each
371         iteration
372     C(iter) = c(iter); % each compliance is stored for each iteration
373     W(:, :, iter) = full(wF); % each force variable is stored for each
374         iteration
375     assignin('base', 'X', X); % each iteration (3rd dimension)
376     assignin('base', 'C', C); % each iteration (3rd dimension)
377     assignin('base', 'W', W); % each iteration (3rd dimension)
378     %% Results

```

```

367     if dis == 1           % display iterations
368         disp([' Iter:' sprintf('%4i',iter) ' Obj:' sprintf('%10.4f',c(
            iter)) ...
369             ' Ftot:' sprintf('%6.3f',sum(full(wP(:)))) ' Diff:' sprintf('
                %6.3f',diff)]]);
370     elseif dis == 2     % display parts of iterations
371         if iter == 1 || iter == disiter
372             if iter == 1
373                 disiter = plotiter;
374             elseif iter == disiter
375                 disiter = disiter + plotiter;
376             end
377             disp([' Iter:' sprintf('%4i',iter) ' Obj:' sprintf('%10.4f',c
                (iter)) ...
378                 ' Ftot:' sprintf('%6.3f',sum(full(wP(:)))) ' Diff:'
                    sprintf('%6.3f',diff)]]);
379         end
380     end
381     if draw == 1        % plot iterations
382         figure(1)
383         subplot(2,1,1)
384         colormap(gray); imagesc(1-xF);
385         set(gca, 'XTick', [], 'YTick', [], 'XTicklabel', [], ...
386             'YTicklabel', [], 'xcolor', '[0.7 0.7 0.7]', 'ycolor', '[0.7 0.7
                0.7]');
387         xlabel(sprintf('c = %.2f',c(iter)), 'Color', 'k')
388         axis equal; axis tight
389         drawnow;
390         hold on
391         if iter == 1
392             % Plot coloured dots for constraints
393             for i = 1:length(fix)
394                 npx(i) = ceil(fix(i)/(2*(ny+1))) - 0.5;
395                 nplot = ceil(fix(i)/2);
396                 while nplot > (ny+1)
397                     nplot = nplot - (ny+1);
398                 end
399                 npy(i) = nplot - 0.5;
400             end
401             plot(npx, npy, 'r.', 'MarkerSize', 20)
402         end
403         % Plot coloured dots for force application
404         Fmaxplot = min(min(full(F)));
405         for i = 1:length(Fe)
406             if F(Fe(i)) < wplot*Fmaxplot
407                 npfx(i) = ceil(Fe(i)/(2*(ny+1))) - 0.5;
408                 nplot = ceil(Fe(i)/2);
409                 while nplot > (ny+1)
410                     nplot = nplot - (ny+1);
411                 end
412                 npfy(i) = nplot - 0.5;
413             end
414         end

```

```

415     if iter > 1
416         delete(Dof)
417     end
418     if exist('npfx','var')
419         Dof = plot(npfx(npfx(:)>0),npfy(npfy(:)>0),'b.','MarkerSize'
420                 ,20);
421         clear npfx; clear npfy;
422         uistack(Dof,'top')
423     end
424     % Plot coloured arrows for force application
425     if (((diff < tol) && iter >= piter+1) || iter >= miter)
426         for i = 1:length(Fe)
427             npfx(i) = ceil(Fe(i)/(2*(ny+1)))-0.5;
428             nplot = ceil(Fe(i)/2);
429             while nplot > (ny+1)
430                 nplot = nplot-(ny+1);
431             end
432             npfy(i) = nplot-0.5;
433         end
434         for i = 1:length(Fe)
435             if F(Fe(i)) < wplot*Fmaxplot
436                 headsize = 1/sqrt(length(nonzeros(F(Fe)<0.5*Fmaxplot)
437                 ));
438                 if mod(Fe(i),2)
439                     arrowz([npfx(i) npfy(i)],[npfx(i)+0.5*ny*F(Fe(i))
440                             /Fmaxplot npfy(i)],headsize,2,[0 0 1])
441                 else
442                     arrowz([npfx(i) npfy(i)],[npfx(i) npfy(i)+0.5*ny*
443                             F(Fe(i))/Fmaxplot],headsize,2,[0 0 1])
444                 end
445             end
446         end
447     end
448     % Plot compliance plot
449     figure(1)
450     subplot(2,1,2)
451     plot(c(1:iter))
452     set(gca,'YTick',[],'YTicklabel',[])
453     xlabel('Iterations')
454     ylabel('Compliance')
455     xaxmax = c(iter);
456     yaxmax = max(c);
457     yaxmin = min(c(1:iter));
458     if pcon == 0
459         yaxmax = mean([yaxmin yaxmax]);
460     end
461     ylim([0.95*yaxmin yaxmax])
462     xlim([1 min(iter+10,miter)])
463     elseif draw == 2 % plot parts of iterations
464         if iter == 1 || iter == drawiter
465             if iter == 1
466                 drawiter = plotiter;
467             elseif iter == drawiter

```

```

464         drawiter = drawiter + plotiter;
465     end
466     figure(1)
467     subplot(2,1,1)
468     colormap(gray); imagesc(1-xF);
469     set(gca, 'XTick', [], 'YTick', [], 'XTicklabel', [], ...
470         'YTicklabel', [], 'xcolor', '[0.7 0.7 0.7]', 'ycolor', '[0.7
         0.7 0.7]');
471     xlabel(sprintf('c = %.2f', c(iter)), 'Color', 'k')
472     axis equal; axis tight
473     drawnow;
474     hold on
475     if iter == 1
476         % Plot coloured dots for constraints
477         for i = 1:length(fix)
478             npx(i) = ceil(fix(i)/(2*(ny+1))) - 0.5;
479             nplot = ceil(fix(i)/2);
480             while nplot > (ny+1)
481                 nplot = nplot - (ny+1);
482             end
483             npy(i) = nplot - 0.5;
484         end
485         plot(npx, npy, 'r.', 'MarkerSize', 20)
486     end
487     % Plot coloured dots for force application
488     Fmaxplot = min(min(full(F)));
489     for i = 1:length(Fe)
490         if F(Fe(i)) < wplot*Fmaxplot
491             npfx(i) = ceil(Fe(i)/(2*(ny+1))) - 0.5;
492             nplot = ceil(Fe(i)/2);
493             while nplot > (ny+1)
494                 nplot = nplot - (ny+1);
495             end
496             npfy(i) = nplot - 0.5;
497         end
498     end
499     if iter > 1
500         delete(Dof)
501     end
502     if exist('npfx', 'var')
503         Dof = plot(npfx(npfx(:) > 0), npfy(npfy(:) > 0), 'b.', '
         MarkerSize', 20);
504         clear npfx; clear npfy;
505         uistack(Dof, 'top')
506     end
507     % Plot coloured arrows for force application
508     if (((diff < tol) && iter >= piter+1) || iter >= miter)
509         for i = 1:length(Fe)
510             npfx(i) = ceil(Fe(i)/(2*(ny+1))) - 0.5;
511             nplot = ceil(Fe(i)/2);
512             while nplot > (ny+1)
513                 nplot = nplot - (ny+1);
514             end

```

```

515         npfy(i) = nplot - 0.5;
516     end
517     for i = 1:length(Fe)
518         if F(Fe(i)) < wplot*Fmaxplot
519             headsize = 1/sqrt(length(nonzeros(F(Fe) < 0.5*
520                 Fmaxplot)));
521             if mod(Fe(i), 2)
522                 arrowz([npfx(i) npfy(i)], [npfx(i) + 0.5*ny*F(Fe
523                     (i))/Fmaxplot npfy(i)], headsize, 2, [0 0 1])
524             else
525                 arrowz([npfx(i) npfy(i)], [npfx(i) npfy(i)
526                     + 0.5*ny*F(Fe(i))/Fmaxplot], headsize, 2, [0 0
527                         1])
528             end
529         end
530     end
531 end
532 % Plot compliance plot
533 figure(1)
534 subplot(2,1,2)
535 plot(c(1:iter))
536 set(gca, 'YTick', [], 'YTicklabel', [])
537 xlabel('Iterations')
538 ylabel('Compliance')
539 xaxmax = c(iter);
540 yaxmax = max(c);
541 yaxmin = min(c(1:iter));
542 if pcon == 0
543     yaxmax = mean([yaxmin yaxmax]);
544 end
545 ylim([0.95*yaxmin yaxmax])
546 xlim([1 min(iter+10, iter)])
547 end
548 end
549 end
550 %% ONLY DISPLAY FINAL RESULT
551 if dis == 0 || dis == 2 % display final result
552     disp([' Iter:' sprintf('%4i', iter) ' Obj:' sprintf('%10.4f', c(iter))
553         ...
554         ' Ftot:' sprintf('%6.3f', sum(full(wP(:)))) ' Diff:' sprintf('%6.3
555             f', diff)]);
556 end
557 if draw == 0 || draw == 2 % plot final result
558     figure(1)
559     subplot(2,1,1)
560     colormap(gray); imagesc(1-xF);
561     axis equal; axis tight;
562     set(gca, 'XTick', [], 'YTick', [], 'XTicklabel', [], ...
563         'YTicklabel', [], 'xcolor', '[0.7 0.7 0.7]', 'ycolor', '[0.7 0.7 0.7]'
564     )
565     xlabel(sprintf('c = %.2f', c(iter)), 'Color', 'k')
566     drawnow;
567     hold on

```

```

561 % Plot coloured dots for constraints
562 for i = 1:length(fix)
563     npx(i) = ceil(fix(i)/(2*(ny+1)))-0.5;
564     nplot = ceil(fix(i)/2);
565     while nplot > (ny+1)
566         nplot = nplot-(ny+1);
567     end
568     npy(i) = nplot-0.5;
569 end
570 plot(npx,npy,'r.','MarkerSize',20)
571 % Plot coloured dots for force application
572 Fmaxplot = min(min(full(F)));
573 for i = 1:length(Fe)
574     if F(Fe(i)) < wplot*Fmaxplot
575         npfx(i) = ceil(Fe(i)/(2*(ny+1)))-0.5;
576         nplot = ceil(Fe(i)/2);
577         while nplot > (ny+1)
578             nplot = nplot-(ny+1);
579         end
580         npfy(i) = nplot-0.5;
581     end
582 end
583 if iter > 1
584     delete(Dof)
585 end
586 if exist('npfx','var')
587     Dof = plot(npfx(npfx(:)>0),npfy(npfy(:)>0),'b.','MarkerSize',20);
588     clear npfx; clear npfy;
589     uistack(Dof,'top')
590 end
591 % Plot coloured arrows for force application
592 if (((diff < tol) && iter >= piter+1) || iter >= miter)
593     for i = 1:length(Fe)
594         npfx(i) = ceil(Fe(i)/(2*(ny+1)))-0.5;
595         nplot = ceil(Fe(i)/2);
596         while nplot > (ny+1)
597             nplot = nplot-(ny+1);
598         end
599         npfy(i) = nplot-0.5;
600     end
601     for i = 1:length(Fe)
602         if F(Fe(i)) < wplot*Fmaxplot
603             headsize = 1/sqrt(length(nonzeros(F(Fe)<0.5*Fmaxplot)));
604             if mod(Fe(i),2)
605                 arrowz([npfx(i) npfy(i)],[npfx(i)+0.5*ny*F(Fe(i))/
606                     Fmaxplot npfy(i)],headsize,2,[0 0 1])
607             else
608                 arrowz([npfx(i) npfy(i)],[npfx(i) npfy(i)+0.5*ny*F(Fe
609                     (i))/Fmaxplot],headsize,2,[0 0 1])
610             end
611         end
612     end
613 end

```

```

612 % Plot compliance plot
613 if adv == 0
614     figure(1)
615     subplot(2,1,2)
616     plot(c(1:iter))
617     set(gca, 'YTick', [], 'YTicklabel', [])
618     xlabel('Iterations')
619     ylabel('Compliance')
620     xaxmax = c(iter);
621     yaxmax = max(c);
622     yaxmin = min(c(1:iter));
623     if pcon == 0
624         yaxmax = mean([yaxmin yaxmax]);
625     end
626     ylim([0.95*yaxmin yaxmax])
627     xlim([1 min(iter+10, miter)])
628 end
629 end
630 %% PLOTTING DISPLACEMENT
631 if (def == 1 || def == 2)
632     FileName = ['Displacement_', datestr(now, 'ddmm_HHMMSS'), '.avi']; %
        dynamic filename
633     vidObj = VideoWriter(FileName);
634     vidObj.FrameRate = 3;
635     figure(1)
636     subplot(2,1,1)
637     xaxis = get(gca, 'XLim');
638     yaxis = get(gca, 'YLim');
639     open(vidObj);
640     figure(2)
641     clear mov
642     colormap(gray);
643     Umov = 1; % start movie counter
644     Uim = zeros(5642,1);
645     Uim(2:2:end) = Ui(2:2:end);
646     Uim(1:2:end) = -Ui(1:2:end);
647     Umax = -10/max(abs(Uim)); % define maximum displacement
648     steps = 1; % number of displacement steps
649     set(gca, 'nextplot', 'replacechildren');
650     Upatch = zeros(nx*ny,1);
651     for i = 1:ny*nx
652         Uindex = 2*(i+floor((i-1)/ny))-1+[1 2 2*(ny+1)+1 2*(ny+1)+3];
653         Upatch(i,1) = mean(U(Uindex));
654     end
655     Upatch = reshape(Upatch, ny, nx);
656     Upatchmin = min(min(Upatch));
657     Upatchnorm = -Upatch/Upatchmin;
658     for Udisp = linspace(Umax/steps, Umax, steps) % vary input displacement
659         clf
660         for ely = 1:ny % plot displacements...
661             for elx = 1:nx % for each element...
662                 if xF(ely, elx) > 0 % exclude white regions for plotting
                    purposes

```

```

663         n1 = (ny+1)*(elx-1)+ely;
664         n2 = (ny+1)* elx +ely;
665         Ue = Udisp*Uim([2*n1-1;2*n1; 2*n2-1;2*n2; 2*n2+1;2*n2
        +2; 2*n1+1;2*n1+2],1);
666         ly = ely-1; lx = elx-1;
667         xx = [Ue(1,1)+lx Ue(3,1)+lx+1 Ue(5,1)+lx+1 Ue(7,1)+lx
        ]';
668         yy = [-Ue(2,1)-ly -Ue(4,1)-ly -Ue(6,1)-ly-1 -Ue(8,1)-
        ly-1]';
669         patch([xx xx],[yy yy],[Upatchnorm(ely,elx) Upatchnorm
        (ely,elx)],'LineStyle','none');
670
671         end
672     end
673 end
674 colormap jet           % for better interpatation...
675 axis tight
676 axis equal
677 xticks([0 15 30 45 60 75 90])
678 box on
679 colorbar
680 drawnow               % ...draw coloured densities
681 currFrame = getframe; % get current frame...
682 writeVideo(vidObj,currFrame); % ... write to video file
683 end
684 close(vidObj);
685 end
686 if def == 2           % when def equals 2...
687     implay(FileName) % ...open Matlab Movie Player
688 end
689 toc

```

B.10 Design of Actuator Placement Including Topology Optimization.m

In this section, the complete code of designing optimal actuator placement, including topology optimization is made available. By running this code, the produced picture in Figure 5-6 can be made immediately. The constraints are scaled, in order to prioritize the compliance constraint.

```

1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %
3  % Topology Optimization Using Matlab
4  % Design of Actuator Placement
5  %
6  % Delft University of Technology, Department PME
7  % Master of Science Thesis Project
8  %
9  % Stefan Broxterman
10 %
11 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
12 %
13 tic % start timer
14 %% DEFINE PARAMETERS
15 adv = 0; % use advanced function [0 = off, 1 = on]
16 if adv == 0 % define parameters at behalf of the advanced
17     function
18     nx = 90; % number of elements horizontal
19     ny = 30; % number of elements vertical
20     vol = 0.2; % volume fraction [0-1]
21     pen = 3; % penalty
22     rmin = 1.5; % filter size
23     fil = 1; % filter method [0 = sensitivity filtering, 1 =
24         density filtering, 2 = heaviside filtering]
25     clc; clf; close all; clear X; clear W; % clear workspace
26 end
27 %% DEFINE SOLUTION METHOD
28 sol = 1; % solution method [0 = oc(sens), 1 = mma]
29 pcon = 1; % use continuation method [0 = off, 1 = on]
30 fincheck = 1; % finite difference check [0 = off, 1 = on, 2 =
31     break]
32 %% DEFINE CALCULATION
33 tol = 0.001; % tolerance for convergence criterion [0.01]
34 move = 0.2; % move limit for lagrange [0.2]
35 pcinc = 1.03; % penalty continuation increasing factor [1.03]
36 piter = 20; % number of iteration for starting penalty [20]
37 miter = 1000; % maximum number of iterations [1000]
38 plotiter = 5; % gap of iterations used to plot or draw
39     iterations [5]
40 def = 0; % plot deformations [0 = off, 1 = on, 2 = play
41     video]
42 wplot = 0.20; % define treshold factor of Fmax for force plot
43     [0.20]

```

```

38 h = 1e-6;           % perturbation value for finite difference method
    [1e-6]
39 %% DEFINE OUTPUT
40 draw = 1;           % plot iterations [0 = off, 1 = on, 2 = partial]
41 dis = 1;           % display iterations [0 = off, 1 = on, 2 =
    partial]
42 %% DEFINE MATERIAL
43 E = 1;              % young's modulus of solid [1]
44 Emin = 1e-9;       % young's modulus of void [1e-9]
45 nu = 0.3;          % poisson ratio [0.3]
46 rho = 0e-3;        % density [0e-3]
47 g = 9.81;          % gravitational acceleration [9.81]
48 %% DEFINE FORCE
49 Fe = 2*(ny+1)+45*2*(ny+1):2*(ny+1):2*(ny+1)*(nx+1); % element of force
    application [2*(ny+1)+45*2*(ny+1):2*(ny+1):2*(ny+1)*(nx+1)]
50 Fn = 1;            % number of applied force locations [1]
51 Fv = -1/length(Fe); % value of applied force [-1]
52 %% DEFINE SUPPORTS
53 fix = 1:2*(ny+1);  % fixed degrees of freedom [1:2*(ny+1)]
54 %% DEFINE DESIGN OF ACTUATOR
55 Fmaxnode = 1;      % define max force per node [1]
56 Fmin = -1;         % minimal force constraint [1]
57 sen = 5;           % penalty for actuator design [5]
58 if abs(Fmaxnode) > abs(Fmin) % check for force model
59     Fmma = -Fmin;   % use Fmin as maximum xmma value
60 else
61     Fmin = Fmin/Fmaxnode; % use fraction for constraint function
62     Fmma = Fmaxnode;   % use maximum force per node as maximum xmma
    value
63 end
64 Uarray = 2:2:2*(nx+1)*(ny+1); % define objective area
65 %% DEFINE ELEMENT RESTRICTIONS
66 shap = 0;          % [0 = no restrictions, 1 = circle, 2 = custom]
67 area = 1;          % [0 = no material (passive), 1 = material (
    active)]
68 nodr = [1:ny:nx*ny 2:ny:nx*ny]; % custom restricted nodes [1:ny:nx*ny]
69 %% PREPARE FINITE ELEMENT
70 N = 2*(nx+1)*(ny+1); % total element nodes
71 all = 1:2*(nx+1)*(ny+1); % all degrees of freedom
72 free = setdiff(all,fix); % free degrees of freedom
73 A11 = [12 3 -6 -3; 3 12 3 0; -6 3 12 -3; -3 0 -3 12]; % fem
74 A12 = [-6 -3 0 3; -3 -6 -3 -6; 0 -3 -6 3; 3 -6 3 -6]; % fem
75 B11 = [-4 3 -2 9; 3 -4 -9 4; -2 -9 -4 -3; 9 4 -3 -4]; % fem
76 B12 = [ 2 -3 4 -9; -3 2 9 -2; 4 9 2 3; -9 -2 3 2]; % fem
77 Ke = 1/(1-nu^2)/24*([A11 A12;A12' A11]+nu*[B11 B12;B12' B11]); % element
    stiffness matrix
78 nodes = reshape(1:(nx+1)*(ny+1),1+ny,1+nx); % create node numer matrix
79 dofvec = reshape(2*nodes(1:end-1,1:end-1)+1,nx*ny,1); % create dof vector
80 dofmat = repmat(dofvec,1,8)+repmat([0 1 2*ny+[2 3 0 1] -2 -1],nx*ny,1); %
    create dof matrix
81 iK = reshape(kron(dofmat,ones(8,1))',64*nx*ny,1); % build sparse i
82 jK = reshape(kron(dofmat,ones(1,8))',64*nx*ny,1); % build sparse j
83 %% PREPARE FILTER

```

```

84 iH = ones(nx*ny*(2*(ceil(rmin)-1)+1)^2,1); % build sparse i
85 jH = ones(size(iH)); % create sparse vector of ones
86 kH = zeros(size(iH)); % create sparse vector of zeros
87 m = 0; % index for filtering
88 for i = 1:nx % for each element calculate distance between ...
89     for j = 1:ny % elements' center for filtering
90         r1 = (i-1)*ny+j; % sparse value i
91         for k = max(i-(ceil(rmin)-1),1):min(i+(ceil(rmin)-1),nx) %
           center of element
92             for l = max(j-(ceil(rmin)-1),1):min(j+(ceil(rmin)-1),ny) %
              center of element
93                 r2 = (k-1)*ny+l; % sparse value 2
94                 m = m+1; % update index for filtering
95                 iH(m) = r1; % sparse vector for filtering
96                 jH(m) = r2; % sparse vector for filtering
97                 kH(m) = max(0,rmin-sqrt((i-k)^2+(j-l)^2)); % weight
                   factor
98             end
99         end
100     end
101 end
102 H = sparse(iH,jH,kH); % build filter
103 Hs = sum(H,2); % summation of filter
104 %% DEFINE ELEMENT RESTRICTIONS
105 x = vol*ones(ny,nx); % initial material distribution
106 if shap == 0 % no restrictions
107     efree = (1:nx*ny)'; % all elements are free
108     eres= []; % no restricted elements
109 elseif shap == 1 % circular restrictions
110     rest = zeros(ny,nx); % pre-allocate space
111     for i = 1:nx % start loop
112         for j = 1:ny % for each element
113             if sqrt((j-ny/2)^2+(i-nx/4)^2) < ny/2.5 % circular
              restriction
114                 rest(j,i) = 1; % write restriction
115                 if rest(j,i) == area % check for restriction
116                     x(j,i) = area; % store restrictions in material
                       distribution
117                 end
118             end
119         end
120     end
121 elseif shap == 2 % custom restrictions
122     rest = zeros(ny*nx,1); % pre-allocate space
123     for i = 1:length(nodr) % write restriction
124         resti = nodr(i); % write restriction
125         rest(resti) = 1; % write restriction
126     end
127     rest = reshape(rest,ny,nx);
128     for i = 1:nx % start loop
129         for j = 1:ny % for each element
130             if rest(j,i) == area % check for restriction

```

```

131             x(j,i) = area; % store restrictions in material
                    distribution
132             end
133         end
134     end
135     efree = find(rest ~= 1); % set free elements
136     eres = find(rest == 1); % set restricted elements
137 end
138 if fil == 0 || fil == 1 % sensitivity, density filter
139     xF = x; % set filtered design variables
140 elseif fil == 2 % heaviside filter
141     beta = 1; % hs filter
142     xTilde = x; % hs filter
143     xF = 1-exp(-beta*xTilde)+xTilde*exp(-beta); % set filtered design
                    space
144 end
145 xFree = xF(efree); % define free design matrix
146 %% DEFINE STRUCTURAL
147 Fsiz = size(Fe,1); % size of load vector
148 F = sparse(Fe,Fn,Fv,N,Fsiz); % define load vector
149 %% DESIGN OF ACTUATOR AND TOPOLOGY DISTRIBUTION
150 xsiz = size(xFree,1); % size of topology variables
151 wsiz = size(Fe,2); % size of actuator variables
152 xzer = zeros(xsiz,1); % empty row of zeros for mma usage
153 wzer = zeros(wsiz,1); % empty row of zeros for mma usage
154 wF = F; % plugin initial force distribution
155 wval = F(Fe); % create vector of design variables
156 %% DEFINE MMA PARAMETERS
157 m = 3; % number of constraint functions
158 n = xsiz+wsiz; % number of variables
159 xmin = [1e-9*ones(xsiz,1); -1*ones(wsiz,1)]; % minimum values of x
160 xmax = [ones(xsiz,1); -(1e-9/Fmma)*ones(wsiz,1)]; % maximum values of x
161 xold1 = zeros(n,1); % previous x, to monitor convergence
162 xold2 = xold1; % used by mma to monitor convergence
163 df0dx2 = zeros(n,1); % second derivative of the objective function
164 dfdx2 = zeros(m,n); % second derivative of the constraint function
165 low = xmin; % lower asymptotes from the previous iteration
166 upp = xmax; % upper asymptotes from the previous iteration
167 a0 = 1; % constant a_0 in mma formulation [1]
168 a = zeros(m,1); % constant a_i in mma formulation
169 cmma = 1e3*ones(m,1); % constant c_i in mma formulation
170 d = zeros(m,1); % constant d_i in mma formulation
171 subs = 50; % maximum number of subsolv iterations [200]
172 %% PRE-ALLOCATE SPACE
173 npx = zeros(length(fix),1)'; % pre-allocate constraint dots
174 npy = zeros(length(fix),1)'; % pre-allocate constraint dots
175 npfx = zeros(length(Fe),1)'; % pre-allocate force dots
176 npfy = zeros(length(Fe),1)'; % pre-allocate force dots
177 npdx = zeros(length(nodes),1)'; % pre-allocate force dots
178 npdy = zeros(length(nodes),1)'; % pre-allocate force dots
179 U = zeros(size(F)); % pre-allocate space displacement
180 c = zeros(miter,1); % pre-allocate objective vector
181 L = zeros(N,1); % pre-allocate selection tensor

```

```

182 labda = zeros(N,1);      % pre-allocate lagrange multiplier
183 labda2 = zeros(N,1);    % pre-allocate second lagrange multiplier
184 Fi = zeros(1,N);       % pre-allocate force selection vector
185 Ua = zeros(N,1);       % pre-allocate displacement vector
186 Cons = zeros(miter,1);  % pre-allocate constraint vector
187 Cons2 = zeros(miter,1); % pre-allocate constraint #2 vector
188 Cons3 = zeros(miter,1); % pre-allocate constraint #3 vector
189 %% DEFINE SELECTION TENSOR
190 for j = Uarray          % for each iteration..
191     if mod(j,2) == 0    % ...check for horizontal or vertical
192         L(j) = 1;      % vertical selection value
193     else
194         L(j) = 0;      % horizontal selection value
195     end
196 end
197 %% INITIALIZE LOOP
198 iter = 0;              % initialize loop
199 diff = 1;              % initialize convergence criterion
200 loopbeta = 1;          % initialize beta-loop
201 %% START LOOP
202 while ((diff > tol) || (iter < piter+1)) && iter < miter % convergence
    criterion not met
203     loopbeta = loopbeta + 1; % iteration loop for hs filter
204     iter = iter+1;          % define iteration
205     if pcon == 1           % use continuation method
206         if iter <= piter % first number of iterations...
207             p = 1;        %... set penalty 1
208             s = 0.5;      %... set penalty 0.5 for actuator design
209         elseif iter > piter % after a number of iterations...
210             p = min(pen,pcinc*p); % ... set continuation penalty
211             s = min(sen,1.06*s); % ... set continuation penalty actuator
                design
212         end
213     elseif pcon == 0      % not using continuation method
214         p = pen;          % set penalty
215         s = sen;          % set penalty actuator design
216     end
217     %% Selfweight
218     if rho ~= 0          % gravity is involved
219         xP=zeros(ny,nx); % pre-allocate space
220         xP(xF>0.25) = xF(xF>0.25).^p; % normal penalization
221         xP(xF<=0.25) = xF(xF<=0.25).*(0.25^(p-1)); % below pseudo-density
222         Fsw = zeros(N,1); % pre-allocate self-weight
223         for i=1:nx*ny    % for each element, set gravitational...
224             Fsw(dofmat(i,2:2:end))=Fsw(dofmat(i,2:2:end))-xF(i)*rho
                *9.81/4;
225         end              % force to the attached nodes
226         Fsw=repmat(Fsw,1,size(F,2)); % set self-weight for load cases
227     elseif rho == 0      % no gravity
228         xP = xF.^p;      % penalized design variable
229         Fsw = 0;         % no selfweight
230     end
231     wP = atan(s*wF)/atan(s); % penalized actuator variable

```

```

232 Ftot = Fmma*(wP) + Fsw; % total force
233 %% Finite element analysis
234 kK = reshape(Ke(:)*(Emin+xP(:)')*(E-Emin),64*nx*ny,1); % create
    sparse vector k
235 K = sparse(iK,jK,kK); % combine sparse vectors
236 K = (K+K')/2; % build stiffness matrix
237 Kt = K; % update total force
238 U(free,:) = Kt(free,free)\Ftot(free,:); % displacement solving
239 c(iter) = 0; % set compliance to zero
240 Sens = 0; % set sensitivity to zero
241 Senw = 0; % set constraint sensitivity to zero
242 Cons(iter) = 0; % set constraint to zero
243 Senc = ones(1,N); % set constraint sensitivity
244 %% Calculate compliance and sensitivity
245 for i = 1:size(Fn,2) % for number of load cases
246     Ui = U(:,i); % displacement per load case
247     Ua(Uarray) = Ui(Uarray); % selection of displacement
248     c0 = reshape(sum((Ui(dofmat)*Ke).*Ui(dofmat),2),ny,nx); % initial
        compliance
249     c(iter) = c(iter) + sum(Ua.^2); % objective
250     labda(free) = -sparse(Kt(free,free))\sparse(2*Ua(free)); %
        calculate lagrange multiplier
251     labda2(free) = 2*Ua(free); % calculate second lagrange multiplier
252     c00 = reshape(sum((labda(dofmat)*Ke).*Ui(dofmat),2),ny,nx); %
        initial labda compliance
253     Fi(Fe) = (Fmma*s./((s^2*wF(Fe).^2+1)*(atan(s)))); % force
        selection vector
254     FFi = spdiags(Fi',0,N,N); % force selection vector
255     Sens = Sens + p*(E-Emin)*xF.(p-1).*c00; % calculate density
        sensitivity
256     Senw = Senw - FFi(Fe,Fe)*labda(Fe); % calculate force sensitivity
257     Cons(iter) = Cons(iter) + 10*(sum(xF(:))/(vol*nx*ny)-1); %
        calculate constraint
258     dCdx = 10*Senc(efree)/(vol*ny*nx); % constraint sensitivity
259     Cons2(iter) = Cons2(iter) + 10*(Fmin/sum(sum(wF)))-1; % calculate
        constraint
260     dCdf = 10*Senc(Fe)*Fmin/-(sum(sum(full(wF))))^2; % constraint
        sensitivity
261     Cons3(iter) = Cons3(iter) + (sum(sum((Emin+xF.^p*(E-Emin)).*c0))
        -50); % compliance constraint
262     dCCdx = -p*(E-Emin)*xF.(p-1).*c0; % constraint sensitivity
263     dCCdf = labda2(Fe)'*FFi(Fe,Fe); % constraint sensitivity
264     if iter == 2 % finite difference method
265         F1 = wF; % store force vector
266         X1 = xF; % store density vector
267         [~,S1] = max(abs(Sens(:))); % calculate maximum sensitivity
            value
268         Sens1 = Sens(S1); % store maximum sensitivity value
269         [~,S2] = max(abs(Senw(:))); % calculate maximum sensitivity
            value
270         Sens2 = Senw(S2); % store maximum sensitivity value
271         [~,S3] = max(abs(dCdx(:))); % calculate maximum sensitivity
            value

```

```

272         Sens3 = dCdx(S3); % store maximum sensitivity value
273         [~,S4] = max(abs(dCdf(:))); % calculate maximum sensitivity
                value
274         Sens4 = dCdf(S4); % store maximum sensitivity value
275         [~,S5] = max(abs(dCCdx(:))); % calculate maximum sensitivity
                value
276         Sens5 = dCCdx(S5); % store maximum sensitivity value
277         [~,S6] = max(abs(dCCdf(:))); % calculate maximum sensitivity
                value
278         Sens6 = dCCdf(S6); % store maximum sensitivity value
279     end
280 end
281 if fil == 0 % optimality criterion with sensitivity filter
282     Sens(:) = Sens; % update filtered sensitivity
283     Sencw(:) = Senc; % update filtered sensitivity
284 elseif fil == 1 % optimality criterion with density filter
285     Sens(:) = H*(Sens(:)./Hs); % update filtered sensitivity
286     Sencw(:) = Senc; % update filtered sensitivity of constraint
287 elseif fil == 2 % optimality criterion with heaviside filter
288     dx = beta*exp(-beta*xTilde)+exp(-beta); % update hs parameter
289     Sens(:) = H*(Sens(:).*dx(:)./Hs); % update filtered sensitivity
290     Sencw(:) = Senc; % update filtered sensitivity of constraint
291 end
292 %% Update design variables Optimality Criterion
293 if sol == 0 % use optimality criterion method
294     l1 = 0; % initial lower bound for lagrangian mulitplier
295     l2 = 1e9; % initial upper bound for lagrangian multiplier
296     while (l2-l1)/(l1+l2) > 1e-3 % start loop
297         lag = 0.5*(l1+l2); % average of lagrangian interval
298         xnew = max(0,max(x-move,min(1,min(x+move,x.*sqrt(-Sens./Senc/
                lag))))); % update element densities
299         if fil == 0 % sensitivity filter
300             xF = xnew; % updated result
301         elseif fil == 1 % density filter
302             xF(:) = (H*xnew(:))./Hs; % updated filtered density
                result
303         elseif fil == 2 % heaviside filter
304             xTilde(:) = (H*xnew(:))./Hs; % set filtered density
305             xF(:) = 1-exp(-beta*xTilde)+xTilde*exp(-beta); % updated
                result
306         end
307         if shap == 1 % restriction is on
308             xF(rest==1) = area; % set restricted area
309         end
310         if sum(xF(:)) > vol*nx*ny % check for optimum
311             l1 = lag; % update lower bound to average
312         else
313             l2 = lag; % update upper bound to average
314         end
315     end
316     %% Method of moving asymptotes
317 elseif sol == 1 % use mma solver

```

```

318     xval = [xFree(:);wval(:)]; % store current design variable for
        mma
319     if iter == 1 % for the first iteration...
320         cscale = 1/c(iter); % ...set scaling factor for mma solver
321     end
322     f0 = c(iter)*cscale; % objective at current design variable for
        mma
323     df0dx = [Sens(efree);Senw]*cscale; % store sensitivity for mma
324     f = [Cons(iter);Cons2(iter);Cons3(iter)]; % normalized constraint
        function
325     dfdx =[dCdx wzer'; xzer' dCdf; dCCdx(efree)' dCCdf]; % derivative
        constraint functions
326     [xmma,~,~,~,~,~,~,~,~,low,upp] = ...
327         mmasub(m,n,iter,xval,xmin,xmax,xold1,xold2, ...
328             f0,df0dx,df0dx2,f,dfdx,dfdx2,low,upp,a0,a,cmma,d,subs); % mma
        solver
329     xold2 = xold1; % used by mma to monitor convergence
330     xold1 = [xFree(:);wval(:)]; % previous x, to monitor convergence
331     xnew = xF; % update density result
332     wnew = wF; % update force result
333     xnew(efree) = xmma(1:xsiz); % update mma to density
334     wnew(Fe) = xmma(xsiz+1:end); % update mma to force
335     if fil == 0 % sensitivity filter
336         xF = xnew; % update design variables
337     elseif fil == 1 % density filter
338         xF(:) = (H*xnew(:))./Hs; % update filtered densities result
339     elseif fil == 2 % heaviside filter
340         xTilde(:) = (H*xnew(:))./Hs; % filtered result
341         xF(:) = 1 - exp(-beta*xTilde) + xTilde*exp(-beta); % update design
        variable
342     end
343     if shap == 1 || shap == 2 % if restrictions enableed
344         xF(rest==1) = area; % set restricted area
345     end
346     wF(:) = wnew(:); % update support variables
347     xFree = xnew(efree); % update density variable
348     wval = wnew(Fe); % update support variables
349     end
350     diff = (max(abs(full(Fmma*wnew(:))-full(F(:))))+max(abs(xnew(:)-x(:)
        ))); % difference of maximum element change
351     x = xnew; % update design variable density
352     F = Fmma*wnew; % update design variable force
353     if fil == 2 && beta < 512 && pen == p(end) && (loopbeta >= 50 || diff
        <= tol) % hs filter
354         beta = 2*beta; % increase beta-factor
355         fprintf('beta now is %3.0f\n',beta) % display increase of b-
        factor
356         loopbeta = 0; % set hs filter loop to zero
357         diff = 1; % set convergence to initial value
358     end
359     %% Finite difference method
360     if (fincheck == 1 || fincheck == 2) % check for finite difference
        method

```

```

361     if iter == 2      % on first findif iteration
362         xF = X1;      % store first findif result...
363         xF(S1) = X1(S1)+h; %...and add a small perturbation
364         wF = F1;      % store first findif result
365     elseif iter == 3 % on second findif iteration
366         findif = (c(3)-c(2))/h; % calculate finite difference method
367         Sensdif = abs(max((findif-Sens1)/Sens1,(Sens1-findif)/findif)
368             ); % maximum difference
369         if Sensdif > 0.01 % when difference between sensitivity and
370             findif is too much display
371             disp(['Warning: Sensitivity needs to be checked, max
372                 difference:' sprintf('%10.2f',Sensdif)])
373             if fincheck == 2 % when fincheck is not accomplished...
374                 break %... break the loop and stop the code
375             end
376         end
377         wF = F1;      % store first findif result...
378         wF(Fe(S2)) = F1(Fe(S2))+h; %...and add a small perturbation
379         xF = X1;      % store first findif result
380     elseif iter == 4 % on third findif iteration
381         findif2 = (c(4)-c(2))/h; % calculate finite difference method
382         Sensdif2 = abs(max((findif2-Sens2)/Sens2,(Sens2-findif2)/
383             findif2)); % maximum difference
384         if Sensdif2 > 0.01 % when difference between sensitivity and
385             findif is too much display
386             disp(['Warning: Sensitivity needs to be checked, max
387                 difference:' sprintf('%10.2f',Sensdif2)])
388             if fincheck == 2 % when fincheck is not accomplished...
389                 break %... break the loop and stop the code
390             end
391         end
392         wF = F1;      % store first findif result
393         xF = X1;      % store first findif result...
394         xF(S3) = xF(S3)+h; %...and add a small perturbation
395     elseif iter == 5 % on fourth findif iteration
396         findif3 = (Cons(5)-Cons(2))/h; % calculate finite difference
397         method
398         Sensdif3 = abs(max((findif3-Sens3)/Sens3,(Sens3-findif3)/
399             findif3)); % maximum difference
400         if Sensdif3 > 0.01 % when difference between sensitivity and
401             findif is too much display
402             disp(['Warning: Sensitivity needs to be checked, max
403                 difference:' sprintf('%10.2f',Sensdif3)])
404             if fincheck == 2 % when fincheck is not accomplished...
405                 break %... break the loop and stop the code
406             end
407         end
408         wF = F1;      % store first findif result
409         wF(S4) = wF(S4)+h; % store first findif result...
410         xF = X1; %...and add a small perturbation
411     elseif iter == 6 % on fifth findif iteration
412         findif4 = (Cons2(6)-Cons2(2))/h; % calculate finite
413         difference method

```

```

403     Sensdif4 = abs(max((findif4-Sens4)/Sens4,(Sens4-findif4)/
404         findif4)); % maximum difference
405     if Sensdif4 > 0.01 % when difference between sensitivity and
406         findif is too much display
407         disp(['Warning: Sensitivity needs to be checked, max
408             difference:' sprintf('%10.2f',Sensdif4)])
409         if fincheck == 2 % when fincheck is not accomplished...
410             break %... break the loop and stop the code
411         end
412     end
413     wF = F1; % store first findif result
414     xF = X1; % store first findif result...
415     xF(S5) = xF(S5)+h; %...and add a small perturbation
416 elseif iter == 7 % on sixth findif iteration
417     findif5 = (Cons3(7)-Cons3(2))/h; % calculate finite
418         difference method
419     Sensdif5 = abs(max((findif4-Sens5)/Sens5,(Sens5-findif5)/
420         findif5)); % maximum difference
421     if Sensdif5 > 0.01 % when difference between sensitivity and
422         findif is too much display
423         disp(['Warning: Sensitivity needs to be checked, max
424             difference:' sprintf('%10.2f',Sensdif5)])
425         if fincheck == 2 % when fincheck is not accomplished...
426             break %... break the loop and stop the code
427         end
428     end
429     wF = F1; % store first findif result...
430     wF(Fe(S6)) = F1(Fe(S6))+h; %...and add a small perturbation
431     xF = X1; % store first findif result
432 elseif iter == 8 % on second finidif iteration
433     findif6 = (Cons3(8)-Cons3(2))/h; % calculate finite
434         difference method
435     Sensdif6 = abs(max((findif6-Sens6)/Sens6,(Sens6-findif6)/
436         findif6)); % maximum difference
437     if Sensdif6 > 0.01 % when difference between sensitivity and
438         findif is too much display
439         disp(['Warning: Sensitivity needs to be checked, max
440             difference:' sprintf('%10.2f',Sensdif6)])
441         if fincheck == 2 % when fincheck is not accomplished...
442             break %... break the loop and stop the code
443         end
444     end
445 end
446 end
447 %% Store results into database X
448 X(:, :, iter) = xF; % each element value x is stored for each
449 iteration
450 C(iter) = c(iter); % each compliance is stored for each iteration
451 W(:, :, iter) = full(wF); % each force variable is stored for each
452 iteration
453 assignin('base', 'X', X); % each iteration (3rd dimension)
454 assignin('base', 'C', C); % each iteration (3rd dimension)
455 assignin('base', 'W', W); % each iteration (3rd dimension)

```

```

443     %% Results
444     if dis == 1           % display iterations
445         disp([' Iter:' sprintf('%4i',iter) ' Obj:' sprintf('%10.4f',c(
446             iter)) ...
447             ' Vol:' sprintf('%6.3f',mean(xF(:))) ' Ftot:' sprintf('%6.3f',
448             sum(full(F))) ' Diff:' sprintf('%6.3f',diff)]);
449     elseif dis == 2     % display parts of iterations
450         if iter == 1 || iter == disiter
451             if iter == 1
452                 disiter = plotiter;
453             elseif iter == disiter
454                 disiter = disiter + plotiter;
455             end
456             disp([' Iter:' sprintf('%4i',iter) ' Obj:' sprintf('%10.4f',c(
457                 iter)) ...
458                 ' Vol:' sprintf('%6.3f',mean(xF(:))) ' Ftot:' sprintf('
459                 %6.3f', ...
460                 sum(full(F))) ' Diff:' sprintf('%6.3f',diff)]);
461         end
462     end
463     if draw == 1        % plot iterations
464         figure(1)
465         subplot(2,1,1)
466         colormap(gray); imagesc(1-xF);
467         set(gca, 'XTick', [], 'YTick', [], 'XTicklabel', [], ...
468             'YTicklabel', [], 'xcolor', '[0.7 0.7 0.7]', 'ycolor', '[0.7 0.7
469             0.7]');
470         xlabel(sprintf('c = %.2f',c(iter)), 'Color', 'k')
471         axis equal; axis tight
472         drawnow;
473         hold on
474         if iter == 1
475             % Plot coloured dots for constraints
476             for i = 1:length(fix)
477                 npx(i) = ceil(fix(i)/(2*(ny+1))) - 0.5;
478                 nplot = ceil(fix(i)/2);
479                 while nplot > (ny+1)
480                     nplot = nplot - (ny+1);
481                 end
482                 npy(i) = nplot - 0.5;
483             end
484             plot(npx, npy, 'r.', 'MarkerSize', 20)
485         end
486         % Plot coloured dots for force application
487         Fmaxplot = min(min(full(F)));
488         for i = 1:length(Fe)
489             if F(Fe(i)) < wplot*Fmaxplot
490                 npfx(i) = ceil(Fe(i)/(2*(ny+1))) - 0.5;
491                 nplot = ceil(Fe(i)/2);
492                 while nplot > (ny+1)
493                     nplot = nplot - (ny+1);
494                 end
495             end
496         end
497     end

```

```

491         npfy(i) = nplot - 0.5;
492     end
493 end
494 if iter > 1
495     delete(Dof)
496 end
497 if exist('npfx','var')
498     Dof = plot(npfx(npfx(:) > 0), npfy(npfy(:) > 0), 'b.', 'MarkerSize',
499             ,20);
500     clear npfx; clear npfy;
501     uistack(Dof, 'top')
502 end
503 % Plot coloured arrows for force application
504 if (((diff < tol) && iter >= piter+1) || iter >= miter)
505     for i = 1:length(Fe)
506         npfx(i) = ceil(Fe(i)/(2*(ny+1))) - 0.5;
507         nplot = ceil(Fe(i)/2);
508         while nplot > (ny+1)
509             nplot = nplot - (ny+1);
510         end
511         npfy(i) = nplot - 0.5;
512     end
513     for i = 1:length(Fe)
514         if F(Fe(i)) < wplot*Fmaxplot
515             headsize = 1/sqrt(length(nonzeros(F(Fe) < 0.5*Fmaxplot)
516             ));
517             if mod(Fe(i),2)
518                 arrowz([npfx(i) npfy(i)], [npfx(i)+0.5*ny*F(Fe(i))
519                 /Fmaxplot npfy(i)], headsize, 2, [0 0 1])
520             else
521                 arrowz([npfx(i) npfy(i)], [npfx(i) npfy(i)+0.5*ny*
522                 F(Fe(i))/Fmaxplot], headsize, 2, [0 0 1])
523             end
524         end
525     end
526 end
527 end
528 % Plot compliance plot
529 figure(1)
530 subplot(2,1,2)
531 plot(c(1:iter))
532 set(gca, 'YTick', [], 'YTicklabel', [])
533 xlabel('Iterations')
534 ylabel('Compliance')
535 xaxmax = c(iter);
536 yaxmax = max(c);
537 yaxmin = min(c(1:iter));
538 if pcon == 0
539     yaxmax = mean([yaxmin yaxmax]);
540 end
541 ylim([0.95*yaxmin yaxmax])
542 xlim([1 min(iter+10, miter)])
543 elseif draw == 2 % plot parts of iterations
544     if iter == 1 || iter == drawiter

```

```

540         if iter == 1
541             drawiter = plotiter;
542         elseif iter == drawiter
543             drawiter = drawiter + plotiter;
544         end
545         figure(1)
546         subplot(2,1,1)
547         colormap(gray); imagesc(1-xF);
548         set(gca, 'XTick', [], 'YTick', [], 'XTicklabel', [], ...
549             'YTicklabel', [], 'xcolor', ' [0.7 0.7 0.7]', 'ycolor', ' [0.7
550             0.7 0.7] ');
551         xlabel(sprintf('c = %.2f', c(iter)), 'Color', 'k')
552         axis equal; axis tight
553         drawnow;
554         hold on
555         if iter == 1
556             % Plot coloured dots for constraints
557             for i = 1:length(fix)
558                 npx(i) = ceil(fix(i)/(2*(ny+1))) - 0.5;
559                 nplot = ceil(fix(i)/2);
560                 while nplot > (ny+1)
561                     nplot = nplot - (ny+1);
562                 end
563                 npy(i) = nplot - 0.5;
564             end
565             plot(npx, npy, 'r.', 'MarkerSize', 20)
566         end
567         % Plot coloured dots for force application
568         Fmaxplot = max(max(full(F)));
569         for i = 1:length(Fe)
570             if F(Fe(i)) > wplot * Fmaxplot
571                 npfx(i) = ceil(Fe(i)/(2*(ny+1))) - 0.5;
572                 nplot = ceil(Fe(i)/2);
573                 while nplot > (ny+1)
574                     nplot = nplot - (ny+1);
575                 end
576                 npfy(i) = nplot - 0.5;
577             end
578         end
579         if iter > 1
580             delete(Dof)
581         end
582         if exist('npfx', 'var')
583             Dof = plot(npfx(npfx(:) > 0), npfy(npfy(:) > 0), 'b.', '
584                 MarkerSize', 20);
585             clear npfx; clear npfy;
586             uistack(Dof, 'top')
587         end
588         % Plot coloured arrows for force application
589         if (((diff < tol) && iter >= piter+1) || iter >= miter)
590             for i = 1:length(Fe)
591                 npfx(i) = ceil(Fe(i)/(2*(ny+1))) - 0.5;
592                 nplot = ceil(Fe(i)/2);

```

```

591         while nplot > (ny+1)
592             nplot = nplot-(ny+1);
593         end
594         npfy(i) = nplot-0.5;
595     end
596     for i = 1:length(Fe)
597         if F(Fe(i)) > wplot*Fmaxplot
598             headsize = 1/sqrt(length(nonzeros(F(Fe)>0.5*
599                 Fmaxplot)));
600             if mod(Fe(i),2)
601                 arrowz([npfx(i) npfy(i)],[npfx(i)+0.5*ny*F(Fe
602                     (i))/Fmaxplot npfy(i)],headsize,2,[0 0 1])
603             else
604                 arrowz([npfx(i) npfy(i)],[npfx(i) npfy(i)
605                     +0.5*ny*F(Fe(i))/Fmaxplot],headsize,2,[0 0
606                         1])
607             end
608         end
609     end
610     end
611     end
612     % Plot compliance plot
613     figure(1)
614     subplot(2,1,2)
615     plot(c(1:iter))
616     set(gca,'YTick',[],'YTicklabel',[])
617     xlabel('Iterations')
618     ylabel('Compliance')
619     xaxmax = c(iter);
620     yaxmax = max(c);
621     yaxmin = min(c(1:iter));
622     if pcon == 0
623         yaxmax = mean([yaxmin yaxmax]);
624     end
625     ylim([0.95*yaxmin yaxmax])
626     xlim([1 min(iter+10,iter)])
627 end
628 end
629 end
630 %% ONLY DISPLAY FINAL RESULT
631 if dis == 0 || dis == 2 % display final result
632     disp([' Iter:' sprintf('%4i',iter) ' Obj:' sprintf('%10.4f',c(iter))
633         ...
634         ' Vol:' sprintf('%6.3f',mean(xF(:))) ' Ftot:' sprintf('%6.3f',
635         ...
636         sum(full(wP(:)))) ' Diff:' sprintf('%6.3f',diff)]);
637 end
638 if draw == 0 || draw == 2 % plot final result
639     figure(1)
640     subplot(2,1,1)
641     colormap(gray); imagesc(1-xF);
642     axis equal; axis tight;
643     set(gca,'XTick',[],'YTick',[],'XTicklabel',[],...

```

```

637         'YTicklabel',[], 'xcolor', '[0.7 0.7 0.7]', 'ycolor', '[0.7 0.7 0.7]'
638     )
639     xlabel(sprintf('c = %.2f',c(iter)), 'Color', 'k')
640     drawnow;
641     hold on
642     % Plot coloured dots for constraints
643     for i = 1:length(fix)
644         npx(i) = ceil(fix(i)/(2*(ny+1))) - 0.5;
645         nplot = ceil(fix(i)/2);
646         while nplot > (ny+1)
647             nplot = nplot - (ny+1);
648         end
649         npy(i) = nplot - 0.5;
650     end
651     plot(npx, npy, 'r.', 'MarkerSize', 20)
652     % Plot coloured dots for force application
653     Fmaxplot = max(max(full(F)));
654     for i = 1:length(Fe)
655         if F(Fe(i)) > wplot*Fmaxplot
656             npfx(i) = ceil(Fe(i)/(2*(ny+1))) - 0.5;
657             nplot = ceil(Fe(i)/2);
658             while nplot > (ny+1)
659                 nplot = nplot - (ny+1);
660             end
661             npfy(i) = nplot - 0.5;
662         end
663     end
664     if iter > 1
665         delete(Dof)
666     end
667     if exist('npfx', 'var')
668         Dof = plot(npfx(npfx(:) > 0), npfy(npfy(:) > 0), 'b.', 'MarkerSize', 20);
669         clear npfx; clear npfy;
670         uistack(Dof, 'top')
671     end
672     % Plot coloured arrows for force application
673     if (((diff < tol) && iter >= piter+1) || iter >= miter)
674         for i = 1:length(Fe)
675             npfx(i) = ceil(Fe(i)/(2*(ny+1))) - 0.5;
676             nplot = ceil(Fe(i)/2);
677             while nplot > (ny+1)
678                 nplot = nplot - (ny+1);
679             end
680             npfy(i) = nplot - 0.5;
681         end
682         for i = 1:length(Fe)
683             if F(Fe(i)) > wplot*Fmaxplot
684                 headsize = 1/sqrt(length(nonzeros(F(Fe) > 0.5*Fmaxplot)));
685                 if mod(Fe(i), 2)
686                     arrowz([npfx(i) npfy(i)], [npfx(i)+0.5*ny*F(Fe(i))/
687                         Fmaxplot npfy(i)], headsize, 2, [0 0 1])
688                 else

```

```

687         arrowz([npfx(i) npfy(i)], [npfx(i) npfy(i)+0.5*ny*F(Fe
        (i))/Fmaxplot], headsiz, 2, [0 0 1])
688     end
689     end
690     end
691 end
692 % Plot compliance plot
693 if adv == 0
694     figure(1)
695     subplot(2,1,2)
696     plot(c(1:iter))
697     set(gca, 'YTick', [], 'YTicklabel', [])
698     xlabel('Iterations')
699     ylabel('Compliance')
700     xaxmax = c(iter);
701     yaxmax = max(c);
702     yaxmin = min(c(1:iter));
703     if pcon == 0
704         yaxmax = mean([yaxmin yaxmax]);
705     end
706     ylim([0.95*yaxmin yaxmax])
707     xlim([1 min(iter+10, miter)])
708 end
709 end
710 %% PLOTTING DISPLACEMENT
711 if (def == 1 || def == 2)
712     FileName = ['Displacement_', datestr(now, 'ddmm_HHMMSS'), '.avi']; %
        dynamic filename
713     vidObj = VideoWriter(FileName);
714     vidObj.FrameRate = 3;
715     figure(1)
716     subplot(2,1,1)
717     xaxis = get(gca, 'XLim');
718     yaxis = get(gca, 'YLim');
719     open(vidObj);
720     figure(2)
721     clear mov
722     colormap(gray);
723     Umov = 1; % start movie counter
724     Uim = zeros(5642,1);
725     Uim(2:2:end) = Ui(2:2:end);
726     Uim(1:2:end) = -Ui(1:2:end);
727     Umax = -10/max(abs(Uim)); % define maximum displacement
728     steps = 1; % number of displacement steps
729     set(gca, 'nextplot', 'replacechildren');
730     Upatch = zeros(nx*ny,1);
731     for i = 1:ny*nx
732         Uindex = 2*(i+floor((i-1)/ny))-1+[1 2 2*(ny+1)+1 2*(ny+1)+3];
733         Upatch(i,1) = mean(U(Uindex));
734     end
735     Upatch = reshape(Upatch, ny, nx);
736     Upatchmin = min(min(Upatch));
737     Upatchnorm = -Upatch/Upatchmin;

```

```

738     for Udisp = linspace(Umax/steps,Umax,steps) % vary input displacement
739         clf
740         for ely = 1:ny % plot displacements...
741             for elx = 1:nx % for each element...
742                 if xF(ely,elx) > 0 % exclude white regions for plotting
743                     purposes
744                         n1 = (ny+1)*(elx-1)+ely;
745                         n2 = (ny+1)* elx +ely;
746                         Ue = Udisp*Uim([2*n1-1;2*n1; 2*n2-1;2*n2; 2*n2+1;2*n2
747                             +2; 2*n1+1;2*n1+2],1);
748                         ly = ely-1; lx = elx-1;
749                         xx = [Ue(1,1)+lx Ue(3,1)+lx+1 Ue(5,1)+lx+1 Ue(7,1)+lx
750                             ]';
751                         yy = [-Ue(2,1)-ly -Ue(4,1)-ly -Ue(6,1)-ly-1 -Ue(8,1)-
752                             ly-1]';
753                         patch([xx xx],[yy yy],[Upatchnorm(ely,elx) Upatchnorm
754                             (ely,elx)], 'LineStyle','none');
755
756                 end
757             end
758         end
759         colormap jet % for better interpolation...
760         axis tight
761         axis equal
762         xticks([0 15 30 45 60 75 90])
763         box on
764         colorbar
765         drawnow % ...draw coloured densities
766         currFrame = getframe; % get current frame...
767         writeVideo(vidObj,currFrame); % ... write to video file
768     end
769     close(vidObj);
770 end
771 if def == 2 % when def equals 2...
772     implay(FileName) % ...open Matlab Movie Player
773 end
774 toc

```

Appendix C

Add-in Codes

In this section some add-ins can be found. Keep in mind: it is highly recommended to not just copy and paste the code, but type it yourself. By this way, the user could actually achieve some knowledge over the changes made, and also overcome copy-paste problems.

The add-ins are split up in the following parts: making use of the MMA solution (C.1), using restrictive regions (C.2), solving multiple load cases (C.3), implementing self-weight (C.4), using the continuity method (C.5) and using different filtering techniques (C.6).

Up to here, all functions for two dimensional cases are described. A third dimension can be introduced by applying (C.7). An add-in to be able to calculate compliant mechanisms can be seen in (C.8). Design of supports can be implemented by following (C.9).

Design of actuator placement can be implemented by following the regime (C.10). When also implementing topology optimization, besides the actuator placement, make sure to implement (C.11).

C.1 Basic MMA Add-in.m

As a follow-up from (B.1), an extra implementation of the MMA solution is added (3.1). In the preamble, the solution method can be implemented [between line 23-24]:

```
1 %% DEFINE SOLUTION METHOD
2 sol = 1;           % solution method [0 = oc(sens), 1 = mma]
```

The parameters of this MMA can be implemented accordingly [between line 80-81]:

```
1 %% DEFINE MMA PARAMETERS
2 m = 1;           % number of constraint functions
3 n = size(xF(:,1)); % number of variables
4 xmin = zeros(n,1); % minimum values of x
5 xmax = ones(n,1); % maximum values of x
6 xold1 = zeros(n,1); % previous x, to monitor convergence
7 xold2 = xold1; % used by mma to monitor convergence
8 df0dx2 = zeros(n,1); % second derivative of the objective function
9 dfdx2 = zeros(1,n); % second derivative of the constraint function
10 low = xmin; % lower asymptotes from the previous iteration
11 upp = xmax; % upper asymptotes from the previous iteration
12 a0 = 1; % constant a_0 in mma formulation
13 a = zeros(m,1); % constant a_i in mma formulation
14 cmma = 1e3*ones(m,1); % constant c_i in mma formulation
15 d = zeros(m,1); % constant d_i in mma formulation
16 subs = 200; % maximum number of subsolv iterations
```

A simple if-loop needs to be implemented [between line 108-109]:

```
1 if sol == 0 % use optimality criterion method
```

The actual MMA algorithm can now be implemented [between line 120-121]:

```
1 % Method of moving asymptotes
2 elseif sol == 1 % use mma solver
3 xval = x(:); % store current design variable for mma
4 if iter == 1 % for the first iteration...
5 cscale = 1/c(iter); % ...set scaling factor for mma solver
6 end
7 f0 = c(iter)*cscale; % objective at current design variable for
% mma
8 df0dx = Sens(:)*cscale; % store sensitivity for mma
9 f = (sum(xF(:))/(vol*nx*ny)-1); % normalized constraint function
```

```
10     dfdx = Senc(:)'/(vol*ny*nx); % derivative of the constraint
      function
11     [xmma,~,~,~,~,~,~,~,~,low,upp] = ...
12         mmasub(m,n,iter,xval,xmin,xmax,xold1,xold2, ...
13             f0,df0dx,df0dx2,f,dfdx,dfd2,low,upp,a0,a,cmma,d,subs); % mma
      solver
14     xold2 = xold1; % used by mma to monitor convergence
15     xold1 = x(:); % previous x, to monitor convergence
16     xnew = xF; % update result
17     xnew(:) = xmma; % include restricted elements
18     xnew = reshape(xnew,ny,nx); % reshape xmma vector to original
      size
19     xF = xnew; % update design variables
20 end
```

Some external function are called, which are attached to this report (D.1) and (D.2). [line 12 from above]

C.2 Basic Restrictions Add-in.m

As a follow-up from (B.1), an extra implementation of restricted regions is added (3.2.1). In the preamble, the restricted element parameters is implemented [between line 40-41]:

```

1 %% DEFINE ELEMENT RESTRICTIONS
2 shap = 0;           % [0 = no restrictions, 1 = circle, 2 = custom]
3 area = 0;          % [0 = no material (passive), 1 = material (
    active)]
4 nodr = (round(ny/2)+(0:ny:(nx-1)*ny)); % custom restricted nodes

```

In order to make a work-around for the restrictive regions, an add-in needs to be implemented, before the loop, by making this addition a number of lines needs to be replaced [replace line 76-78]:

```

1 %% DEFINE ELEMENT RESTRICTIONS
2 x = repmat(vol,ny,nx); % initial material distribution
3 if shap == 0           % no restrictions
4     efree = (1:nx*ny)'; % all elements are free
5     eres = [];         % no restricted elements
6 elseif shap == 1      % restrictions
7     rest = zeros(ny,nx); % pre-allocate space
8     for i = 1:nx      % start loop
9         for j = 1:ny  % for each element
10            if sqrt((j-ny/2)^2+(i-nx/4)^2) < ny/2.5 % circular
                restriction
11                rest(j,i) = 1; % write restriction
12                if rest(j,i) == area % check for restriction
13                    x(j,i) = area; % store restrictions in material
                        distribution
14            end
15        end
16    end
17 end
18 efree = find(rest ~= 1); % set free elements
19 eres = find(rest == 1); % set restricted elements
20 end
21 xF = x;                 % set filtered design variables
22 xFree = xF(efree);     % define free design matrix
23 %% DEFINE STRUCTURAL

```

Only when using the MMA method, and already implemented all add-ins as described in (C.1), the restrictions vector needs to be initialized by the MMA method by replacing one variable [replace line 85]:

```

1 n = size(xFree(:),1); % number of variables

```

To set restricted area on, using the Optimality Criteria, a small add-in is made [between line 114-115]:

```

1      if shap == 1 % restriction is on
2          xF(rest==1) = area; % set restricted area
3      end

```

Only when using the MMA method, and already implemented all add-ins as described in (C.1) a smart implementations is made, while using the MMA solution, it can be helpful to skip all restrictive regions from the design space, and after the optimization simple plug them into the design space. This work-around should gain some time performance. The MMA code needs to be replaced [replace line 140-159]:

```

1      %% Method of moving asymptotes
2      elseif sol == 1 % use mma solver
3          xval = xFree(:); % store current design variable for mma
4          if iter == 1 % for the first iteration...
5              cscale = 1/c(iter); % ...set scaling factor for mma solver
6          end
7          f0 = c(iter)*cscale; % objective at current design variable for
           mma
8          df0dx = Sens(efree)*cscale; % store sensitivity for mma
9          f = (sum(xF(:))/(vol*nx*ny)-1); % normalized constraint function
10         dfdx = Senc(efree)'/(vol*ny*nx); % derivative of the constraint
           function
11         [xmma,~,~,~,~,~,~,~,low,upp] = ...
12             mmasub(m,n,iter,xval,xmin,xmax,xold1,xold2, ...
13                 f0,df0dx,df0dx2,f,dfdx,dfdx2,low,upp,a0,a,cmma,d,subs); % mma
           solver
14         xold2 = xold1; % used by mma to monitor convergence
15         xold1 = xFree(:); % previous x, to monitor convergence
16         xnew = xF; % update result
17         xnew(efree) = xmma; % include restricted elements
18         xnew = reshape(xnew,ny,nx); % reshape xmma vector to original
           size
19         xF = xnew; % update design variables
20         if shap == 1 % if restrictions enableed
21             xF(rest==1) = area; % set restricted area
22         end
23     end
24     xFree = xnew(efree); % set non-restricted area

```

C.3 Basic Load Cases Add-in.m

When applying multiple load cases, a small adjustment should be implemented (3.2.2). To adapt the program to perform the compliance and sensitivity analysis for the number of pre-defined load-cases, these lines should be replaced. [replace line 102-105]:

```

1     %% Calculate compliance and sensitivity
2     for i = 1:size(F,2) % for number of load cases
3         Ui = U(:,i); % displacement per load case
4         c0 = reshape(sum((Ui(dofmat)*Ke).*Ui(dofmat)),2),ny,nx); % initial
           compliance
5         c(iter) = c(iter) + sum(sum((Emin+xF.^p*(E-Emin)).*c0)); %
           calculate compliance
6         Sens = Sens -p*(E-Emin)*xF.^(p-1).*c0; % sensitivity
7     end

```

When performing multiple load cases, the force vector needs to be defined accordingly. The example shown in (Figure 3-4b) can be build by replacing the force vector. [replace line 35-38]:

```

1 %% DEFINE FORCE
2 Fe = [2*(nx+1)*(ny+1) 2*(nx+1)*(ny+1)-1]; % element node of applied
           force
3 Fn = [1 2];
4 Fv = [-1 1];

```

C.4 Basic Self-weight Add-in.m

When adding self-weight to the optimization problem some additions can be implemented (3.2.3). The self-weight parameters can be defined [between line 34-35]:

```
1 rho = 0e-3;           % density [0e-3]
2 g = 9.81;            % gravitational acceleration [9.81]
```

When gravity is involved, the force of the gravity needs to be added to the external force [between line 94-95]:

```
1 %% Selfweight
2 if rho ~= 0           % gravity is involved
3     xP=zeros(ny,nx); % pre-allocate space
4     xP(xF>0.25) = xF(xF>0.25).^p; % normal penalization
5     xP(xF<=0.25) = xF(xF<=0.25).*(0.25^(p-1)); % below pseudo-density
6     Fsw = zeros(N,1); % pre-allocate self-weight
7     for i=1:nx*ny     % for each element, set gravitational...
8         Fsw(dofmat(i,2:2:end))=Fsw(dofmat(i,2:2:end))-xF(i)*rho
9             *9.81/4;
10    end                % force to the attached nodes
11    Fsw= repmat(Fsw,1,size(F,2)); % set self-weight for load cases
12 elseif rho == 0     % no gravity
13     xP = xF.^p;      % penalized design variable
14     Fsw = 0;         % no selfweight
15 end
16 Ftot = F + Fsw;     % total force
```

To adapt the finite element analysis to the added self-weight, some replacements are needed [replace line 95-99]:

```
1 %% Finite element analysis
2 kK = reshape(Ke(:)*(Emin+xP(:)'.*(E-Emin)),64*nx*ny,1); % create
3     sparse vector k
4 K = sparse(iK,jK,kK); % combine sparse vectors
5 K = (K+K')/2;        % build stiffness matrix
6 U(free,:) = K(free,free)\Ftot(free,:); % displacement solving
```

The sensitivity analysis needs to be adapted accordingly by replacing one line [replace line 105]:

```
1 Sens = Sens + reshape(2*Ui(dofmat)*repmat([0;-9.81*rho/4],4,1),ny
2     ,nx) -p*(E-Emin)*xF.^(p-1).*c0; % sensitivity
```

C.5 Basic Continuity Add-in.m

To implement the continuity approach, a number of changes needs to be made to the program (3.2.4). First, one line of specifying whether or not the user want the continuity method [between line 23-24]:

```
1 pcon = 0;           % use continuation method [0 = off, 1 = on]
```

Next, the continuity parameters can be defined [between line 26-27]:

```
1 pcinc = 1.03;      % penalty continuation increasing factor [1.03]
2 piter = 20;        % number of iteration for starting penalty [20]
```

The iteration loop needs to be adapted, in order to fulfill the maximum "continuity" iterations, these replacements should be made [replace line 91-94]:

```
1 while ((diff > tol) || (iter < piter+1)) && iter < miter % convergence
   criterion not met
2   iter = iter+1;      % define iteration
3   if pcon == 1       % use continuation method
4       if iter <= piter % first number of iterations...
5           p = 1;      %... set penalty 1
6       elseif iter > piter % after a number of iterations...
7           p = min(pen,pcinc*p); % ... set continuation penalty
8       end
9   elseif pcon == 0   % not using continuation method
10      p = pen;        % set penalty
11  end
```

In order to display the compliance of the iterations at a correct scale, one adaption should be made to the plotting code. [between line 171-172]:

```
1     if pcon == 0
2         yaxmax = mean([yaxmin yaxmax]);
3     end
```

The exact same addition should be added further on [between line 218-219]:

```
1     if pcon == 0
2         yaxmax = mean([yaxmin yaxmax]);
3     end
```

C.6 Basic Filters Add-in.m

During the report some filtering techniques are introduced. (3.2.5) both the sensitivity, density and the heaviside projection filter are implemented in this section. At first, determine the type of filter [between line 21-22]:

```
1      fil = 0;                % filter method [0 = sensitivity filtering, 1 =
      density filtering, 2 = heaviside filtering]
```

Set the design space accordingly to the filter technique specified [between line 75-76]:

```
1  if fil == 0 || fil == 1 % sensitivity, density filter
2      xF = x;                % set filtered design variables
3  elseif fil == 2        % heaviside filter
4      beta = 1;            % hs filter
5      xTilde = x;         % hs filter
6      xF = 1-exp(-beta*xTilde)+xTilde*exp(-beta); % set filtered design
      space
7  end
```

Initialize the loop-number of β [between line 90-91]:

```
1  loopbeta = 1;            % initialize beta-loop
```

Make sure this β is updated during each loop [between line 92-93]:

```
1      loopbeta = loopbeta +1; % iteration loop for hs filter
```

The sensitivity analysis needs to be adapted to the new filter inputs, by replacing the original sensitivity line [replace line 107]:

```
1      if fil == 0          % optimality criterion with sensitivity filter
2          Sens(:) = H*(x(:).*Sens(:))./Hs./max(1e-3,x(:)); % update
      filtered sensitivity
3      elseif fil == 1     % optimality criterion with density filter
4          Sens(:) = H*(Sens(:)./Hs); % update filtered sensitivity
5          Senc(:) = H*(Senc(:)./Hs); % update filtered sensitivity of
      constraint
6      elseif fil == 2     % optimality criterion with heaviside filter
7          dx = beta*exp(-beta*xTilde)+exp(-beta); % update hs parameter
```

```

8         Sens(:) = H*(Sens(:).*dx(:)./Hs); % update filtered sensitivity
9         Senc(:) = H*(Senc(:).*dx(:)./Hs); % update filtered sensitivity
           of constraint
10    end

```

After the Optimality Criteria, the design variables needs to be filtered accordingly by replacing the update step [replace line 114]:

```

1         if fil == 0 % sensitivity filter
2             xF = xnew; % updated result
3         elseif fil == 1 % density filter
4             xF(:) = (H*xnew(:))./Hs; % updated filtered density
           result
5         elseif fil == 2 % heaviside filter
6             xTilde(:) = (H*xnew(:))./Hs; % set filtered density
7             xF(:) = 1-exp(-beta*xTilde)+xTilde*exp(-beta); % updated
           result
8         end

```

Only when the MMA method is implemented (C.1), and only this implemented is done, replace the same line as above [replace line 158]:

```

1         if fil == 0 % sensitivity filter
2             xF = xnew; % updated result
3         elseif fil == 1 % density filter
4             xF(:) = (H*xnew(:))./Hs; % updated filtered density
           result
5         elseif fil == 2 % heaviside filter
6             xTilde(:) = (H*xnew(:))./Hs; % set filtered density
7             xF(:) = 1-exp(-beta*xTilde)+xTilde*exp(-beta); % updated
           result
8         end

```

After the optimization step, update the β of the heaviside projection filter. [between line 122-123]:

```

1 if fil == 2 && beta < 512 && pen == p(end) && (loopbeta >= 50 || diff <=
   tol) % hs filter
2     beta = 2*beta; % increase beta-factor
3     fprintf('beta now is %3.0f\n',beta) % display increase of b-
       factor
4     loopbeta = 0; % set hs filter loop to zero
5     diff = 1; % set convergence to initial value
6     end

```

C.7 3D Add-in.m

Up to here, only two dimensions are taken into account. However, there is an add-in available in order to upgrade the Basic (B.1) to three dimensions (3.3). This add-in consists of a lot of manipulations, replacements and additions. At first, define the number of lateral elements [between line 18-19]:

```
1      nz = 5;                % number of elements lateral
```

Because the discretization can vary over time, it is helpful to clear the big X-matrix each run [between line 21-22]:

```
1      clear X;              % clear the big X matrix
```

In order to force a black-white solution, a so-called gray-scale filter is implemented, the associated parameter and the option itself can be implemented to enable or disable the filter technique. [between line 27-28]:

```
1  graysc = 1;              % use gray-scale filter [0 = off, 1 = on]
2  q = 1;                   % gray-scale parameter
3  qmax = 2;                % maximum gray-scale parameter
4  plotiter = 5;           % gap of iterations used to plot or draw
    iterations
```

Because the third dimension costs a lot of computational time, it can be helpful to plot the iterations and graphics only partially by introducing the plotiter, which is a definition of the output steps of the iterations [replace line 34-36]:

```
1  plotiter = 5;           % gap of iterations used to plot or draw
    iterations
2  %% DEFINE OUTPUT
3  draw = 1;               % plot iterations [0 = off, 1 = on, 2 = partial]
4  dis = 1;                % display iterations [0 = off, 1 = on, 2 =
    partial]
```

The number of elements is increased, by the introduction of a third dimension. As an example, to reproduce the example shown in Figure 3-7, change the load [replace line 36]:

```
1 Fe = (3*(nx+1)*(ny+1)-1)+(3*(nx+1)*(ny+1))*(0:nz)'; % element of force
    application
```

The number of elements is increased, by the introduction of a third dimension. As an example, to reproduce the example shown in Figure 3-7, change the fixed locations [replace line 40]:

```
1 fix = repmat((1:3*(ny+1))',1,nz+1)+repmat((0:nz)*3*(nx+1)*(ny+1),length
    ((1:3*(ny+1))),1);
2 fix = fix(:); % fixed elements
```

The finite element analysis needs to be rebuilt. The number of changes is big, so the best way is by making a replacement of the current preparation of the finite elements [replace line 41-48]:

```
1 %% PREPARE FINITE ELEMENT
2 N = 3*(nx+1)*(ny+1)*(nz+1); % total elements nodes
3 all = 1:3*(nx+1)*(ny+1)*(nz+1); % all degrees of freedom
4 free = setdiff(all,fix); % free degrees of freedom
5 A = [32 6 -8 6 -6 4 3 -6 -10 3 -3 -3 -4 -8;
6     -48 0 0 -24 24 0 0 0 12 -12 0 12 12 12]; % fem
7 k = 1/72*A'*[1; nu]; % simple stiffness matrix
```

The next lines are meant to replace the element stiffness matrix and eventually introduce a new way to define the nodes and dof vectors and matrices [replace line 49-54]:

```
1 %% GENERATE SIX SUB-MATRICES AND THEN GET KE MATRIX
2 K1 = [k(1) k(2) k(2) k(3) k(5) k(5);
3     k(2) k(1) k(2) k(4) k(6) k(7);
4     k(2) k(2) k(1) k(4) k(7) k(6);
5     k(3) k(4) k(4) k(1) k(8) k(8);
6     k(5) k(6) k(7) k(8) k(1) k(2);
7     k(5) k(7) k(6) k(8) k(2) k(1)]; % stiffness matrix
8 K2 = [k(9) k(8) k(12) k(6) k(4) k(7);
9     k(8) k(9) k(12) k(5) k(3) k(5);
10    k(10) k(10) k(13) k(7) k(4) k(6);
11    k(6) k(5) k(11) k(9) k(2) k(10);
12    k(4) k(3) k(5) k(2) k(9) k(12)
13    k(11) k(4) k(6) k(12) k(10) k(13)]; % stiffness matrix
14 K3 = [k(6) k(7) k(4) k(9) k(12) k(8);
15    k(7) k(6) k(4) k(10) k(13) k(10);
16    k(5) k(5) k(3) k(8) k(12) k(9);
17    k(9) k(10) k(2) k(6) k(11) k(5);
18    k(12) k(13) k(10) k(11) k(6) k(4);
19    k(2) k(12) k(9) k(4) k(5) k(3)]; % stiffness matrix
20 K4 = [k(14) k(11) k(11) k(13) k(10) k(10);
```

```

21     k(11) k(14) k(11) k(12) k(9)  k(8);
22     k(11) k(11) k(14) k(12) k(8)  k(9);
23     k(13) k(12) k(12) k(14) k(7)  k(7);
24     k(10) k(9)  k(8)  k(7)  k(14) k(11);
25     k(10) k(8)  k(9)  k(7)  k(11) k(14)]; % stiffness matrix
26 K5 = [k(1) k(2)  k(8)  k(3) k(5)  k(4);
27       k(2) k(1)  k(8)  k(4) k(6)  k(11);
28       k(8) k(8)  k(1)  k(5) k(11) k(6);
29       k(3) k(4)  k(5)  k(1) k(8)  k(2);
30       k(5) k(6)  k(11) k(8) k(1)  k(8);
31       k(4) k(11) k(6)  k(2) k(8)  k(1)]; % stiffness matrix
32 K6 = [k(14) k(11) k(7)  k(13) k(10) k(12);
33       k(11) k(14) k(7)  k(12) k(9)  k(2);
34       k(7)  k(7)  k(14) k(10) k(2)  k(9);
35       k(13) k(12) k(10) k(14) k(7)  k(11);
36       k(10) k(9)  k(2)  k(7)  k(14) k(7);
37       k(12) k(2)  k(9)  k(11) k(7)  k(14)]; % stiffness matrix
38 Ke = 1/((nu+1)*(1-2*nu))*...
39     [ K1  K2  K3  K4;
40       K2' K5  K6  K3';
41       K3' K6  K5' K2';
42       K4  K3  K2  K1']; % element stiffness matrix
43 nodes = reshape(1:(nx+1)*(ny+1),1+ny,1+nx); % create node number matrix
44 nodes2 = reshape(nodes(1:end-1,1:end-1),ny*nx,1); % create node number
45         matrix
46 nodes3 = 0:(ny+1)*(nx+1):(nz-1)*(ny+1)*(nx+1); % create node number
47         matrix
48 nodes4 = repmat(nodes2, size(nodes3))+repmat(nodes3, size(nodes2)); %
49         create node number matrix
50 dofvec = 3*nodes4(:)+1; % create dof vector
51 dofmat = repmat(dofvec,1,24)+repmat([0 1 2 3*ny+[3 4 5 0 1 2] -3 -2 -1
52         3*(ny+1)*(nx+1) + [0 1 2 3*ny + [3 4 5 0 1 2] -3 -2 -1]],nx*ny*nz,1);
53         % create dof matrix
54 iK = kron(dofmat,ones(24,1))'; % build sparse i
55 jK = kron(dofmat,ones(1,24))'; % build sparse j

```

Now, the filter needs to be redefined to account for the third dimension, here again, the number of changes is big, so a complete replacement is recommended [replace line 55-77]:

```

1 %% PREPARE FILTER
2 iH = ones(nx*ny*nz*(2*(ceil(rmin)-1)+1)^2,1); % build sparse i
3 jH = ones(size(iH)); % create sparse vector of ones
4 kH = zeros(size(iH)); % create sparse vector of zeros
5 m = 0; % index for filtering
6 for h = 1:nz % for each element calculate...
7     for i = 1:nx % distance between elements'...
8         for j = 1:ny % centre for filtering
9             r1 = (h-1)*nx*ny + (i-1)*ny+j; % sparse value 1
10            for k2 = max(h-(ceil(rmin)-1),1):min(h+(ceil(rmin)-1),nz) %
11                centre of element

```

```

11         for k = max(i-(ceil(rmin)-1),1):min(i+(ceil(rmin)-1),nx)
           % centre of element
12         for l = max(j-(ceil(rmin)-1),1):min(j+(ceil(rmin)-1),
           ny) % centre of element
13             r2 = (k2-1)*nx*ny + (k-1)*ny+1; % sparse value 2
14             m = m+1; % update index for filtering
15             iH(m) = r1; % sparse vector for filtering
16             jH(m) = r2; % sparse vector for filtering
17             kH(m) = max(0,rmin-sqrt((i-k)^2+(j-l)^2)+(h-k2)
                ^2); % weight factor
18         end
19     end
20 end
21 end
22 end
23 end
24 H = sparse(iH,jH,kH); % build filter
25 Hs = sum(H,2); % summation of filter
26 %% DEFINE STRUCTURAL
27 x = repmat(vol,ny,nx,nz); % initial material distribution

```

The constraint dots are pre-allocated, with the introduction of a third dimensions, the pre-allocations should be implemented also [between line 83-84]:

```

1 npz = zeros(length(fix),1)'; % pre-allocate constraint dots

```

The force dots are pre-allocated, with the introduction of a third dimensions, the pre-allocations should be implemented also [between line 85-86]:

```

1 npfz = zeros(length(Fe),1)'; % pre-allocate force dots

```

The loop is starting, when the gray-scale filter is enabled, this implementation uses a continuation method, in order to apply the correct gray-scale filter parameter [between line 94-95]:

```

1     if gray == 1 % if grayscale is enabled
2         if iter <= 15 % within 15 iterations
3             q = 1; % don't use grayscale
4         else % after 15 iterations
5             q = min(qmax,1.01*q); % use continuation method to pick a
                gray-scale factor
6         end
7     end

```

The stiffness matrix is reshaped to account a third dimension, therefore the sparse vector k needs to be replaced [replace line 96]:

```
1      kK = Ke(:)*(Emin+xF(:)'.^p*(E-Emin)); % create sparse vector k
```

The compliance and sensitivity analysis needs to be replaced also to account for the newly introduced dimension. This can be done by a simple replacement of the lines [replace line 102-106]:

```
1      %% Calculate compliance and sensitivity
2      c0 = reshape(sum((U(dofmat)*Ke).*U(dofmat),2),ny,nx,nz); % initial
      compliance
3      c(iter) = c(iter) + sum(sum(sum((Emin+xF.^p*(E-Emin)).*c0))); %
      calculate compliance
4      Sens = Sens -p*(E-Emin)*xF.^(p-1).*c0; % sensitivity
5      Senc = ones(ny,nx,nz); % set constraint sensitivity
```

When the gray-scale filter is enabled, the Optimality Criteria method should be used, to filter the calculated elements to achieve a black-white solution [replace line 113]:

```
1      if q == 0          % don't use grayscale
2          xnew = max(0,max(x-move,min(1,min(x+move,x.*sqrt(-Sens./Senc/
      lag))))); % update element densities
3      elseif q == 1    % use grayscale
4          xnew = max(0,max(x-move,min(1,min(x+move,x.*sqrt(-Sens./Senc/
      lag)).^q))); % update element densities
5      end
```

The optimum solution within the Optimality Criteria should also account for the third dimension [replace line 115]:

```
1      if sum(xF(:)) > vol*nx*ny*nz; % check for optimum
```

The third dimension should be stored also in the big X-matrix, by replacing the existing line [replace line 124]:

```
1      X(:,:,iter) = xF; % each element value x is stored for each
      iteration
```

In order to show a graphical output of the results, a complete rewritten part of the code is needed. The changes from plot to plot3d are that big, all the results lines should be replaced with the following collection of lines [replace line 128-223]:

```

1     %% Results
2     if dis == 1           % display iterations
3         disp([' Iter:' sprintf('%4i',iter) ' Obj:' sprintf('%10.4f',c(
4             iter)) ...
5             ' Vol:' sprintf('%6.3f',mean(xF(:))) ' Diff:' sprintf('%6.3f',
6                 ,diff)]]);
7     elseif dis == 2     % display parts of iterations
8         if iter == 1 || iter == disiter
9             if iter == 1
10                disiter = plotiter;
11            elseif iter == disiter
12                disiter = disiter + plotiter;
13            end
14            disp([' Iter:' sprintf('%4i',iter) ' Obj:' sprintf('%10.4f',c
15                (iter)) ...
16                ' Vol:' sprintf('%6.3f',mean(xF(:))) ' Diff:' sprintf('
17                    %6.3f',diff)]]);
18        end
19    end
20    if draw == 1         % plot iterations
21        figure(1)
22        subplot(2,1,1)
23        [nely,nelx,nelz] = size(xF);
24        hx = 1; hy = 1; hz = 1;           % User-defined unit element
25        size
26        face = [1 2 3 4; 2 6 7 3; 4 3 7 8; 1 5 8 4; 1 2 6 5; 5 6 7 8];
27        for k = 1:nelz
28            z = (k-1)*hz;
29            for i = 1:nelx
30                xplot = (i-1)*hx;
31                for j = 1:nely
32                    y = nely*hy - (j-1)*hy;
33                    if (xF(j,i,k) > 0.5) % User-defined display density
34                        threshold
35                        vert = [xplot y z; xplot y-hx z; xplot+hx y-hx z;
36                            xplot+hx y z; xplot y z+hx;xplot y-hx z+hx;
37                            xplot+hx y-hx z+hx;xplot+hx y z+hx];
38                        vert(:,[2 3]) = vert(:,[3 2]); vert(:,2,:) = -
39                            vert(:,2,:);
40                        patch('Faces',face,'Vertices',vert,'FaceColor',
41                            ,[0.2+0.8*(1-xF(j,i,k)),0.2+0.8*(1-xF(j,i,k))
42                                ,0.2+0.8*(1-xF(j,i,k))]);
43                    end
44                end
45            end
46        end
47    end

```

```

36         end
37     end
38     axis equal; axis tight;
39     set(gca, 'XTick', [], 'YTick', [], 'ZTick', [], 'XTicklabel', [], ...
40         'YTicklabel', [], 'ZTicklabel', [], 'xcolor', 'w', 'ycolor', 'w', '
            zcolor', 'w')
41     view([30,30]);
42     xlabel(sprintf('c = %.2f', c(iter)), 'Color', 'k')
43     drawnow;
44     hold on
45     if iter == 1
46         % Plot coloured dots for constraints
47         for i = 1:length(fix)
48             nplotx = ceil(fix(i)/(3*(ny+1)));
49             while nplotx > (nx+1)
50                 nplotx = nplotx -(nx+1);
51             end
52             npx(i) = nplotx-1;
53             nplot = ceil(fix(i)/3);
54             while nplot > (ny+1)
55                 nplot = nplot-(ny+1);
56             end
57             npy(i) = nplot-1;
58             npz(i) = 1-ceil(fix(i)/(3*(nx+1)*(ny+1)));
59         end
60         plot3(npx, npz, npy, 'r.', 'MarkerSize', 20)
61         % Plot coloured dots for force application
62         for i = 1:length(Fe)
63             nplotx = ceil(Fe(i)/(3*(ny+1)));
64             while nplotx > (nx+1)
65                 nplotx = nplotx -(nx+1);
66             end
67             npfx(i) = nplotx-1;
68             nplot = ceil(Fe(i)/3);
69             while nplot > (ny)
70                 nplot = nplot-(ny+1);
71             end
72             npfy(i) = nplot;
73             npfz(i) = 1-ceil(Fe(i)/(3*(nx+1)*(ny+1)));
74         end
75         plot3(npfx, npfz, npfy, 'g.', 'MarkerSize', 20)
76         drawnow;
77     end
78     % Plot compliance plot
79     figure(1)
80     subplot(2,1,2)
81     plot(c(1:iter))
82     xaxmax = c(iter);
83     yaxmax = max(c);
84     yaxmin = min(c(1:iter));
85     ylim([0.95*yaxmin yaxmax])
86     xlim([0 iter+10])
87     elseif draw == 2           % plot parts of iterations

```

```

88     if iter == 1 || iter == drawiter
89         if iter == 1
90             drawiter = plotiter;
91         elseif iter == drawiter
92             drawiter = drawiter + plotiter;
93         end
94         figure(1)
95         subplot(2,1,1)
96         [nely,nelx,nelz] = size(xF);
97         hx = 1; hy = 1; hz = 1;           % User-defined unit
98         element size
99         face = [1 2 3 4; 2 6 7 3; 4 3 7 8; 1 5 8 4; 1 2 6 5; 5 6 7
100              8];
101         for k = 1:nelz
102             z = (k-1)*hz;
103             for i = 1:nelx
104                 xplot = (i-1)*hx;
105                 for j = 1:nely
106                     y = nely*hy - (j-1)*hy;
107                     if (xF(j,i,k) > 0.5) % User-defined display
108                         density threshold
109                         vert = [xplot y z; xplot y-hx z; xplot+hx y-
110                                hx z; xplot+hx y z; xplot y z+hx;xplot y-
111                                hx z+hx; xplot+hx y-hx z+hx;xplot+hx y z+
112                                hx];
113                         vert(:,[2 3]) = vert(:,[3 2]); vert(:,2,:) =
114                             -vert(:,2,:);
115                         patch('Faces',face,'Vertices',vert,'FaceColor
116                               ',[0.2+0.8*(1-xF(j,i,k)),0.2+0.8*(1-xF(j,i
117                               ,k)),0.2+0.8*(1-xF(j,i,k))]);
118                         hold on;
119                     end
120                 end
121             end
122         end
123     end
124     axis equal; axis tight;
125     set(gca,'XTick',[],'YTick',[],'ZTick',[],'XTicklabel',[],...
126         'YTicklabel',[],'ZTicklabel',[],'xcolor','w','ycolor','w',
127         'zcolor','w');
128     view([30,30]);
129     xlabel(sprintf('c = %.2f',c(iter)),'Color','k');
130     drawnow;
131     hold on
132     if iter == 1
133         % Plot coloured dots for constraints
134         for i = 1:length(fix)
135             nplotx = ceil(fix(i)/(3*(ny+1)));
136             while nplotx > (nx+1)
137                 nplotx = nplotx -(nx+1);
138             end
139             npx(i) = nplotx-1;
140             nplot = ceil(fix(i)/3);
141             while nplot > (ny+1)

```

```

131         nplot = nplot-(ny+1);
132     end
133     npy(i) = nplot-1;
134     npz(i) = 1-ceil(fix(i)/(3*(nx+1)*(ny+1)));
135 end
136 plot3(npz,npz,npy,'r.','MarkerSize',20)
137 % Plot coloured dots for force application
138 for i = 1:length(Fe)
139     nplotx = ceil(Fe(i)/(3*(ny+1)));
140     while nplotx > (nx+1)
141         nplotx = nplotx -(nx+1);
142     end
143     npfx(i) = nplotx-1;
144     nplot = ceil(Fe(i)/3);
145     while nplot > (ny)
146         nplot = nplot-(ny+1);
147     end
148     npfy(i) = nplot;
149     npfz(i) = 1-ceil(Fe(i)/(3*(nx+1)*(ny+1)));
150 end
151 plot3(npfx,npfz,npfy,'g.','MarkerSize',20)
152 drawnow;
153 end
154 % Plot compliance plot
155 figure(1)
156 subplot(2,1,2)
157 plot(c(1:iter))
158 xaxmax = c(iter);
159 yaxmax = max(c);
160 yaxmin = min(c(1:iter));
161 ylim([0.95*yaxmin yaxmax])
162 xlim([0 iter+10])
163 end
164 end
165 end
166 %% ONLY DISPLAY FINAL RESULT
167 if dis == 0 || dis == 2 % display final result
168     disp([' Iter:' sprintf('%4i',iter) ' Obj:' sprintf('%10.4f',c(iter))
169         ...
170         ' Vol:' sprintf('%6.3f',mean(xF(:))) ' Diff:' sprintf('%6.3f',
171         diff)]);
172 end
173 if draw == 0 || draw == 2 % plot final result
174     figure(1)
175     subplot(2,1,1)
176     [nely,nelx,nelz] = size(xF);
177     hx = 1; hy = 1; hz = 1; % User-defined unit element size
178     face = [1 2 3 4; 2 6 7 3; 4 3 7 8; 1 5 8 4; 1 2 6 5; 5 6 7 8];
179     for k = 1:nelz
180         z = (k-1)*hz;
181         for i = 1:nelx
182             xplot = (i-1)*hx;
183             for j = 1:nely

```

```

182         y = nely*hy - (j-1)*hy;
183         if (xF(j,i,k) > 0.5) % User-defined display density
            threshold
184             vert = [xplot y z; xplot y-hx z; xplot+hx y-hx z;
                    xplot+hx y z; xplot y z+hx;xplot y-hx z+hx; xplot+
                    hx y-hx z+hx;xplot+hx y z+hx];
185             vert(:,[2 3]) = vert(:,[3 2]); vert(:,2,:) = -vert
                (:,2,:);
186             patch('Faces',face,'Vertices',vert,'FaceColor'
                    ,[0.2+0.8*(1-xF(j,i,k)),0.2+0.8*(1-xF(j,i,k))
                    ,0.2+0.8*(1-xF(j,i,k))]);
187             hold on;
188         end
189     end
190 end
191 end
192 axis equal; axis tight;
193 set(gca,'XTick',[],'YTick',[],'ZTick',[],'XTicklabel',[],...
194     'YTicklabel',[],'ZTicklabel',[],'xcolor','w','ycolor','w','zcolor
    ','w');
195 view([30,30]);
196 xlabel(sprintf('c = %.2f',c(iter)),'Color','k');
197 drawnow;
198 hold on
199 % Plot coloured dots for constraints
200 for i = 1:length(fix)
201     nplotx = ceil(fix(i)/(3*(ny+1)));
202     while nplotx > (nx+1)
203         nplotx = nplotx -(nx+1);
204     end
205     npx(i) = nplotx-1;
206     nplot = ceil(fix(i)/3);
207     while nplot > (ny+1)
208         nplot = nplot-(ny+1);
209     end
210     npy(i) = nplot-1;
211     npz(i) = 1-ceil(fix(i)/(3*(nx+1)*(ny+1)));
212 end
213 plot3(npx,npz,npy,'r.','MarkerSize',20)
214 % Plot coloured dots for force application
215 for i = 1:length(Fe)
216     nplotx = ceil(Fe(i)/(3*(ny+1)));
217     while nplotx > (nx+1)
218         nplotx = nplotx -(nx+1);
219     end
220     npfx(i) = nplotx-1;
221     nplot = ceil(Fe(i)/3);
222     while nplot > (ny)
223         nplot = nplot-(ny+1);
224     end
225     npfy(i) = nplot;
226     npfz(i) = 1-ceil(Fe(i)/(3*(nx+1)*(ny+1)));
227 end

```

```
228     plot3(npfx,npfz,npfy,'g.','MarkerSize',20)
229     % Plot compliance plot
230     figure(1)
231     subplot(2,1,2)
232     plot(c(1:iter))
233     xaxmax = c(iter);
234     yaxmax = max(c);
235     yaxmin = min(c(1:iter));
236     ylim([0.95*yaxmin yaxmax])
237     xlim([0 iter+10])
238 end
239 toc           % stop timer
```

C.8 Complaint Mechanisms Add-in.m

In this section, an add-in is made available to compute a variety of complaint mechanisms, as described in (3.4).

Using the previously described BASIC-code (B.3) as basis, the following lines should upgrade the code to calculate compliant mechanisms. At first, change the move limit for the Optimality Criteria, to allow for smaller steps in the optimization [replace line 30]:

```
1 move = 0.1; % move limit for lagrange [0.1]
```

Since compliance mechanisms often consist of a symmetric problem, a new symmetry-function is build in. Also, an option for plotting small displacement is implemented [between line 33-34]:

```
1 sym = 2; % symmetry [0 = off, 1 = x-axis, 2 = y-axis]
2 def = 1; % plot deformations [0 = off, 1 = on]
```

In compliance mechanisms it is helpful to describe and calculate a displacement, in stead of a force. Therefore a stiffness for the input and output load can be defined [between line 42-43]:

```
1 Kin = 5e-2; % spring stiffness at input force [5e-4]
2 Kout = 5e-4; % spring stiffness at output force [5e-4]
```

The compliant mechanism case as described in 3-11a can be created by changing the force and supports [replace line 44-48]:

```
1 %% DEFINE FORCE
2 Uin = 2*(ny+1)-1; % input force node
3 Uout = 2*(nx+1)*(ny+1)-1; % output force node
4 Fe = [Uin Uout]; % element of force application [Uin Uout]
5 Fn = [1 2]; % number of applied force locations [1 2]
6 Fv = [1 -1]; % value of applied force [1 -1]
7 %% DEFINE SUPPORTS
8 fix = [1:4 (Uin+1):2*(ny+1):(Uout+1)]; % fixed elements
```

In order to implement the stiffness at the input and output nodes, the predefined spring stiffness needs to be added to the existing stiffness matrix [between line 177-178]:

```

1      K(Uin,Uin) = K(Uin,Uin) + Kin; % add input spring stiffness
2      K(Uout,Uout) = K(Uout,Uout) + Kout; % add output spring stiffness

```

The adjoint load cases, as well as the new objective needs to be defined by replacing the original line [replace line 182-187]:

```

1      U1 = U(:,1); U2 = U(:,2);
2      c0 = reshape(sum((U1(dofmat)*Ke).*U2(dofmat),2),ny,nx);
3      c(iter) = U(Uout,1);
4      Sens = p*(E-Emin)*xF.^(p-1).*c0;

```

When using the Optimality Criteria method, the convergence criteria is changed [replace line 203-205]:

```

1      while (l2-l1)/(l1+l2) > 1e-4 && l2 > 1e-40; % start loop
2          lag = 0.5*(l1+l2); % average of lagrangian interval
3          xnew = max(0,max(x-move,min(1,min(x+move,x.*(max(1e-10,-Sens
                ./lag)).^0.3))))); % update element densities

```

With this compliant mechanisms, in order to compare the in- and output displacements, it could be helpful to display these displacement in the workspace [replace line 269-271]:

```

1  disp([' Iter:' sprintf('%4i',iter) ' Uin:' sprintf('%6.2f',U(Uin)) ...
2      ' Uout:' sprintf('%6.2f',c(iter)) ' Con:' sprintf('%6.2f',
        diff) ' Vol:' sprintf('%6.2f',mean(xF(:))) ' Diff:'
        sprintf('%6.3f',diff)]);

```

When symmetry case is implemented, this requires an additional figure, which displays the symmetric case scenario, using the live optimization [between line 316-317]:

```

1      if sym ~= 0          % apply symmetry
2          if sym == 1      % symmetry around x-axis
3              xFlip = fliplr(xF);
4              xFlipplot = [xFlip xF];
5          end
6          if sym == 2      % symmetry around y-axis
7              xFlip = flip(xF);
8              xFlipplot = [xF; xFlip];
9          end
10     colormap gray

```

```

11         imagesc(1-xFliplot)
12         axis equal
13         axis off
14     end

```

Also display the final results, when not using the display output [replace line 320-322]:

```

1 disp([' Iter:' sprintf('%4i',iter) ' Uin:' sprintf('%6.2f',U(Uin)) ...
2      ' Uout:' sprintf('%6.2f',c(iter)) ' Con:' sprintf('%6.2f',
      diff) ' Vol:' sprintf('%6.2f',mean(xF(:))) ' Diff:'
      sprintf('%6.3f',diff)]);

```

As defined before, also an implementation for the fast implementation, without live optimization needs to be implemented [between line 367-368]:

```

1     if sym ~= 0           % apply symmetry
2         if sym == 1      % symmetry around x-axis
3             xFlip = fliplr(xF);
4             xFliplot = [xFlip xF];
5         end
6         if sym == 2      % symmetry around y-axis
7             xFlip = flip(xF);
8             xFliplot = [xF; xFlip];
9         end
10        colormap gray
11        imagesc(1-xFliplot)
12        axis equal
13        axis off
14    end

```

In case displacement needs to be plotted, a small add-in to create a movie for different input displacement is made available [between line 368-369]:

```

1 %% PLOTTING DISPLACEMENT (COMPLIANT MECHANISMS)
2 figure(2)
3 xaxis = get(gca,'XLim');
4 yaxis = get(gca,'YLim');
5 if def == 1
6     figure(3)
7     clear mov
8     colormap(gray);
9     Umov = 1;           % start movie counter
10    Umax = 0.0025;      % define maximum displacement
11    for Udisp = linspace(0,Umax,10); % vary input displacement

```

```

12     clf
13     for ely = 1:ny      % plot displacements...
14         for elx = 1:nx % for each element...
15             if xF(ely,elx) > 0 % exclude white regions for plotting
16                 purposes
17                     n1 = (ny+1)*(elx-1)+ely;
18                     n2 = (ny+1)* elx +ely;
19                     Ue = -Udisp*U([2*n1-1;2*n1; 2*n2-1;2*n2; 2*n2+1;2*n2
20                         +2; 2*n1+1;2*n1+2],1);
21                     ly = ely-1; lx = elx-1;
22                     xx = [Ue(1,1)+lx Ue(3,1)+lx+1 Ue(5,1)+lx+1 Ue(7,1)+lx
23                         ]';
24                     yy = [-Ue(2,1)-ly -Ue(4,1)-ly -Ue(6,1)-ly-1 -Ue(8,1)-
25                         ly-1]';
26                     patch([xx xx],[yy+ny -yy-ny],[-xF(ely,elx) -xF(ely,
27                         elx)], 'LineStyle', 'none');
28                 end
29             end
30         end
31     end
32     xlim(xaxis)
33     ylim(yaxis-ny)
34     drawnow
35     mov(Umov) = getframe(3); % movie
36     Umov = Umov +1;      % update counter
37 end
38
39 toc % stop timer
40 movlip = flip(mov); % create symmetry
41 movull = [mov movlip]; % create symmetry
42 FileName = ['Compliant_',datestr(now, 'ddmm_HHMMSS'),'avi']; %
43     dynamic filename
44 movie2avi(movull, FileName, 'compression', 'None', 'FPS', 10); % save
45     video
46 end

```

C.9 Design of Supports Add-in.m

In this section, an add-in is made available to include design of supports, as described in (4.1). Using the previously described BASIC-code (B.3) as basis, the following lines should upgrade the code to include computation of the optimal support design. At first, change the pre-ambble to clear a big Z matrix, which stores the support design each iteration. [replace line 23]:

```
1      clc; clf; close all; clear X; clear Z; % clear workspace
```

Because the design of support costs a lot of computational time, it can be helpful to plot the iterations and graphics only partially by introducing the plotiter, which is a definition of the output steps of the iterations [replace line 34-36]:

```
1  plotiter = 5;           % gap of iterations used to plot or draw
    iterations [5]
2  %% DEFINE OUTPUT
3  draw = 1;              % plot iterations [0 = off, 1 = on, 2 = partial]
4  dis = 1;              % display iterations [0 = off, 1 = on, 2 =
    partial]
```

Design of support implementation require some additional input parameters. By implementing the following inputs. Additionally, the force as shown in Figure 4-4 needs to be changed. [replace line 43-44]:

```
1  %% DEFINE DESIGN OF SUPPORTS
2  supp = [1:ny (1:ny)+(nx-1)*ny ny:ny:nx*ny]; % support area [1:ny (1:ny)+(
    nx-1)*ny ny:ny:nx*ny]
3  supp = unique(supp); % create unique support area
4  zvol = 0.2;          % maximum support area [0.2]
5  cost = 1;           % set maximum cost of supports [1]
6  k0 = 0.01;          % spring stiffness for support stiffness
7  q = 5;              % penalty for support design [3]
8  zmin = 1e-9;        % minimum support design variable [1e-9]
9  dist = 0;           % cost distribution [0 = off, 1 = x-distributed,
    2 = y-distribution]
10 %% DEFINE FORCE
11 Fe = 2:2*(ny+1):2*(ny+1)*(nx+1); % element of force application [2:2*(ny
    +1):2*(ny+1)*(nx+1)]
```

To implement the case as shown in Figure 4-4, the fixed supports need to be changed. The same example includes a solid road at the upper size, so an element restriction needs to be defined. [replace line 47-52]:

```

1 %% DEFINE SUPPORTS
2 fix = [1:2 2*(ny+1)*nx+(1:2)]; % define fixed locations [1:2 2*(ny+1)*nx
   +(1:2)]
3 %% DEFINE ELEMENT RESTRICTIONS
4 shap = 2; % [0 = no restrictions, 1 = circle, 2 = custom]
5 area = 1; % [0 = no material (passive), 1 = material (
   active)]
6 nodr = 1:ny:nx*ny; % custom restricted nodes [1:ny:nx*ny]

```

To handle the element restriction, the following lines need to be inserted. [between line 104-105]:

```

1 elseif shap == 2 % custom restrictions
2     rest = zeros(ny*nx,1); % pre-allocate space
3     for i = 1:length(nodr) % write restriction
4         resti = nodr(i); % write restriction
5         rest(resti) = 1; % write restriction
6     end
7     rest = reshape(rest,ny,nx);
8     for i = 1:nx % start loop
9         for j = 1:ny % for each element
10            if rest(j,i) == area % check for restriction
11                x(j,i) = area; % store restrictions in material
                    distribution
12            end
13        end
14    end

```

The design of support implementation needs some more actions, which needs to be defined. Also, a distribution of costs can be defined over here. [between line 118-119]:

```

1 %% DESIGN OF SUPPORT DISTRIBUTION
2 xsiz = size(xFree(:),1); % size of design variables
3 zsiz = size(supp,2); % size of support design variables
4 xzer = zeros(xsiz,1); % empty row of zeros for mma usage
5 zzer = zeros(zsiz,1); % empty row of zeros for mma usage
6 z = zeros(ny,nx); % create design of support domain
7 z(supp) = zvol; % plugin initial support design variables
8 zval = z'; % create vector of design variables
9 Si = 1; % counter
10 if dist == 1 % x-axis cost distribution
11     Scos = linspace(1,cost,nx); % x-axis cost distribution
12     Scost = zeros(nx,nx); % create multiplication matrix
13     for i = 1:nx % create weighted cost matrix
14         Scost(Si,i) = Scos(i); % plug-in cost values
15         Si = Si+1; % update counter
16     end

```

```

17 elseif dist == 2           % y-axis cost distribution
18     Scos = linspace(1,cost,ny); % y-axis cost distribution
19     Scost = zeros(ny,ny); % create multiplication matrix
20     for i = 1:ny           % create weighted cost matrix
21         Scost(Si,i) = Scos(i); % plug-in cost values
22         Si = Si+1;         % update counter
23     end
24 end
25 Adofsup = dofmat(supp,:); % degrees of freedom for support locations
26 Asup = unique(Adofsup(:)); % unique support locations
27 zF = z;                   % set design of support
28 zval = zval(zval ~= 0); % create configurable design of support vector

```

The number of constraints is increased, by the introduction of the design of supports. Consequentially, the number of variables is increased, since the design space is enlarged. A minimum variable of the support density is introduced, to prevent the solution to lock in a local optimum. [replace line 120-122]:

```

1 m = 2;                       % number of constraint functions
2 n = xsiz+zsiz;               % number of variables
3 xmin = [zeros(xsiz,1);zmin*ones(zsiz,1)]; % minimum values of x

```

A small error is fixed, to include the number of constraint functions. [replace line 127]:

```

1 dfdx2 = zeros(m,n);         % second derivative of the constraint function

```

A pre-allocation step is required for plotting purposes, as explained further on. [between line 139-140]:

```

1 npdx = zeros(length(nodes),1)'; % pre-allocate force dots
2 npdy = zeros(length(nodes),1)'; % pre-allocate force dots

```

Design of supports require an additional calculation of the springs stiffnesses of the supports. This stiffness tensor needs to be calculated each loop and is added to the external stiffness tensor, resulting in a final stiffness matrix for that iteration. [replace line 178]:

```

1     Kfvec = zeros(2*(ny+1)*(nx+1),1); % build zeros support vector
2     for i = 1:length(supp) % for each support element...
3         dofsup = dofmat(supp(i),:); %...find the corresponding dof
4         for j = 1:length(dofsup) % calculate new stiffness vector

```

```

5         Kfvec(dofsup(j)) = Kfvec(dofsup(j))+(zF(supp(i))^q)*k0;
6     end
7 end
8 Kf = spdiags(Kfvec,0,2*(ny+1)*(nx+1),2*(ny+1)*(nx+1)); % create
    diagonal Kf
9 Kt = K+Kf; % update total force
10 U(free,:) = Kt(free,free)\Ftot(free,:); % displacement solving

```

The added constraint requires an additional sensitivity analysis. Also, the objective is changed, since it includes now the design of supports factor. By replacing the following lines the compliance and subsequent sensitivities are correctly calculated. [replace line 181-188]:

```

1     Senz = 0; % set constraint sensitivity to zero
2 %% Calculate compliance and sensitivity
3 for i = 1:size(Fn,2) % for number of load cases
4     Ui = U(:,i); % displacement per load case
5     c0 = reshape(sum((Ui(dofmat)*Ke).*Ui(dofmat),2),ny,nx); % initial
    compliance
6     cz0 = reshape(sum((Ui(dofmat)*k0).*Ui(dofmat),2),ny,nx); %
    initial support compliance
7     c(iter) = c(iter) + sum(sum((Emin+xF.^p*(E-Emin)).*c0)) + sum(sum
    ((zF.^q).*cz0)); % calculate compliance
8     Sens = Sens + reshape(2*Ui(dofmat)*repmat([0;-9.81*rho/4],4,1),ny
    ,nx) -p*(E-Emin)*xF.^(p-1).*c0; % sensitivity
9     Senz = Senz + -q*zF.^(q-1).*cz0; % calculate sensitivity to
    support variable
10 end
11 Senc = ones(ny,nx); % set constraint sensitivity
12 if dist == 0
13     Sencz = ones(ny,nx);
14 elseif dist == 1
15     Sencz =ones(ny,nx)*Scost; % set weighted cost constraint
    sensitivity
16 elseif dist == 2
17     Sencz = Scost*ones(ny,nx); % set weighted cost constraint
    sensitivity
18 end

```

The MMA solver as described in C.1 needs some changes. The design variable space is enlarged, by the introduction of the support design. The sensitivities of the support constraint is now added to the MMA solver. A cost distribution is used in the MMA-solver, to get the optimal result with respect to the cost function objective. [replace line 225-232]:

```

1     xval = [xFree(:);zval(:)]; % store current design variable for
    mma
2     if iter == 1 % for the first iteration...

```

```

3         cscale = 1/c(iter); % ...set scaling factor for mma solver
4     end
5     f0 = c(iter)*cscale; % objective at current design variable for
        mma
6     df0dx = [Sens(efree)*cscale; Senz(supp)']*cscale]; % store
        sensitivity for mma
7     if dist == 0 % no cost distribution
8         Scosts = zF; % cost-funcion no influence
9     elseif dist == 1 % x-axis cost distribution
10        Scosts = zF*Scost; % update weighted constraint function
11    elseif dist == 2 % y-axis cost distribution
12        Scosts = Scost*zF; % update weighted constraint function
13    end
14    f = [(sum(xF(:))/(vol*nx*ny)-1);(sum(Scosts(supp))/(zvol*size(
        supp,2))-1)]; % normalized constraint function
15    dfdx = [Senc(efree)'/((vol*ny*nx) zzer'; xzer' Sencz(supp)/(zvol*
        size(supp,2))]; % derivative of the constraint function

```

The output of the MMA solver is changed, so different commands are needed to get the right results. The output from the MMA solver is received into density and support design. [replace line 237-240]:

```

1     xold1 = [xFree(:);zval(:)]; % previous x, to monitor convergence
2     xnew = xF; % update result
3     xnew(efree) = xmma(1:xsiz); % include restricted elements
4     znew = zF; % update design result
5     znew(supp) = xmma(xsiz+1:end); % include mma solved supports
6     xnew = reshape(xnew,ny,nx); % reshape xmma vector to original
        size
7     znew = reshape(znew,ny,nx); % reshape support vector to original
        size

```

To enable the restriction, after the filtering step, as well as update the new design of support, requires an additional step, which is found here. [replace line 249-253]:

```

1     if shap == 1 || shap == 2 % if restrictions enableed
2         xF(rest==1) = area; % set restricted area
3     end
4     zF(:) = znew(:); % update support variables
5     zval = znew(supp); % update support variables
6     end

```

Substitute here the correct support design variables. [between line 256-257]:

```
1      z = znew;          % update support design variable
```

Store each support variable in a big Z-matrix for each iteration. [between line 265-266]:

```
1      Z(:,:,iter) = zF; % each support variable is stored for each
      iteration
```

Make sure the big Z-matrix is stored to the workspace. [between line 267-268]:

```
1      assignin('base', 'Z', Z); % each iteration (3rd dimension)
```

The introduced plotiter, needs some different output setting. The output is hold and out-putted each plotiter iteration. Also, for each display setting, the amount of support material is shown. [replace line 270-271]:

```
1      disp([' Iter:' sprintf('%4i',iter) ' Obj:' sprintf('%10.4f',c(
      iter)) ...
2          ' Vol:' sprintf('%6.3f',mean(xF(:))) ' Diff:' sprintf('%6.3f'
      ,diff) ' ZVol:' sprintf('%6.3f',mean(Scosts(supp)))]);
3  elseif dis == 2 % display parts of iterations
4  if iter == 1 || iter == disiter
5  if iter == 1
6  disiter = plotiter;
7  elseif iter == disiter
8  disiter = disiter + plotiter;
9  end
10 disp([' Iter:' sprintf('%4i',iter) ' Obj:' sprintf('%10.4f',c
      (iter)) ...
11      ' Vol:' sprintf('%6.3f',mean(xF(:))) ' Diff:' sprintf('
      %6.3f',diff) ' ZVol:' sprintf('%6.3f',mean(Scosts(supp)
      )))]);
12 end
```

The support design elements can be plotted using blue dots. Using a threshold value of 0.99 to determine whether or not to plot a support design element. [between line 304-305]:

```
1      % Plot coloured dots for design of supports
2      for i = 1:nx*ny
3          if zF(i) > 0.99 % treshold for plotting supports
4              if ceil(i/ny) == nx
5                  npdx(i) = ceil(i/ny) + 0.5;
```

```

6         elseif ceil(i/ny) == 1
7             npdx(i) = ceil(i/ny) - 0.5;
8         else
9             npdx(i) = ceil(i/ny);
10        end
11        nplot = i;
12        while nplot > ny
13            nplot = nplot-ny;
14        end
15        if nplot == ny
16            npdy(i) = nplot+0.5;
17        elseif nplot == 1
18            npdy(i) = nplot-0.5;
19        else
20            npdy(i) = nplot;
21        end
22    end
23 end
24 if exist('Dos(1)') %#ok<EXIST>
25     delete(Dos(1))
26 end
27 if exist('npdx') %#ok<EXIST>
28     Dos = plot(nonzeros(npdx),nonzeros(npdy),'b.','MarkerSize',
29             ,20);
30     clear npdx; clear npdy;
31     uistack(Dos,'bottom')
32 end

```

When enabling partial drawing, the following lines needs to be added into the code, to work around with this method. [between line 316-317]:

```

1     elseif draw == 2 % plot parts of iterations
2         if iter == 1 || iter == drawiter
3             if iter == 1
4                 drawiter = plotiter;
5             elseif iter == drawiter
6                 drawiter = drawiter + plotiter;
7             end
8             figure(1)
9             subplot(2,1,1)
10            colormap(gray); imagesc(1-xF);
11            set(gca,'XTick',[],'YTick',[],'XTicklabel',[],...
12                'YTicklabel',[],'xcolor','[0.7 0.7 0.7]','ycolor','[0.7
13                0.7 0.7]')
14            xlabel(sprintf('c = %.2f',c(iter)),'Color','k')
15            axis equal; axis tight
16            drawnow;
17            hold on
18            if iter == 1
19                % Plot coloured dots for force application

```

```

19         for i = 1:length(Fe)
20             npfx(i) = ceil(Fe(i)/(2*(ny+1))) - 0.5;
21             nplot = ceil(Fe(i)/2);
22             while nplot > (ny+1)
23                 nplot = nplot - (ny+1);
24             end
25             npfy(i) = nplot - 0.5;
26         end
27         plot(npfx, npfy, 'g.', 'MarkerSize', 20)
28         % Plot coloured dots for constraints
29         for i = 1:length(fix)
30             npx(i) = ceil(fix(i)/(2*(ny+1))) - 0.5;
31             nplot = ceil(fix(i)/2);
32             while nplot > (ny+1)
33                 nplot = nplot - (ny+1);
34             end
35             npy(i) = nplot - 0.5;
36         end
37         plot(npx, npy, 'r.', 'MarkerSize', 20)
38     end
39     % Plot coloured dots for design of supports
40     for i = 1:nx*ny
41         if zF(i) > 0.99 % threshold for plotting supports
42             if ceil(i/ny) == nx
43                 npdx(i) = ceil(i/ny) + 0.5;
44             elseif ceil(i/ny) == 1
45                 npdx(i) = ceil(i/ny) - 0.5;
46             else
47                 npdx(i) = ceil(i/ny);
48             end
49             nplot = i;
50             while nplot > ny
51                 nplot = nplot - ny;
52             end
53             if nplot == ny
54                 npdy(i) = nplot + 0.5;
55             elseif nplot == 1
56                 npdy(i) = nplot - 0.5;
57             else
58                 npdy(i) = nplot;
59             end
60         end
61     end
62     if exist('Dos(1)') %#ok<EXIST>
63         delete(Dos(1))
64     end
65     if exist('npdx') %#ok<EXIST>
66         Dos = plot(nonzeros(npdx), nonzeros(npdy), 'b.', 'MarkerSize', 20);
67         clear npdx; clear npdy;
68         uistack(Dos, 'bottom')
69     end
70     % Plot compliance plot

```

```

71         figure(1)
72         subplot(2,1,2)
73         plot(c(1:iter))
74         set(gca, 'YTick', [], 'YTicklabel', [])
75         xlabel('Iterations')
76         ylabel('Compliance')
77         xaxmax = c(iter);
78         yaxmax = max(c);
79         yaxmin = min(c(1:iter));
80         if pcon == 0
81             yaxmax = mean([yaxmin yaxmax]);
82         end
83         ylim([0.95*yaxmin yaxmax])
84         xlim([1 min(iter+10, miter)])
85     end

```

When disabling outputs, the design of support volume still needs to be displayed, at the end of the optimization process. [replace line 320-324]:

```

1  if dis == 0 || dis == 2 % display final result
2      disp([' Iter:' sprintf('%4i', iter) ' Obj:' sprintf('%10.4f', c(iter))
3          ...
4          ' Vol:' sprintf('%6.3f', mean(xF(:))) ' Diff:' sprintf('%6.3f',
5          diff) ' ZVol:' sprintf('%6.3f', mean(Scosts(supp))]);
6  end
7  if draw == 0 || draw == 2 % plot final result

```

When disabling outputs, the support design elements still can be plotted using blue dots. Using a threshold value to determine whether or not to plot a support design element. [between line 353-354]:

```

1      % Plot coloured dots for design of supports
2      for i = 1:nx*ny
3          if zF(i) > 0.99 % threshold for plotting supports
4              if ceil(i/ny) == nx
5                  npdx(i) = ceil(i/ny) + 0.5;
6              elseif ceil(i/ny) == 1
7                  npdx(i) = ceil(i/ny) - 0.5;
8              else
9                  npdx(i) = ceil(i/ny);
10             end
11             nplot = i;
12             while nplot > ny
13                 nplot = nplot - ny;
14             end
15             if nplot == ny
16                 npdy(i) = nplot + 0.5;

```

```
17         elseif nplot == 1
18             npdy(i) = nplot - 0.5;
19         else
20             npdy(i) = nplot;
21         end
22     end
23 end
24 if exist('Dos(1)') %#ok<EXIST>
25     delete(Dos(1))
26 end
27 if exist('npdx') %#ok<EXIST>
28     Dos = plot(nonzeros(npdx), nonzeros(npdy), 'b.', 'MarkerSize',
29             ,20);
29     clear npdx; clear npdy;
30     uistack(Dos, 'bottom')
31 end
```

C.10 Design of Actuator Placement Add-in.m

In this section, an add-in is made available to include design of forces, as described in (5). Using the previously described BASIC-code (B.3) as basis, the following lines should upgrade the code to include computation of the optimal placement of actuator design.

At first, in order to disable topology optimization, the volume can be fixed at a total void regime. [replace line 19]:

```
1      vol = 1;                % volume fraction [0-1]
```

Next, change the pre-amble to clear a big W matrix, which stores the force design for each iteration. [replace line 23]:

```
1      clc; clf; close all; clear X; clear W; % clear workspace
```

Now to enable finite difference check, and a way to enable or disable this finite difference check method. In the pre-amble a variable for this check can be build in. [between line 27-28]:

```
1  fincheck = 1;              % finite difference check [0 = off, 1 = on, 2 =
    break]
```

Because the design of actuator placement take a lot of computational time, it can be helpful to plot the iterations and graphics only partially by introducing the plotiter, which is a definition of the output steps of the iterations. Also, it is possible to create a deformed shape after the optimization. A threshold factor for plotting the forces, as well as a small perturbation value are implemented. [replace line 34-36]:

```
1  plotiter = 5;              % gap of iterations used to plot or draw
    iterations [5]
2  def = 0;                   % plot deformations [0 = off, 1 = on, 2 = play
    video]
3  wplot = 0.20;              % define treshold factor of Fmax for force plot
    [0.20]
4  h = 1e-6;                  % perturbation value for finite difference method
    [1e-6]
5  %% DEFINE OUTPUT
6  draw = 1;                  % plot iterations [0 = off, 1 = on, 2 = partial]
7  dis = 1;                   % display iterations [0 = off, 1 = on, 2 =
    partial]
```

The actual implementation of the design of actuator placement is made here. The minimal force constraint and maximum force per actuator can be defined here, as well as setting an area for the objective. [between line 48-49]:

```

1 %% DEFINE DESIGN OF ACTUATOR
2 Fmaxnode = 1;           % define max force per node [1]
3 Fmin = -1;             % minimal force constraint [1]
4 sen = 5;              % penalty for actuator design [5]
5 if abs(Fmaxnode) > abs(Fmin) % check for force model
6     Fmma = -Fmin;      % use Fmin as maximum xmma value
7 else
8     Fmin = Fmin/Fmaxnode; % use fraction for constraint function
9     Fmma = Fmaxnode;   % use maximum force per node as maximum xmma
                        value
10 end
11 Uarray = 1:2*(nx+1)*(ny+1); % define objective area

```

The initial distribution of the actuator lay-out is implemented here. Also, the MMA parameters are changed to handle with the force, since the force should be negative. [replace line 119-123]:

```

1 %% DESIGN OF ACTUATOR DISTRIBUTION
2 wsiz = size(Fe,2);     % size of actuator variables
3 wzer = zeros(wsiz,1); % empty row of zeros for mma usage
4 wF = F;               % plugin initial force distribution
5 wval = F(Fe);        % create vector of design variables
6 %% DEFINE MMA PARAMETERS
7 m = 1;               % number of constraint functions
8 n = wsiz;           % number of variables
9 xmin = -1*ones(n,1); % minimum values of x
10 xmax = -(1e-9/Fmma)*ones(wsiz,1); % maximum values of x

```

Additional pre-allocation of variables is needed, to speed up the optimization program. [between line 139-140]:

```

1 npdx = zeros(length(nodes),1)'; % pre-allocate force dots
2 npdy = zeros(length(nodes),1)'; % pre-allocate force dots
3 L = zeros(N,1);           % pre-allocate selection tensor
4 labda = zeros(N,1);      % pre-allocate lagrange multiplier
5 Fi = zeros(1,N);        % pre-allocate force selection vector
6 Cons = zeros(miter,1);   % pre-allocate constraint vector

```

To be able to make a selection of certain area, which needs to be optimized, a selection

vector can be defined. This vector makes it easy to switch between horizontal and vertical displacements. [between line 141-142]:

```

1 %% DEFINE SELECTION TENSOR
2 for j = Uarray           % for each iteration..
3     if mod(j,2) == 0     % ...check for horizontal or vertical
4         L(j) = 1;       % vertical selection value
5     else
6         L(j) = 1;       % horizontal selection value
7     end
8 end

```

To be able to penalize the force, a newly introduced penalty approach is made. The calculation of the continuation method should be changed, to include correct penalization of the force. [replace line 153-157]:

```

1         s = 0.5;        %... set penalty 0.5 for actuator design
2     elseif iter > piter % after a number of iterations...
3         p = min(pen,pcinc*p); % ... set continuation penalty
4         s = min(sen,1.06*s); % ... set continuation penalty actuator
           design
5     end
6     elseif pcon == 0    % not using continuation method
7         p = pen;        % set penalty
8         s = sen;        % set penalty actuator design

```

To actually calculate the penalization of the force, and subsequently scale the force to force the MMA solver to search the optimal value between 0 and 1, some adjustments should be made. [replace line 173]:

```

1     wP = atan(s*wF)/atan(s); % penalized actuator variable
2     Ftot = Fmma*(wP) + Fsw; % total force

```

Since the finite difference is built in inside the calculation loop, some pre-allocation steps are needed inside this loop. [between line 180-181]:

```

1     Senw = 0;           % set constraint sensitivity to zero
2     Cons(iter) = 0;     % set constraint to zero
3     Senc = ones(1,N);   % set constraint sensitivity

```

The actual finite difference check inside the loop needs to be included. Also, the objective is updated here to optimize towards minimum displacement. The associated sensitivities are calculated and filtered accordingly. [replace line 182-198]:

```

1     for i = 1:size(Fn,2) % for number of load cases
2         Ui = U(:,i); % displacement per load case
3         c0 = reshape(sum((Ui(dofmat)*Ke).*Ui(dofmat),2),ny,nx); % initial
           compliance
4         c(iter) = c(iter) - sum(sum(Ui)); % objective
5         labda(free) = -K(free,free)\L(free); % calculate lagrange
           multiplie
6         Fi(Fe) = (Fmma*s./((s^2*wF(Fe).^2+1)*(atan(s))));% force
           selection vector
7         FFi = spdiags(Fi',0, N,N); % force selection vector
8         Sens = Sens + FFi(Fe,Fe)*labda(Fe); % calculate sensitivity
9         Cons(iter) = Cons(iter) + Fmma*(Fmin/sum(sum(wF)))-1; % calculate
           constraint
10        dCdf = Senc(Fe)'*Fmma*full(Fmin)/-(sum(sum(full(wF))))^2; %
           constraint sensitivity
11        if iter == 2 % finite difference method
12            wF1 = wF; % store first force vector
13            [~,S1] = max(abs(Sens(:))); % calculate maximum sensitivity
           value
14            Sens1 = Sens(S1); % store maximum sensitivity value
15            [~,S2] = max(abs(dCdf(:))); % calculate maximum sensitivity
           value
16            Sens2 = dCdf(S2); % store maximum sensitivity value
17        end
18    end
19    if fil == 0 % optimality criterion with sensitivity filter
20        Sens(:) = Sens; % update filtered sensitivity
21        Sencw(:) = Senc; % update filtered sensitivity
22    elseif fil == 1 % optimality criterion with density filter
23        Sens(:) = Sens; % update filtered sensitivity of constraint
24        Sencw(:) = Senc; % update filtered sensitivity of constraint
25    elseif fil == 2 % optimality criterion with heaviside filter
26        dx = beta*exp(-beta*xTilde)+exp(-beta); % update hs parameter
27        Sens(:) = H*(Sens(:).*dx(:)./Hs); % update filtered sensitivity
28        Sencw(:) = Senc; % update filtered sensitivity of constraint
29    end

```

The MMA solver can here be adjusted to store the current force distribution as design variable. [replace line 225]:

```

1         xval = wval(:); % store current design variable for mma

```

Since the sensitivity and constraint values are calculated inside the loop, some adjustments

should be made inside the MMA loop. [replace line 230-232]:

```

1         df0dx = Sens*cyscale; % store sensitivity for mma
2         f = Cons(iter);      % normalized constraint function
3         dfdx = dCdf;        % derivative of normalized constraint
           function

```

The MMA solver should be updated to update the force design. [replace line 237-240]:

```

1         xold1 = wval(:); % previous x, to monitor convergence
2         xnew = xF;      % update density result
3         wnew = wF;      % update force result
4         wnew(Fe) = xmmma(1:end); % include mma result

```

The update of the force distribution is here built in. Also, an adjustment is made for the tolerance, to check the difference between force vectors. [replace line 253-256]:

```

1         wF(:) = wnew(:); % update force variables
2         wval = wnew(Fe); % update force variables
3         end
4         diff = max(abs(full(Fmma*wnew(:))-full(F(:)))); % difference of
           maximum element change
5         F = Fmma*wnew; % update design variable

```

The actual finite difference check is made after each loop. Here, the values are changed using a small perturbation. The pre-allocation of fincheck can be used to check, stop, or skip the finite difference method. [between line 262-263]:

```

1         %% Finite difference method
2         if (fincheck == 1 || fincheck == 2) % check for finite difference
           method
3             if iter == 2 % on first findif iteration
4                 wF = wF1; % store first findif result...
5                 wF(Fe(S1)) = wF1(Fe(S1))+h; %...and add a small pertubation
6             elseif iter == 3 % on second findif iteration
7                 findif = (c(3)-c(2))/h; % calculate finite difference method
8                 Sensdif = abs(max((findif-Sens1)/Sens1,(Sens1-findif)/findif)
           ); % maximum difference
9                 if Sensdif > 0.01 % when difference between sensitivity and
           findif is too much display
10                    disp(['Warning: Sensitivity needs to be checked, max
           difference:' sprintf('%10.2f',Sensdif)])
11                    if fincheck == 2 % when fincheck is not accomplished...

```

```

12         break %... break the loop and stop the code
13     end
14 end
15     wF = wF1; % store first findif result...
16     wF(Fe(S2)) = wF1(Fe(S2))+h; %...and add a small pertubation
17 elseif iter == 4 % on third findif iteration
18     findif2 = (Cons(4)-Cons(2))/h; % calculate finite difference
        method
19     Sensdif2 = abs(max((findif2-Sens2)/Sens2,(Sens2-findif2)/
        findif2)); % maximum difference
20     if Sensdif2 > 0.01 % when difference between sensitivity and
        findif is too much display
21         disp(['Warning: Sensitivity needs to be checked, max
            difference:' sprintf('%10.2f',Sensdif2)])
22         if fincheck == 2 % when fincheck is not accomplished...
23             break %... break the loop and stop the code
24         end
25     end
26 end
27 end

```

Make a separate value index, which stores each force distribution for each iteration. [between line 265-266]:

```

1     W(:,:,iter) = full(wF); % each force variable is stored for each
        iteration

```

Additionally, store this variable to the workspace. [between line 267-268]:

```

1     assignin('base', 'W', W); % each iteration (3rd dimension).

```

The introduced plotiter, needs some different output setting. The output is hold and out-putted each plotiter iteration. Also, for each display setting, the amount of total force is shown. [replace line 270-271]:

```

1     disp([' Iter:' sprintf('%4i',iter) ' Obj:' sprintf('%10.4f',c(
        iter)) ...
2         ' Ftot:' sprintf('%6.3f',sum(full(wP(:)))) ' Diff:' sprintf('
        %6.3f',diff)]);
3     elseif dis == 2 % display parts of iterations
4         if iter == 1 || iter == disiter
5             if iter == 1
6                 disiter = plotiter;
7             elseif iter == disiter

```

```

8         disiter = disiter + plotiter;
9     end
10    disp([' Iter:' sprintf('%4i',iter) ' Obj:' sprintf('%10.4f',c
        (iter)) ...
11         ' Ftot:' sprintf('%6.3f',sum(full(wP(:)))) ' Diff:'
        sprintf('%6.3f',diff)]);
12 end

```

The force distribution can be plotted by blue dots and attached arrows. Using a threshold value of *wplot* to determine whether or not to plot a force application. [replace line 277-304]:

```

1     set(gca, 'XTick', [], 'YTick', [], 'XTicklabel', [], ...
2         'YTicklabel', [], 'xcolor', '[0.7 0.7 0.7]', 'ycolor', '[0.7 0.7
3         0.7]');
4     xlabel(sprintf('c = %.2f',c(iter)), 'Color', 'k')
5     axis equal; axis tight
6     drawnow;
7     hold on
8     if iter == 1
9         % Plot coloured dots for constraints
10        for i = 1:length(fix)
11            npx(i) = ceil(fix(i)/(2*(ny+1))) - 0.5;
12            nplot = ceil(fix(i)/2);
13            while nplot > (ny+1)
14                nplot = nplot - (ny+1);
15            end
16            npy(i) = nplot - 0.5;
17        end
18        plot(npx, npy, 'r.', 'MarkerSize', 20)
19    end
20    % Plot coloured dots for force application
21    Fmaxplot = min(min(full(F)));
22    for i = 1:length(Fe)
23        if F(Fe(i)) < wplot * Fmaxplot
24            npfx(i) = ceil(Fe(i)/(2*(ny+1))) - 0.5;
25            nplot = ceil(Fe(i)/2);
26            while nplot > (ny+1)
27                nplot = nplot - (ny+1);
28            end
29            npfy(i) = nplot - 0.5;
30        end
31    end
32    if iter > 1
33        delete(Dof)
34    end
35    if exist('npfx', 'var')
36        Dof = plot(npfx(npfx(:) > 0), npfy(npfy(:) > 0), 'b.', 'MarkerSize',
37            20);
38        clear npfx; clear npfy;

```

```

37         uistack(Dof, 'top')
38     end
39     % Plot coloured arrows for force application
40     if (((diff < tol) && iter >= piter+1) || iter >= miter)
41         for i = 1:length(Fe)
42             npfx(i) = ceil(Fe(i)/(2*(ny+1)))-0.5;
43             nplot = ceil(Fe(i)/2);
44             while nplot > (ny+1)
45                 nplot = nplot-(ny+1);
46             end
47             npfy(i) = nplot-0.5;
48         end
49         for i = 1:length(Fe)
50             if F(Fe(i)) < wplot*Fmaxplot
51                 headsize = 1/sqrt(length(nonzeros(F(Fe)<0.5*Fmaxplot)
52                 ));
53                 if mod(Fe(i),2)
54                     arrowz([npfx(i) npfy(i)], [npfx(i)+0.5*ny*F(Fe(i))
55                     /Fmaxplot npfy(i)], headsize, 2, [0 0 1])
56                 else
57                     arrowz([npfx(i) npfy(i)], [npfx(i) npfy(i)+0.5*ny*
58                     F(Fe(i))/Fmaxplot], headsize, 2, [0 0 1])
59                 end
60             end
61         end
62     end
63 end

```

When enabling partial drawing, the following lines needs to be added into the code, to work around with this method. [between line 316-317]:

```

1     elseif draw == 2 % plot parts of iterations
2         if iter == 1 || iter == drawiter
3             if iter == 1
4                 drawiter = plotiter;
5             elseif iter == drawiter
6                 drawiter = drawiter + plotiter;
7             end
8             figure(1)
9             subplot(2,1,1)
10            colormap(gray); imagesc(1-xF);
11            set(gca, 'XTick', [], 'YTick', [], 'XTicklabel', [], ...
12                'YTicklabel', [], 'xcolor', '[0.7 0.7 0.7]', 'ycolor', '[0.7
13                0.7 0.7]')
14            xlabel(sprintf('c = %.2f', c(iter)), 'Color', 'k')
15            axis equal; axis tight
16            drawnow;
17            hold on
18            if iter == 1
19                % Plot coloured dots for constraints
20                for i = 1:length(fix)

```

```

20         npx(i) = ceil(fix(i)/(2*(ny+1))) - 0.5;
21         nplot = ceil(fix(i)/2);
22         while nplot > (ny+1)
23             nplot = nplot - (ny+1);
24         end
25         npy(i) = nplot - 0.5;
26     end
27     plot(npx, npy, 'r.', 'MarkerSize', 20)
28 end
29 % Plot coloured dots for force application
30 Fmaxplot = min(min(full(F)));
31 for i = 1:length(Fe)
32     if F(Fe(i)) < wplot*Fmaxplot
33         npx(i) = ceil(Fe(i)/(2*(ny+1))) - 0.5;
34         nplot = ceil(Fe(i)/2);
35         while nplot > (ny+1)
36             nplot = nplot - (ny+1);
37         end
38         npfy(i) = nplot - 0.5;
39     end
40 end
41 if iter > 1
42     delete(Dof)
43 end
44 if exist('npx', 'var')
45     Dof = plot(npx(npx(:) > 0), npfy(npfy(:) > 0), 'b.', '
46             MarkerSize', 20);
47     clear npx; clear npfy;
48     uistack(Dof, 'top')
49 end
50 % Plot coloured arrows for force application
51 if (((diff < tol) && iter >= piter+1) || iter >= miter)
52     for i = 1:length(Fe)
53         npx(i) = ceil(Fe(i)/(2*(ny+1))) - 0.5;
54         nplot = ceil(Fe(i)/2);
55         while nplot > (ny+1)
56             nplot = nplot - (ny+1);
57         end
58         npfy(i) = nplot - 0.5;
59     end
60     for i = 1:length(Fe)
61         if F(Fe(i)) < wplot*Fmaxplot
62             headsize = 1/sqrt(length(nonzeros(F(Fe) < 0.5*
63             Fmaxplot)));
64             if mod(Fe(i), 2)
65                 arrowz([npx(i) npfy(i)], [npx(i)+0.5*ny*F(Fe
66                 (i))/Fmaxplot npfy(i)], headsize, 2, [0 0 1])
67             else
68                 arrowz([npx(i) npfy(i)], [npx(i) npfy(i)
69                 +0.5*ny*F(Fe(i))/Fmaxplot], headsize, 2, [0 0
70                 1])
71             end
72         end
73     end
74 end

```

```

68         end
69     end
70     % Plot compliance plot
71     figure(1)
72     subplot(2,1,2)
73     plot(c(1:iter))
74     set(gca, 'YTick', [], 'YTicklabel', [])
75     xlabel('Iterations')
76     ylabel('Compliance')
77     xaxmax = c(iter);
78     yaxmax = max(c);
79     yaxmin = min(c(1:iter));
80     if pcon == 0
81         yaxmax = mean([yaxmin yaxmax]);
82     end
83     ylim([0.95*yaxmin yaxmax])
84     xlim([1 min(iter+10, miter)])
85 end

```

When disabling outputs, the design of force placement still needs to be displayed, at the end of the optimization process. [replace line 320-353]:

```

1 %% ONLY DISPLAY FINAL RESULT
2 if dis == 0 || dis == 2 % display final result
3     disp([' Iter:' sprintf('%4i', iter) ' Obj:' sprintf('%10.4f', c(iter))
4         ...
5         ' Ftot:' sprintf('%6.3f', sum(full(wP(:)))) ' Diff:' sprintf('%6.3
6         f', diff)]);
7 end
8 if draw == 0 || draw == 2 % plot final result
9     figure(1)
10    subplot(2,1,1)
11    colormap(gray); imagesc(1-xF);
12    axis equal; axis tight;
13    set(gca, 'XTick', [], 'YTick', [], 'XTicklabel', [], ...
14        'YTicklabel', [], 'xcolor', '0.7 0.7 0.7', 'ycolor', '0.7 0.7 0.7'
15        )
16    xlabel(sprintf('c = %.2f', c(iter)), 'Color', 'k')
17    drawnow;
18    hold on
19    % Plot coloured dots for constraints
20    for i = 1:length(fix)
21        npx(i) = ceil(fix(i)/(2*(ny+1))) - 0.5;
22        nplot = ceil(fix(i)/2);
23        while nplot > (ny+1)
24            nplot = nplot - (ny+1);
25        end
26        npy(i) = nplot - 0.5;
27    end
28    plot(npx, npy, 'r.', 'MarkerSize', 20)

```

```

26 % Plot coloured dots for force application
27 Fmaxplot = min(min(full(F)));
28 for i = 1:length(Fe)
29     if F(Fe(i)) < wplot*Fmaxplot
30         npfx(i) = ceil(Fe(i)/(2*(ny+1)))-0.5;
31         nplot = ceil(Fe(i)/2);
32         while nplot > (ny+1)
33             nplot = nplot-(ny+1);
34         end
35         npfy(i) = nplot-0.5;
36     end
37 end
38 if iter > 1
39     delete(Dof)
40 end
41 if exist('npfx','var')
42     Dof = plot(npfx(npfx(:)>0),npfy(npfy(:)>0),'b.','MarkerSize',20);
43     clear npfx; clear npfy;
44     uistack(Dof,'top')
45 end
46 % Plot coloured arrows for force application
47 if (((diff < tol) && iter >= piter+1) || iter >= miter)
48     for i = 1:length(Fe)
49         npfx(i) = ceil(Fe(i)/(2*(ny+1)))-0.5;
50         nplot = ceil(Fe(i)/2);
51         while nplot > (ny+1)
52             nplot = nplot-(ny+1);
53         end
54         npfy(i) = nplot-0.5;
55     end
56     for i = 1:length(Fe)
57         if F(Fe(i)) < wplot*Fmaxplot
58             headsize = 1/sqrt(length(nonzeros(F(Fe)<0.5*Fmaxplot)));
59             if mod(Fe(i),2)
60                 arrowz([npfx(i) npfy(i)],[npfx(i)+0.5*ny*F(Fe(i))/
61                     Fmaxplot npfy(i)],headsize,2,[0 0 1])
62             else
63                 arrowz([npfx(i) npfy(i)],[npfx(i) npfy(i)+0.5*ny*F(Fe
64                     (i))/Fmaxplot],headsize,2,[0 0 1])
65             end
66         end
67     end
68 end

```

To show deformed shape of the structure, please add-in the following lines. [between line 368-369]:

```

1 %% PLOTTING DISPLACEMENT
2 if (def == 1 || def == 2)
3     FileName = ['Displacement_',datestr(now, 'ddmm_HHMMSS'),'avi']; %

```

```

    dynamic filename
4   vidObj = VideoWriter(FileName);
5   vidObj.FrameRate = 3;
6   figure(1)
7   subplot(2,1,1)
8   xaxis = get(gca, 'XLim');
9   yaxis = get(gca, 'YLim');
10  open(vidObj);
11  figure(2)
12  clear mov
13  colormap(gray);
14  Umov = 1;                % start movie counter
15  Uim = zeros(5642,1);
16  Uim(2:2:end) = Ui(2:2:end);
17  Uim(1:2:end) = -Ui(1:2:end);
18  Umax = -10/max(abs(Uim)); % define maximum displacement
19  steps = 1;              % number of displacement steps
20  set(gca, 'nextplot', 'replacechildren');
21  Upatch = zeros(nx*ny,1);
22  for i = 1:ny*nx
23      Uindex = 2*(i+floor((i-1)/ny))-1+[1 2 2*(ny+1)+1 2*(ny+1)+3];
24      Upatch(i,1) = mean(U(Uindex));
25  end
26  Upatch = reshape(Upatch,ny,nx);
27  Upatchmin = min(min(Upatch));
28  Upatchnorm = -Upatch/Upatchmin;
29  for Udisp = linspace(Umax/steps,Umax,steps) % vary input displacement
30      clf
31      for ely = 1:ny % plot displacements...
32          for elx = 1:nx % for each element...
33              if xF(ely,elx) > 0 % exclude white regions for plotting
34                  purposes
35                      n1 = (ny+1)*(elx-1)+ely;
36                      n2 = (ny+1)* elx +ely;
37                      Ue = Udisp*Uim([2*n1-1;2*n1; 2*n2-1;2*n2; 2*n2+1;2*n2
38                          +2; 2*n1+1;2*n1+2],1);
39                      ly = ely-1; lx = elx-1;
40                      xx = [Ue(1,1)+lx Ue(3,1)+lx+1 Ue(5,1)+lx+1 Ue(7,1)+lx
41                          ]';
42                      yy = [-Ue(2,1)-ly -Ue(4,1)-ly -Ue(6,1)-ly-1 -Ue(8,1)-
43                          ly-1]';
44                      patch([xx xx],[yy yy],[Upatchnorm(ely,elx) Upatchnorm
45                          (ely,elx)], 'LineStyle', 'none');
46              end
47          end
48      end
49      colormap jet % for better interpolation...
50      axis tight
51      axis equal
52      xticks([0 15 30 45 60 75 90])
53      box on
54      colorbar

```

```

51         drawnow                % ...draw coloured densities
52         currFrame = getframe; % get current frame...
53         writeVideo(vidObj,currFrame); % ... write to video file
54     end
55     close(vidObj);
56 end
57 if def == 2                % when def equals 2...
58     implay(FileName)        % ...open Matlab Movie Player
59 end
60 toc

```

C.11 Topology Add-in.m

In this section, an add-in is made available to include topology, to work with design of actuator placement, as described in (5.4). Using the previously described Actuator Placement-code (C.10) as basis, the following lines should upgrade the code to include computation of the optimal placement of actuator design. The implementation of topology optimization will result in a much longer computational time.

First, the volume constraint can now be used, so a change to the volume should be made. [replace line 19]:

```

1     vol = 0.2;                % volume fraction [0-1]

```

Since the topology is now included, the sizes should be calculated and included. [replace line 149-150]:

```

1 %% DESIGN OF ACTUATOR AND TOPOLOGY DISTRIBUTION
2 xsiz = size(xFree,1);      % size of topology variables
3 wsiz = size(Fe,2);        % size of actuator variables
4 xzer = zeros(xsiz,1);     % empty row of zeros for mma usage

```

The number of constraints should be updates, and so does the size of the number of variables. [replace line 155-158]:

```

1 m = 3;                    % number of constraint functions
2 n = xsiz+wsiz;           % number of variables
3 xmin = [1e-9*ones(xsiz,1); -1*ones(wsiz,1)]; % minimum values of x
4 xmax = [ones(xsiz,1); -(1e-9/Fmma)*ones(wsiz,1)]; % maximum values of x

```

The topology optimization add-in results in an additional number of calculations inside the loop, so extra allocation steps are needed. [replace line 181-182]:

```

1 labda2 = zeros(N,1);      % pre-allocate second lagrange multiplier
2 Fi = zeros(1,N);        % pre-allocate force selection vector
3 Ua = zeros(N,1);        % pre-allocate displacement vector
4 Cons = zeros(miter,1);  % pre-allocate constraint vector
5 Cons2 = zeros(miter,1); % pre-allocate constraint #2 vector
6 Cons3 = zeros(miter,1); % pre-allocate constraint #3 vector

```

In this case, only the vertical selection is included, so no need for horizontal. [replace line 188]:

```

1          L(j) = 0;          % horizontal selection value

```

The change of design variables will result in a huge amount of changes inside the loop. Additional sensitivities needs to be calculated. An additional compliance constraint is added, in order to create physically possible structures. Also, the finite difference method is extended for all sensitivities. [replace line 240-253]:

```

1          Ua(Uarray) = Ui(Uarray); % selection of displacement
2          c0 = reshape(sum((Ui(dofmat)*Ke).*Ui(dofmat),2),ny,nx); % initial
           compliance
3          c(iter) = c(iter) + sum(Ua.^2); % objective
4          labda(free) = -sparse(Kt(free,free))\sparse(2*Ua(free)); %
           calculate lagrange multiplier
5          labda2(free) = 2*Ua(free); % calculate second lagrange multiplier
6          c00 = reshape(sum((labda(dofmat)*Ke).*Ui(dofmat),2),ny,nx); %
           initial labda compliance
7          Fi(Fe) = (Fmma*s./((s.^2*wF(Fe).^2+1)*(atan(s)))); % force
           selection vector
8          FFi = spdiags(Fi',0, N,N); % force selection vector
9          Sens = Sens + p*(E-Emin)*xF.^(p-1).*c00; % calculate density
           sensitivity
10         Senw = Senw - FFi(Fe,Fe)*labda(Fe); % calculate force sensitivity
11         Cons(iter) = Cons(iter) + 10*(sum(xF(:))/(vol*nx*ny)-1); %
           calculate constraint
12         dCdx = 10*Senc(efree)/(vol*ny*nx); % constraint sensitivity
13         Cons2(iter) = Cons2(iter) + 10*(Fmin/sum(sum(wF)))-1; % calculate
           constraint
14         dCdf = 10*Senc(Fe)*Fmin/-(sum(sum(full(wF))))^2; % constraint
           sensitivity
15         Cons3(iter) = Cons3(iter) + (sum(sum((Emin+xF.^p*(E-Emin)).*c0))
           -50); % compliance constraint
16         dCCdx = -p*(E-Emin)*xF.^(p-1).*c0; % constraint sensitivity

```

```

17     dCCdf = labda2(Fe)'*FFi(Fe,Fe); % constraint sensitivity
18     if iter == 2 % finite difference method
19         F1 = wF; % store force vector
20         X1 = xF; % store density vector
21         [~,S1] = max(abs(Sens(:))); % calculate maximum sensitivity
           value
22         Sens1 = Sens(S1); % store maximum sensitivity value
23         [~,S2] = max(abs(Senw(:))); % calculate maximum sensitivity
           value
24         Sens2 = Senw(S2); % store maximum sensitivity value
25         [~,S3] = max(abs(dCdx(:))); % calculate maximum sensitivity
           value
26         Sens3 = dCdx(S3); % store maximum sensitivity value
27         [~,S4] = max(abs(dCdf(:))); % calculate maximum sensitivity
           value
28         Sens4 = dCdf(S4); % store maximum sensitivity value
29         [~,S5] = max(abs(dCCdx(:))); % calculate maximum sensitivity
           value
30         Sens5 = dCCdx(S5); % store maximum sensitivity value
31         [~,S6] = max(abs(dCCdf(:))); % calculate maximum sensitivity
           value
32         Sens6 = dCCdf(S6); % store maximum sensitivity value

```

The MMA solver should work with more design variables. [replace line 294]:

```

1     xval = [xFree(:);wval(:)]; % store current design variable for
           mma

```

The MMA solver is here updated to extend the number of constraints and additional sensitivities. [replace line 299-301]:

```

1     df0dx = [Sens(efree);Senw]*cscale; % store sensitivity for mma
2     f = [Cons(iter);Cons2(iter);Cons3(iter)]; % normalized constraint
           function
3     dfdx =[dCdx wzer'; xzer' dCdf; dCCdx(efree)' dCCdf]; % derivative
           constraint functions

```

The previous design variable is here updated to include topology design. [replace line 306]:

```

1     xold1 = [xFree(:);wval(:)]; % previous x, to monitor convergence

```

The MMA result is split to update the topology and actuator placement. [replace line 309]:

```
1      xnew(efree) = xmma(1:xsiz); % update mma to density
2      wnew(Fe) = xmma(xsiz+1:end); % update mma to force
```

The density design variables should be updated. [between line 321-322]:

```
1      xFree = xnew(efree); % update density variable
```

The tolerance is updated, as a summation of changes in force and changes in density. [replace line 324]:

```
1      diff = (max(abs(full(Fmma*wnew(:))-full(F(:))))+max(abs(xnew(:)-x(:))
2      x = xnew; % update design variable density
```

The finite difference method checks six different sensitivities. This results in an extension of the code. [replace line 335-354]:

```
1      xF = X1; % store first findif result...
2      xF(S1) = X1(S1)+h; %...and add a small pertubation
3      wF = F1; % store first findif result
4      elseif iter == 3 % on second findif iteration
5          findif = (c(3)-c(2))/h; % calculate finite difference method
6          Sensdif = abs(max((findif-Sens1)/Sens1,(Sens1-findif)/findif
7          ); % maximum difference
8          if Sensdif > 0.01 % when difference between sensitivity and
9          findif is too much display
10             disp(['Warning: Sensitivity needs to be checked, max
11             difference:' sprintf('%10.2f',Sensdif)])
12             if fincheck == 2 % when fincheck is not accomplished...
13                 break %... break the loop and stop the code
14             end
15         end
16         wF = F1; % store first findif result...
17         wF(Fe(S2)) = F1(Fe(S2))+h; %...and add a small pertubation
18         xF = X1; % store first findif result
19         elseif iter == 4 % on third findif iteration
20             findif2 = (c(4)-c(2))/h; % calculate finite difference method
21             Sensdif2 = abs(max((findif2-Sens2)/Sens2,(Sens2-findif2)/
22             findif2)); % maximum difference
23             if Sensdif2 > 0.01 % when difference between sensitivity and
24             findif is too much display
```

```

20         disp(['Warning: Sensitivity needs to be checked, max
                difference:' sprintf('%10.2f',Sensdif2)])
21         if fincheck == 2 % when fincheck is not accomplished...
22             break %... break the loop and stop the code
23         end
24     end
25     wF = F1; % store first findif result
26     xF = X1; % store first findif result...
27     xF(S3) = xF(S3)+h; %...and add a small pertubation
28 elseif iter == 5 % on fourth findif iteration
29     findif3 = (Cons(5)-Cons(2))/h; % calculate finite difference
                method
30     Sensdif3 = abs(max((findif3-Sens3)/Sens3,(Sens3-findif3)/
                findif3)); % maximum difference
31     if Sensdif3 > 0.01 % when difference between sensitivity and
                findif is too much display
32         disp(['Warning: Sensitivity needs to be checked, max
                difference:' sprintf('%10.2f',Sensdif3)])
33         if fincheck == 2 % when fincheck is not accomplished...
34             break %... break the loop and stop the code
35         end
36     end
37     wF = F1; % store first findif result
38     wF(S4) = wF(S4)+h; % store first findif result...
39     xF = X1; %...and add a small pertubation
40 elseif iter == 6 % on fifth findif iteration
41     findif4 = (Cons2(6)-Cons2(2))/h; % calculate finite
                difference method
42     Sensdif4 = abs(max((findif4-Sens4)/Sens4,(Sens4-findif4)/
                findif4)); % maximum difference
43     if Sensdif4 > 0.01 % when difference between sensitivity and
                findif is too much display
44         disp(['Warning: Sensitivity needs to be checked, max
                difference:' sprintf('%10.2f',Sensdif4)])
45         if fincheck == 2 % when fincheck is not accomplished...
46             break %... break the loop and stop the code
47         end
48     end
49     wF = F1; % store first findif result
50     xF = X1; % store first findif result...
51     xF(S5) = xF(S5)+h; %...and add a small pertubation
52 elseif iter == 7 % on sixth findif iteration
53     findif5 = (Cons3(7)-Cons3(2))/h; % calculate finite
                difference method
54     Sensdif5 = abs(max((findif5-Sens5)/Sens5,(Sens5-findif5)/
                findif5)); % maximum difference
55     if Sensdif5 > 0.01 % when difference between sensitivity and
                findif is too much display
56         disp(['Warning: Sensitivity needs to be checked, max
                difference:' sprintf('%10.2f',Sensdif5)])
57         if fincheck == 2 % when fincheck is not accomplished...
58             break %... break the loop and stop the code
59     end

```

```

60         end
61         wF = F1;      % store first findif result...
62         wF(Fe(S6)) = F1(Fe(S6))+h; %...and add a small pertubation
63         xF = X1;      % store first findif result
64     elseif iter == 8 % on second finidif iteration
65         findif6 = (Cons3(8)-Cons3(2))/h; % calculate finite
           difference method
66         Sensdif6 = abs(max((findif6-Sens6)/Sens6,(Sens6-findif6)/
           findif6)); % maximum difference
67         if Sensdif6 > 0.01 % when difference between sensitivity and
           findif is too much display
68             disp(['Warning: Sensitivity needs to be checked, max
           difference:' sprintf('%10.2f',Sensdif6)])
69             if fincheck == 2 % when fincheck is not accomplished...
70                 break %... break the loop and stop the code

```

An update is made, to include topology in the output window. [replace line 369]:

```

1         ' Vol:' sprintf('%6.3f',mean(xF(:))) ' Ftot:' sprintf('%6.3f',
           , ...
2         sum(full(F))) ' Diff:' sprintf('%6.3f',diff)];

```

An update is made, to include topology in the output window. [replace line 378]:

```

1         ' Vol:' sprintf('%6.3f',mean(xF(:))) ' Ftot:' sprintf('%6.3f',
           , ...
2         sum(full(F))) ' Diff:' sprintf('%6.3f',diff)];

```

An update is made, to include topology in the output window. [replace line 549]:

```

1         ' Vol:' sprintf('%6.3f',mean(xF(:))) ' Ftot:' sprintf('%6.3f',
           , ...
2         sum(full(F))) ' Diff:' sprintf('%6.3f',diff)];

```

An update is made, to include topology in the output window. [replace line 549]:

```

1         ' Vol:' sprintf('%6.3f',mean(xF(:))) ' Ftot:' sprintf('%6.3f',
           , ...
2         sum(full(F))) ' Diff:' sprintf('%6.3f',diff)];

```

Appendix D

Supplementary Codes

In this section some supplementary MATLAB codes can be found. The prescribed MMA solution (C.1) method calls two external functions, in order to calculate the optimal solution. ~~These functions can be found in (D.1) and (D.2).~~ A function to create arrows can be found in (D.3).

D.3 Arrowz.m

In order to be able to plot force application as an arrow representation, a new code is written. This code can be used to draw a certain arrow from start- to endpoint, with an adjustable shaft- and headsize. Also, the color of these arrows can be adjusted (Broxterman, 2016).

```

1 function arrowz(startpair,endpair,varargin)
2 % Written in Sep 2016 by Stefan Broxterman (TU Delft)
3 %
4 % ARROWZ draws an easily adjustable arrow from startpair to endpair.
   These
5 % pairs should be vectors of length 2. The input of ARROWZ can vary from
   2
6 % to 6 inputs.
7 %
8 % ARROWZ(starpair,endpair) creates an easy arrow, a line plot from
9 % startpair to endpair, with an additional head within the direction of
   the
10 % endpair. Input format is [x y],[x y].
11 %
12 % ARROWZ(starpair,endpair,headsize) is able to adjust the size of the
   head.
13 % Default size is 1.
14 %
15 % ARROWZ(starpair,endpair,headsize,shaftsize) sets the thickness of the
16 % shaft to the desired size. Default size is 1.
17 %
18 % ARROWZ(starpair,endpair,headsize,shaftsize,color) specifies the color
   of
19 % the total arrow. These values should be provided as RGB. Default is
   black
20 % [0 0 0].
21 %
22 % ARROWZ(starpair,endpair,headsize,shaftsize,headcolor,shaftcolor) colors
23 % the shaft of the arrow into a seperate color. Default is black [0 0 0].
24 %
25 % Many thanks to Ryan Molecke
26 switch nargin % Check number of inputs
27     case 2
28         headsize = 1;
29         shaftsize = 1;
30         headcolor = [0 0 0];
31         shaftcolor = [0 0 0];
32     case 3
33         headsize = varargin{1};
34         shaftsize = 1;
35         headcolor = [0 0 0];
36         shaftcolor = [0 0 0];
37     case 4
38         headsize = varargin{1};
39         shaftsize = varargin{2};
40         headcolor = [0 0 0];
41         shaftcolor = [0 0 0];

```

```
42     case 5
43         headsize = varargin{1};
44         shaftsize = varargin{2};
45         headcolor = varargin{3};
46         shaftcolor = varargin{3};
47     case 6
48         headsize = varargin{1};
49         shaftsize = varargin{2};
50         headcolor = varargin{3};
51         shaftcolor = varargin{4};
52 end
53 % Begin drawing
54 v1 = headsize*(startpair-endpair)/2.5; % Create drawing vector
55
56 alfa = pi/8; % 45*pi/360
57 R = [cos(alfa) -sin(alfa) ; sin(alfa) cos(alfa)]; % Rotational matrix
58 R1 = [cos(-alfa) -sin(-alfa) ; sin(-alfa) cos(-alfa)]; % Reverse Rot
    mat
59
60 v2 = v1*R; % Create right-hand vector
61 v3 = v1*R1; % Create left-hand vector
62 x1 = endpair; % Top of the arrow
63 x2 = x1 + v2; % Right-hand arrowhead point
64 x3 = x1 + v3; % Left-hand arrowhead point
65 x4 = 0.5*(x2+x3); % Create endpoint of shaft
66 hold on;
67 % Begin plot
68 plot([startpair(1) x4(1)],[startpair(2) x4(2)],...
69     'linewidth',shaftsize,'color',shaftcolor);
70 fill([x1(1) x2(1) x3(1)],[x1(2) x2(2) x3(2)],headcolor);
```

Bibliography

- Alves da Silveira, O. A., Ono Fonseca, J. S., and Santos, I. F. (2015). Actuator topology design using the controllability gramian. *Structural and Multidisciplinary Optimization*, 51(1):145–157.
- Andreassen, C. S., Gersborg, A. R., and Sigmund, O. (2009). Topology optimization of microfluidic mixers. *International Journal for Numerical Methods in Fluids*, 61(5):498–513.
- Andreassen, E., Clausen, A., Schevenels, M., Lazarov, B., and Sigmund, O. (2011). Efficient topology optimization in matlab using 88 lines of code. *Structural and Multidisciplinary Optimization*, 43(1):1–16.
- Barboni, R., Mannini, A., Fantini, E., and Gaudenzi, P. (2000). Optimal placement of pzt actuators for the control of beam dynamics. *Smart Materials and Structures*, 9(1):110.
- Begg, D. W., Liu, X., and Matravers, D. R. (1997). Optimal topology/actuator placement design of structures using sa.
- Bendsoe, M. P. and Kikuchi, N. (1988). Generating optimal topologies in structural design using a homogenization method. *Computer Methods in Applied Mechanics and Engineering*, 71(2):197 – 224.
- Bendsoe, M. P. and Sigmund, O. (2003). *Topology Optimization, theory, methods and applications*. Springer.
- Bourdin, B. (2001). Filters in topology optimization. *International Journal for Numerical Methods in Engineering*, 50(9):2143–2158.
- Broxterman, S. (2016). Mathworks file exchange: Arrowz.m. <https://nl.mathworks.com/matlabcentral/fileexchange/59660-arrowz-startpair-endpair-varargin->.
- Bruyneel, M. and Duysinx, P. (2005). Note on topology optimization of continuum structures including self-weight. *Structural and Multidisciplinary Optimization*, 29(4):245–256.

- Buhl, T. (2002). Simultaneous topology optimization of structure and supports. *Structural and Multidisciplinary Optimization*, 23(5):336–346.
- Chickermane, H. and Gea, H. C. (1997). Design of multi-component structural systems for optimal layout topology and joint locations. *Engineering with Computers*, 13(4):235–243.
- Foutsitzi, G. A., Gogos, C. G., Hadjigeorgiou, E. P., and Stavroulakis, G. E. (2013). Actuator location and voltages optimization for shape control of smart beams using genetic algorithms. *Actuators*, 2(4):111–128.
- Groenwold, A. and Etman, L. (2009). A simple heuristic for gray-scale suppression in optimality criterion-based topology optimization. *Structural and Multidisciplinary Optimization*, 39(2):217–225.
- Groenwold, A. A. and Etman, L. F. P. (2010). A quadratic approximation for structural topology optimization. *International Journal for Numerical Methods in Engineering*, 82(4):505–524.
- Guest, J. K., Prévost, J. H., and Belytschko, T. (2004). Achieving minimum length scale in topology optimization using nodal design variables and projection functions. *International Journal for Numerical Methods in Engineering*, 61(2):238–254.
- Huang, X. and Xie, Y. (2007). Convergent and mesh-independent solutions for the bi-directional evolutionary structural optimization method. *Finite Elements in Analysis and Design*, 43(14):1039 – 1049.
- Jihong, Z. and Weihong, Z. (2006). Maximization of structural natural frequency with optimal support layout. *Structural and Multidisciplinary Optimization*, 31(6):462–469.
- Langelaar, M. (2012). Engineering optimization: Concepts and applications, wb1440. Engineering Optimization Course WB1440.
- Liu, K. and Tovar, A. (2014). An efficient 3d topology optimization code written in matlab. *Structural and Multidisciplinary Optimization*, 50(6):1175–1196.
- Ma, Z.-D., Kikuchi, N., and Cheng, H.-C. (1995). Topological design for vibrating structures. *Computer Methods in Applied Mechanics and Engineering*, 121(1-4):259–280.
- Maeda, Y., Nishiwaki, S., Izui, K., Yoshimura, M., Matsui, K., and Terada, K. (2006). Structural topology optimization of vibrating structures with specified eigenfrequencies and eigenmode shapes. *International Journal for Numerical Methods in Engineering*, 67(5):597–628.
- Qian, Z. and Ananthasuresh, G. K. (2004). Optimal embedding of rigid objects in the topology design of structures. *Mechanics Based Design of Structures and Machines*, 32(2):165–193.
- Querin, O., Steven, G., and Xie, Y. (2000). Evolutionary structural optimisation using an additive algorithm. *Finite Elements in Analysis and Design*, 34(3 - 4):291 – 308.
- Reliant Systems Inc. (2017). Wafer inspection stage assemblies. <http://reliantsystemsinc.com/contract-manufacturing/>.

- Rixen, D. J. (2008). Engineering dynamics lecture notes. Engineering Dynamics Course WB1418.
- Rozvany, G., Zhou, M., and Birker, T. (1992). Generalized shape optimization without homogenization. *Structural optimization*, 4(3-4):250–252.
- Sheng, L. and Kapania, R. K. (2001). Genetic algorithms for optimization of piezoelectric actuator locations. *AIAA Journal*, 39(9):1818–1822.
- Sigmund, O. (1997). On the design of compliant mechanisms using topology optimization. *Mechanics of Structures and Machines*, 25(4):493–524.
- Sigmund, O. (2000). Topology optimization: A tool for the tailoring of structures and materials. *Philosophical Transactions - Mathematical Physical and Engineering Sciences*, 358(1765):211–288.
- Sigmund, O. (2001a). A 99 line topology optimization code written in matlab. *Structural and Multidisciplinary Optimization*, 21(2):120–127.
- Sigmund, O. (2001b). Design of multiphysics actuators using topology optimization - part i one-material structures. *Computer Methods in Applied Mechanics and Engineering*, 190(49 - 50):6577 – 6604.
- Sigmund, O. and Petersson, J. (1998). Numerical instabilities in topology optimization a survey on procedures dealing with checkerboards, mesh-dependencies and local minima. *Structural optimization*, 16(1):68–75.
- Svanberg, K. (1987). The method of moving asymptotes - a new method for structural optimization. *International Journal for Numerical Methods in Engineering*, 24(2):359–373.
- Thomas, H., Zhou, M., and Schramm, U. (2002). Issues of commercial optimization software development. *Structural and Multidisciplinary Optimization*, 23(2):97–110.
- Zhu, J. and Zhang, W. (2010). Integrated layout design of supports and structures. *Computer Methods in Applied Mechanics and Engineering*, 199(9-12):557–569.
- Zhu, J.-H., Hou, J., Zhang, W.-H., and Li, Y. (2014). Structural topology optimization with constraints on multi-fastener joint loads. *Structural and Multidisciplinary Optimization*, 50(4):561–571.

