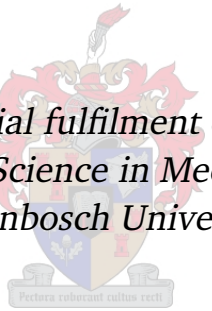
Predominantly solid-void three-dimensional topology optimisation using open source software

by

William Hunter



Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Mechanical Engineering at Stellenbosch University

Department of Mechanical and Mechatronic Engineering
University of Stellenbosch
Private Bag X1
Matieland
7602
Republic of South Africa

Supervisor: Professor Albert A. Groenwold

March 2009

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, and that I am the owner of the copyright thereof (unless to the extend explicitly otherwise stated) and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Signature: .....
William Hunter

Date: .....

Copyright © 2009 Stellenbosch University
All rights reserved.

Uittreksel

Oorwegend swart-wit drie-dimensionele topologie optimering deur die gebruik van oopbronsagteware

William Hunter

Departement Meganiese en Megatroniese Ingenieurswese
Universiteit van Stellenbosch
Privaatsak X1
Matieland
7602
Republiek van Suid Afrika

Tesis: MScIng (Meganies)

Maart 2009

Hierdie tesis is deur die gewilde 99-lyn MATLAB kode van Sigmund vir die minimum buigbaarheid (maksimum styfheid) topologie optimeringsprobleem geïnspireer. 'n Oopbronsagteware weergawe van die MATLAB kode, genaamd ToPy, is ontwikkel in Python. ToPy brei op verskillende maniere op die 99-lyn kode van Sigmund uit.

Eerstens kan ToPy drie verskillende probleemtipes oplos, naamlik minimum buigbaarheid, warmte geleiding en meganisme sintese (samestelling), in twee-dimensionele (2D) of drie-dimensionele (3D) ruimte. Tweedens, deur gebruik te maak van gevestigde oopbronsagteware (Pysparse en sy iteratiewe oplosser) om die yl eindige element stelsel van vergelykings op te los, verskaf ToPy verbeterde spoed en skalering. Die gebruik van ToPy is relatief eenvoudig, en verg slegs die verandering van 'n eenvoudige intreelêer.

ToPy maak ook voorsiening vir 'n grysskaalfilter wat dit moontlik maak om oorwegend, of selfs volledige, swart-wit ontwerpe te verkry.

Die eksponensiële benadering van die doelfunksie is geïmplementeer. Hierdie benadering is 'n veralgemening van die resiproke benadering; laasgenoemde is veral gewild in struktuuroptimering. Die waardes van die eksponente kan gebasseer word op vorige gradiënte in die optimeringsproses, of voorgeskrewe waardes wat direk gespesifiseer word in die gees van optimaliteitsmetodes.

'n Verdere veralgemening is 'n diagonaal kwadratiese benadering tot die eksponensiële benadering wat in 'n sekwensiële benaderde optimeringsomgewing geïmplimenteer is. Dit is van verdere belang dat die diagonaal kwadratiese benadering tot die eksponensiële benadering suksesvol met die grysskaalfilter gekombineer is. Hierdie bydrae is van belang aangesien dit voorheen voorspel was dat die grysskaalfilter slegs saam met monotoon dalende funksies soos die resiproke benadering gebruik kan word.

Abstract

Predominantly solid-void three-dimensional topology optimisation using open source software

William Hunter

Department of Mechanical and Mechatronic Engineering
University of Stellenbosch
Private Bag X1
Matieland
7602
Republic of South Africa

Thesis: MScEng (Mechanical)

March 2009

Inspired by Sigmund's 99-line MATLAB code for minimum compliance (maximum stiffness) topology optimisation, this thesis presents an open source software (OSS) version developed in Python, denoted ToPy. ToPy extends the 99-line code of Sigmund in a number of ways.

Firstly, ToPy can solve three different problem types, namely minimum compliance, heat conduction and mechanism synthesis, in two-dimensional (2D) or three-dimensional (3D) space. This is accomplished by simply changing an input file. Secondly, by using established open source software (Pysparse and its iterative solver) for solving the sparse finite element (FE) systems of equations, the ToPy code provides improved speed and scalability.

ToPy also provides for grey-scale filtering (GSF) to yield predominantly, or even purely, solid-void or black-and-white designs in 2D and 3D space.

In addition, an exponential approximation to the objective function is implemented. This approximation is a generalisation of the reciprocal approximation so popular in structural optimisation; the values of the exponents may be based on gradient information in previously visited iterates, or fixed exponents may be prescribed, in the spirit of optimality criterion (OC) methods.

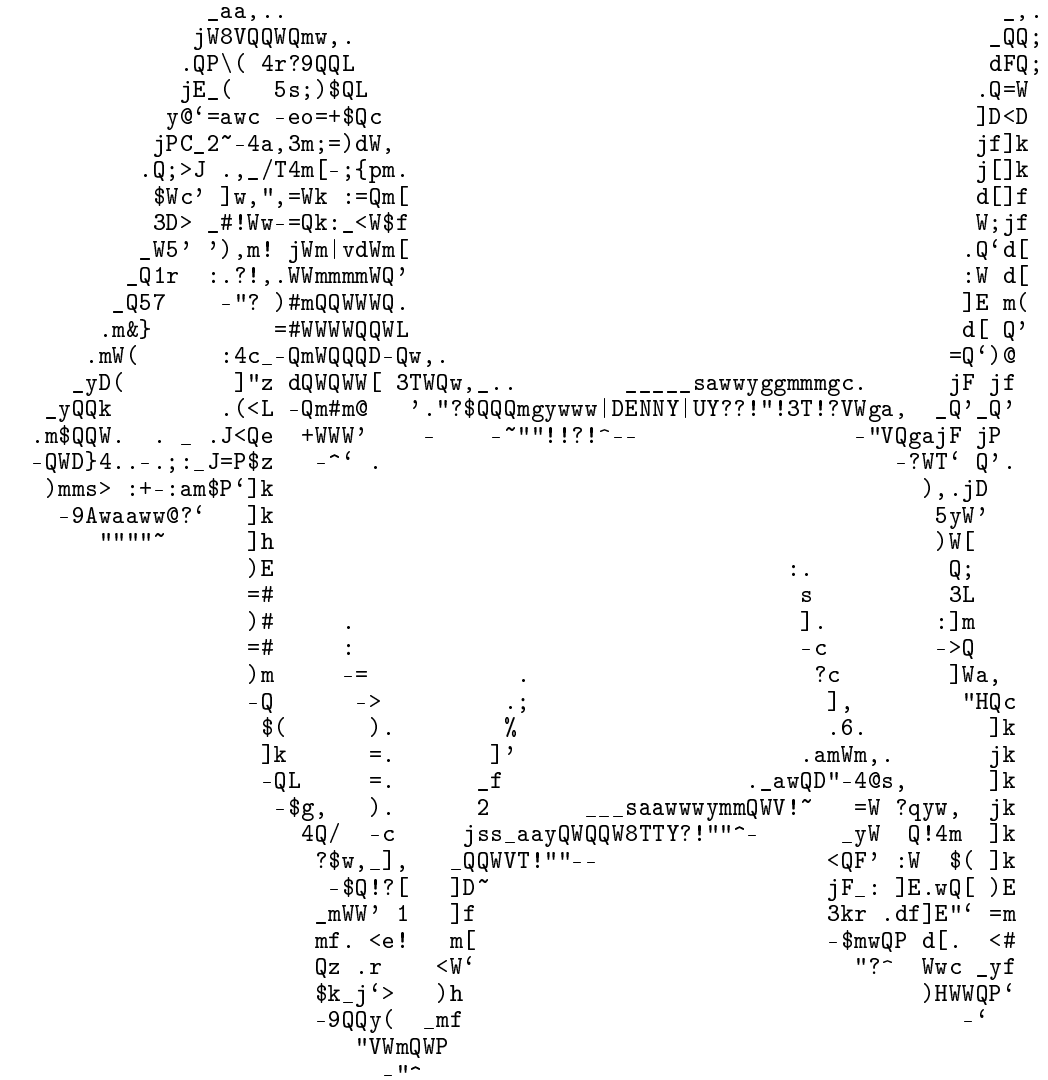
As a further generalisation, the diagonal quadratic approximation to the exponential approximation in an SAO setting is also implemented. What is more: the diagonal quadratic approximation to the exponential approximation was successfully used in combination with GSF. This is a novelty of some importance as it was previously suggested that GSF can only be used in combination with strictly monotonic objective functions, like the reciprocal approximation.

Acknowledgements

I would herewith sincerely like to thank my study leader, Professor Albert A. Groenwold, for giving me the freedom to follow my nose (and bump my head), for helping me to get rid of some hard-to-find bugs, for being patient, enthusiastic, and for always having time to solve a problem and answer (even silly) questions.

It is superfluous to say that I could not have done this without his help.

Dedications



Contents

List of Figures

List of Tables

Nomenclature

Abbreviations

BW	Black-and-white
CAD	Computer-aided design/drawing
DoF	Degrees of freedom
FEA	Finite element analysis
FE	Finite element
FEM	Finite element method
GNU	Recursive acronym for “GNU’s Not Unix”
GSF	Grey-scale filter
ISE	Incomplete series expansion
KKT	Karush-Kuhn-Tucker
MBB	Messerschmitt-Bölkow-Blohm (beam)
NURBS	Non-uniform rational basis spline
OC	Optimality criterion
OSS	Open source software
SAO	Sequential approximate optimisation
SIMP	Solid isotropic microstructure (or material) with penalisation
S-V	Solid-void
ToPy	Topology optimisation with Python
TPD	ToPy problem definition
Vol	Volume
2D	Two-dimensional (space)
3D	Three-dimensional (space)
99-line	Sigmund’s 99-line MATLAB code for compliance minimisation

Mathematical symbols and constants

δ	Move limit (for trust region)
ϵ	Tolerance
η	Damping factor
λ	Lagrange multiplier
a	Exponent (for exponential approximation)
c	Curvature
C	Closed convex set
f	Objective function
$\log_e$	Natural logarithm
L	Lagrangian (function)
m	Number of constraints
n	Number of design variables ($\boldsymbol{x}$)
v	Volume of (finite) element
v_0	Total volume of the design domain
$\bar{v}$	Prescribed limit on the final volume *fraction*
x	Design variable
$\check{x}$	Lower bound on x_i
$\hat{x}$	Upper bound on x_i

Matrices (uppercase) and vectors (lowercase)

$\boldsymbol{K}$	Finite element global (assembled) stiffness matrix
$\boldsymbol{K}_i$	Finite element stiffness matrix for element i
$\boldsymbol{K}_e$	Finite element stiffness matrix
$\boldsymbol{x}$	Design variables
$\boldsymbol{q}$	Finite element displacement vector

$\boldsymbol{r}$	Finite element load vector

Subscripts

i	The i^{th} design variable (or i^{th} element)
e	Relating to a (finite) element
L	Linear
I	Intervening
R	Reciprocal
E	Exponential

Superscripts

a	Exponent (for exponential approximation)
p	Penalisation factor in SIMP
q	Parameter in GSF
$\{k\}$	The k^{th} iteration
$*$	Indicates the solution to the optimisation problem

Miscellaneous

.py	Python file extension
.tpd	ToPy problem definition file extension

Chapter

1

Introduction

In this chapter we motivate and state the objectives of this study; give a very brief introduction of mathematical optimisation and an overview of the thesis.

1.1 Motivation and objectives

Topology optimisation has become a well established[1] technology used by engineers in the fields of aeronautical, civil, materials, mechanical and structural optimisation.

For a given area or volume (referred to as the design domain), topology optimisation may be described as the process of finding an optimal distribution of a limited amount of material in the design domain, to perform some predefined function. For example, for the 'standard' minimum compliance problem, one seeks a distribution of material that produces a maximally stiff structure. What is more, we seek a discrete, solid-void[2] (or black-and-white) solution, *i.e.*, regions containing either material, or no material at all.

The SIMP (**s**olid **i**sotropic **m**icrostructure or **m**aterial with **p**enalisation) method of Rozvany (2000) (see Section 2.3) is an efficient approach used to solve the discrete topology optimisation problem by transforming it to a relaxed and penalised continuous problem. The method does however have a drawback: It produces intermediate (or grey) values in the design variables, *i.e.*, one does *not* obtain a discrete *solid-void* solution, as is indeed sought. The intermediate values can be eliminated to some extent by using higher values for the penalisation parameter, but at the cost of introducing local minima and creating an ill-posed problem when the finite element equations are solved.

SIMP is however fairly easy to code; Sigmund's popular 99-line MATLAB code (Sigmund, 2001) is proof thereof. The 99-line code solves the minimum compliance problem in 2D

[1] Confirmed, to some degree, by the availability of commercial software such as TOSCA Structure, Genesis Topology Optimization, Altair OptiStruct, ANSYS Topology Optimization and MSC.Nastran Topology Optimization.

[2] The term "black-and-white", while perfectly acceptable in 2-dimensional (2D) space, makes less sense in 3-dimensional (3D) space. In 2D we never have a situation where one element (design variable) obscures another, and it is therefore practical to simply represent an element's value with a shade of grey. In 3D, on the other hand, any element that is grey, or even white, can always obscure (hide) other elements. For this reason we prefer to refer to 'black' elements as 'solid', 'white' as 'void' and 'grey' as 'intermediate' in 3D space.

space and is publically available at `www.topopt.dtu.dk`. The downside is that it requires MATLAB itself to use the 99-line code, which is expensive proprietary software. 'Expensive' is of course subjective, but it is perhaps fair to say that software is expensive when it costs more than twice the price of the required hardware it runs on[3].

In this thesis the focus is on solving

1. the minimum compliance topology optimisation problem in 3D space and
2. extending the coverage to heat conduction and
3. compliant mechanism synthesis problems.

In addition, the ultimate goal is to obtain a result consisting predominantly, or even purely, of discrete solid-void regions within the original design domain. This is not possible with the original code of Sigmund.

Finally, we do not wish to make use of any commercial software. Instead, we develop an open source Python package making use of other publically available open source software (OSS). The developed software is to be made available in the public domain and is inspired and based on the very popular 99-line topology optimisation code written by Sigmund.

Most importantly, we do not merely present a 3D open source version of the 2D 99-line topology optimisation code of Sigmund. Instead, we make provision for

1. predominantly solid-void (black-and-white) designs,
2. exponential intervening variables (the value of which may be arbitrarily chosen or automatically determined using first-order gradient information at the previously visited point) rather than reciprocal intervening variables (the latter is of course also possible) and
3. diagonal quadratic approximations. These have never before been used in topology optimisation and certainly not to yield predominantly solid-void designs.

All of the above is available for 2D and 3D problems. Also, a very easy to use text interface is provided, so that the user does not even have to write a single line of code!

The question may arise, why use OSS? The answer is twofold: cost and transparency. Cost is obvious; publically available OSS is free. Transparency carries with it practical implications, such as a possible large user base, which in turn has spin-offs such as user contributions in terms of bug fixes, improvements to the code, implementation of new methods, and documentation and testing.

[3] In South Africa, it is certainly fair to say that MATLAB is expensive indeed.

1.2 Optimisation

In the simplest terms, (mathematical) optimisation is the process of minimising (or maximising) some function. In general, we have a function that has to be *minimised with respect to* some variable(s), *subject to* certain limitations. The function that we want to minimise is called the *objective function* and the limitations are called *constraints*. The latter can be a mixture of inequality and equality constraint functions, and side constraint equations.

A general nonlinear constrained optimisation problem can be written as:

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & f(\boldsymbol{x}), && \boldsymbol{x} = [x_1, x_2, \cdots, x_n]^T \in \mathbb{R}^n, \\ \text{subject to} \quad & g_j(\boldsymbol{x}) \leq 0, && j = 1, 2, \cdots, m, \\ & h_k(\boldsymbol{x}) = 0, && k = 1, 2, \cdots, r, \\ & \check{x}_i \leq x_i \leq \hat{x}_i, && i = 1, 2, \cdots, n, \end{aligned} \tag{1.2.1}$$

where $f(\boldsymbol{x})$, $g_j(\boldsymbol{x})$ and $h_k(\boldsymbol{x})$ are scalar functions of the real column vector $\boldsymbol{x}$, the design variables. In the above, each $g_j(\boldsymbol{x})$ represents an inequality constraint and each $h_k(\boldsymbol{x})$ an equality constraint. The last equations directly define bounds on the design variables $\boldsymbol{x}$ and are referred to as the side constraint equations (Vanderplaats, 2001). It is usually convenient to treat the latter separately since they define the region of search for the optimum, namely between the lower bound $\check{x}$ and the upper bound $\hat{x}$ (inclusive).

1.2.1 Minimum compliance

Here we seek to find the optimal distribution of material that renders the stiffest possible structure for a given volume fraction[4] subjected to loads and tractions (forces) and boundary conditions (supports). It is easy to show that maximising for stiffness is equivalent to minimising for compliance or strain energy.

1.2.2 Heat conduction

Our objective is to find the optimal distribution of a conductive material for the purpose of heat transfer, *i.e.*, we want to maximise the conductivity of a design domain. This is the same as minimising the temperature gradient over the domain, since a *poor* conductor will create a *large* temperature gradient (or difference) between the source and the sink.

1.2.3 Mechanism synthesis

The objective of compliant mechanism synthesis (or design), is to obtain a device that can convert an input displacement in one location to an output displacement in another location. The goal of applying topology optimisation to mechanism synthesis is to maximise the output for a specific input force, or minimise the input for a specified output force. To accomplish this, the amount of material at ones disposal must be distributed in the structurally most efficient way. By their very nature, mechanisms are non-linear, but we will restrict ourselves to linear models herein, *e.g.*, see Bendsøe and Sigmund (2004).

[4] Ratio of the solid volume to the total design domain volume.

1.3 Overview of the thesis

The contents of each chapter and appendix are given below:

Chapter 2 A brief history of SIMP and solid-void topology optimisation and an overview of the topology optimisation problems we set out to solve, namely, (a) minimum compliance, (b) heat conduction, and (c) mechanism synthesis.

Chapter 3 We present solution strategies for solving the problems of Chapter 2, in particular the sequential approximate optimisation (SAO) method with exponential intervening variables. Reciprocal intervening variables is a special case of exponential intervening variables.

Next, the optimality criteria (OC) method is briefly discussed and we then move on to the so-called grey-scale filter (GSF), which enables us to produce predominantly, or even pure, solid-void designs.

Finally, we cover quadratic approximation combined with GSF.

Chapter 4 The formulation of a 3D penalised equilibrium finite element for bending dominated problems is presented.

Chapter 5 We cover the implementation of the solution strategies of Chapter 3 by briefly presenting the software, "**t**opology **o**ptimisation with **Py**thon" (ToPy) developed herein.

Chapter 6 We present results for 3D topology optimisation using ToPy.

Chapter 7 Conclusions and suggestions for future work are presented.

Appendix A The installation and use of ToPy, as well as a listing of the complete source code.

Appendix B Some of the TPD (ToPy problem definition) files we used to produce our results are listed in this appendix.

Appendix C We present simple examples demonstrating the use and application of the Falk dual to separable problems.

Appendix D Two listings of wxMaxima sessions used to produce the symbolic results of Chapter 4.

Chapter

2

Popular topology optimisation problems

Three types of topology optimisation problems are considered, namely (a) minimum compliance, (b) heat conduction and (c) mechanism synthesis. These were selected simply because they are so popular and frequently used, *e.g.*, see Bendsøe and Sigmund (2004). A complete problem statement is given for (a) and only brief statements for problems (b) and (c) since the forms are almost the same.

2.1 A brief history of SIMP

Finite element based topology optimisation can be attributed to Rossow and Taylor (1973), who used an unpenalised relation and therefore obtained results containing intermediate ('grey') densities. Bendsøe and Kikuchi (1988), in a milestone paper, suggested homogenisation based on rectangular or square holes (voids), which produced a degree of penalisation (Rozvany, 2009).

The possibility of a SIMP-like method, first introduced under the names "the direct approach/method" or "artificial density approach/method", was also mentioned by Bendsøe (1989), although at the time he objected to the method based on theoretical grounds (net dependance and fictitiousness of material properties); he expressed preference for homogenisation methods.

The efforts of Rozvany and Zhou (1991) and the *University of Essen* research group should also be acknowledged for independently suggesting and significantly popularising SIMP. The term 'SIMP' was coined by Rozvany and first introduced in Rozvany *et al.* (1992).

Probably because SIMP is easy to code, as mentioned in Section 1.1, and since there are numerous established FEA software programs on the market, most industrial software for topology optimisation today employ SIMP as it is fairly easy to incorporate in an established finite element analysis software suite.

For a more detailed history of SIMP, refer to the already cited papers and also Long *et al.* (2008).

2.2 A brief history of solid-void topology optimisation

Methods previously proposed to generate predominantly solid-void or black-and-white designs include post-processing methods like contour plotting, which can in general not satisfy the constraints. On the other hand, established discrete optimisation methods like branch-and-bound (B&B) can also not be used, due to the very high dimensionality of the optimisation problem, *e.g.*, see Section 2.3 below. Even neighbourhood search (NHS) methods are for this reason effectively disqualified. Methods based on rounding are computationally efficient, but are often not guaranteed to satisfy the constraints, while they may also prove to be very ineffective if very high fractions of intermediate density ('grey') material are present before rounding is initiated.

Other methods previously proposed include the following (Groenwold and Etman, 2009):

1. Volumetric penalisation by Zhou and Rozvany (1991) and Bruns (2005), which uses a concave constraint which may be undesirable in methods based on convexity which is very involved,
2. a nonlinear diffusion technique by Wang *et al.* (2004),
3. hierarchical constraining of the slope of density by Zhou *et al.* (2001); an iterative procedure requiring three consecutive operations,
4. a discrete dual method by Beckers (1996) that requires heuristics to solve the discrete dual,
5. a hierarchical neighbourhood search method by Svanberg and Werme (2005) that is limited in the number of variables it can handle and
6. a family of morphology-based filters by Sigmund (2007), which require very many iterations.

2.3 Minimum compliance: problem statement

The 'real' minimum compliance problem is a distributed, discrete valued design problem, which consists of calculating the compliance (the inverse of stiffness) for each possible permutation of the design domain. Thus, if we discretise a 2D domain into an X-by-Y mesh of finite elements, and knowing that each element has two possible values (0 and 1), we have $2^{(X\times Y)}$ possible permutations of the domain. As we can imagine, this is extremely expensive to compute, *e.g.*, for a small 4-by-4 domain, we have to calculate $2^{(4\times 4)} = 65536$ possible material distributions (designs) and evaluate each one in order to find the design's compliance, with each requiring a finite element analysis (FEA)[1]. The problem is further compounded in that each FEA becomes computationally more expensive as the domain discretisation is increased (there are more elements). We therefore need an alternative approach to solve the discrete problem, one that will require far less computational effort.

[1] Strictly speaking, this is of course not true, since designs that violate the volume constraint, need not be evaluated.

The above mentioned problem can be solved using several different approaches. In this thesis we will use the SIMP method (see Sigmund (2001)). The method was independently proposed by Bendsøe (1989) and Rozvany and Zhou (1991). Basically, the approach is to replace the discrete variables with continuous variables and then to introduce some form of penalty that will drive the solution to discrete solid-void values. The element stiffness matrix is then modified so that it becomes a function of the continuous variables, where the latter are now the design variables. The continuous design variables are commonly interpreted as the density of the material.

We rewrite the general form of a mathematical optimisation problem (1.2.1) as a *minimum compliance* topology optimisation problem. The discrete problem is transformed into a relaxed and penalised continuous problem, which means that we can make use of sequential approximate optimisation (SAO) or the optimailty criteria (OC) method. (As we will see later, these are in fact equivalent). The SIMP problem for minimum compliance is

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & f(\boldsymbol{x}) = \boldsymbol{q}^T \boldsymbol{r} = \sum_{i=1}^{n} (x_i)^p \boldsymbol{q}_i^T \boldsymbol{K}_i \boldsymbol{q}_i, \\ \text{subject to} \quad & g_j(\boldsymbol{x}) \leq \boldsymbol{0}, \qquad j = 1, 2, \ldots, m, \\ & \boldsymbol{K}\boldsymbol{q} = \boldsymbol{r}, \\ & 0 < \check{x}_i \leq x_i \leq 1, \qquad i = 1, 2, \ldots, n. \end{aligned} \tag{2.3.1}$$

The objective function $f(\boldsymbol{x})$ represents compliance or strain energy, x_i represents the design variable, *i.e.*, a finite element. Thus, $(x_i)^p$ represents the penalised design variable (or density) and p is the SIMP penalty parameter (see Bendsøe and Sigmund (2004) for details). Note that we make use of a lower bound $\check{x}_i$ on the density in order to prevent any possible singularity of the equilibrium problem. The $\boldsymbol{q}$ represents the finite element global displacement vector, $\boldsymbol{q}_i$ represents the elemental displacement vector, $\boldsymbol{r}$ the global load vector, $\boldsymbol{K}_i$ the element stiffness matrix and $\boldsymbol{K}$ the global assembled finite element stiffness matrix. The m linear constraints are represented by $g_j(\boldsymbol{x})$ and the last equation represents the side constraints on x_i. Subscript i indicates elemental quantities and operators, *i.e.*, $\boldsymbol{q}_i$ refers to the displacement contribution of element i.

We will restrict ourselves to n equally sized finite elements in the discretised design domain (the finite element mesh), with the corresponding vector of design variables $\boldsymbol{x} = [x_1, x_2, \ldots, x_n]^T$.It is assumed that the load vector, $\boldsymbol{r}$, is independent of the design variables $\boldsymbol{x}$, *i.e.*, $\boldsymbol{r} \neq f(\boldsymbol{x})$.

To obtain the well known 'standard' minimum compliance problem with a single constraint, we set $m = 1$ and formulate the constraint

$$g(\boldsymbol{x}) = \frac{v(\boldsymbol{x})}{v_0} - \bar{v} = \frac{1}{v_0} \sum_{i=1}^{n} v_i x_i - \bar{v} \leq 0,$$

where $v(\boldsymbol{x}) = v$ represents the final volume[2] occupying the design domain, v_0 the total volume of the design domain, $\bar{v}$ is the prescribed limit on the final volume *fraction*, and v_i is the volume of each element. When the finite elements are all equally sized, v_i is the same for each element and can therefore be moved "in front of" the summation symbol and be

[2]Note that *volume* is understood to refer to *area*×1 in the 2D domain.

replaced by v_e. $\boldsymbol{K}_i$ too, is the same for each element, and is replaced by $\boldsymbol{K}_e$. Finally we can write (2.3.1) as

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & f(\boldsymbol{x}) = \sum_{i=1}^{n} (x_i)^p \boldsymbol{q}_i^T \boldsymbol{K}_e \boldsymbol{q}_i, \\ \text{subject to} \quad & g(\boldsymbol{x}) = \frac{v_e}{v_0} \sum_{i=1}^{n} x_i - \bar{v} \leq 0, \\ & \boldsymbol{K}\boldsymbol{q} = \boldsymbol{r}, \\ & 0 < \check{x}_i \leq x_i \leq 1, \qquad i = 1, 2, \ldots, n. \end{aligned} \tag{2.3.2}$$

2.4 Heat conduction: problem statement

In thermodynamics, *heat* is defined as energy transfer due to temperature gradients (differences). Only two modes of heat transfer are recognised by thermodynamics, namely conduction and radiation (convection is the transport of energy by motion of a medium, such as air (Mills, 1995)). For a material that can be modelled as isotropic, *e.g.*, steel, conductivity is the same in all directions.

The finite element formulation for heat transfer gives the following equation for *steady-state* conditions:

$$\boldsymbol{K}\boldsymbol{q} = \boldsymbol{r}, \tag{2.4.1}$$

where the $\boldsymbol{q}$ now represents the finite element global nodal temperature vector, $\boldsymbol{r}$ the global thermal load vector, $\boldsymbol{K}_e$ the element conductivity matrix and $\boldsymbol{K}$ represents the global assembled finite element conductivity matrix. As suggested by Bendsøe and Sigmund (2004, Section 5.1.6), we formulate the optimisation problem as

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & f(\boldsymbol{x}) = \sum_{i=1}^{n} \{\check{x}_i + (1 - \check{x}_i)(x_i)^p\} \boldsymbol{q}_i^T \boldsymbol{K}_e \boldsymbol{q}_i, \\ \text{subject to} \quad & g(\boldsymbol{x}) = \frac{v_e}{v_0} \sum_{i=1}^{n} x_i - \bar{v} \leq 0, \\ & \boldsymbol{K}\boldsymbol{q} = \boldsymbol{r}, \\ & 0 < \check{x}_i \leq x_i \leq 1, \qquad i = 1, 2, \ldots, n. \end{aligned} \tag{2.4.2}$$

The adopted material model may be understood as representing a 'mixture' of two materials, one a good and the other a poor conductor, where we want to determine the optimal distribution of these two materials, see Bendsøe and Sigmund (2004, Section 2.9.2) for details.

2.5 Compliant mechanism synthesis: problem statement

A mechanism is a device used to transfer motion and/or force. Traditional mechanisms consist of parts (or members) connected to each other by means of hinges. A compliant

mechanism, unlike traditional mechanisms, derives some of its mobility from the displacement of flexible parts, *i.e.*, certain joints deform in order to give mobility to parts. Large displacements may cause mechanical locking of some members, thus, we need to perform non-linear analysis to obtain useful results for large displacement mechanisms (Bendsøe and Sigmund, 2004).

Nevertheless, as a "first attempt" design, linear models can certainly be used to approximate the behaviour of non-linear (realistic) mechanisms. The linear compliant mechanism design problem may typically be expressed as

$$\begin{aligned} \max_{\boldsymbol{x}} \quad & f(\boldsymbol{x}) = \boldsymbol{l}^T \boldsymbol{q}, \\ \text{subject to} \quad & g(\boldsymbol{x}) = \frac{v_e}{v_0} \sum_{i=1}^{n} x_i - \bar{v} \leq 0, \\ & \boldsymbol{K}\boldsymbol{q} = \boldsymbol{r}, \\ & 0 < \check{x}_i \leq x_i \leq 1, \qquad i = 1, 2, \ldots, n, \end{aligned} \tag{2.5.1}$$

with $\boldsymbol{l}$ a d-vector, and d the total number of degrees of freedom present in the structure (with $d \geq n$). The components $l_i = 0 \ \forall \ i$, except for the to-be-specified characteristic displacements, for which $l_i = 1$. The design variables x_i may be obtained using the *adjoint method* (also known as the *dummy-load method*), *e.g.*, see Bendsøe and Sigmund (2004, Section 2.6.4).

Chapter

3

Solution strategies

In this chapter we present the sequential approximate optimisation (SAO) method for solving the topology problem, as well as the popular optimality criteria (OC) method. We will then show that these methods are in fact *equivalent*! Additionally, we will present a grey-scale filter (GSF) to produce predominantly solid-void designs, and finally we will present diagonal quadratic approximations to the exponential and reciprocal approximations.

3.1 Sequential approximate optimisation (SAO)

A popular approach to solve the minimum compliance topology optimisation problem (2.3.2) is to use SAO. SAO constructs primal analytical subproblems of the original problem (2.3.2) at successive approximations. We repeat problem (2.3.2) here for convenience:

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & f(\boldsymbol{x}) = \sum_{i=1}^{n} (x_i)^p \boldsymbol{q}_i^T \boldsymbol{K}_e \boldsymbol{q}_i, \\ \text{subject to} \quad & g(\boldsymbol{x}) = \frac{v_e}{v_0} \sum_{i=1}^{n} x_i - \bar{v} \leq 0, \\ & \boldsymbol{K}\boldsymbol{q} = \boldsymbol{r}, \\ & 0 < \check{x}_i \leq x_i \leq 1, \qquad i = 1, 2, \ldots, n. \end{aligned} \tag{3.1.1}$$

In addition, we introduce a trust region or move limit, δ, on each new point we use for creating the approximation, and set

$$\begin{aligned} \check{x}_i \leftarrow \check{x}_i^{\{k+1\}} &= \max\left(x_i^{\{k*\}} - \delta, \check{x}\right), \\ \hat{x}_i \leftarrow \hat{x}_i^{\{k+1\}} &= \min\left(x_i^{\{k*\}} + \delta, \hat{x}\right), \end{aligned} \tag{3.1.2}$$

where $x_i^{\{k*\}}$ is the solution to subproblem at the kth iteration, $\check{x}$ is the prescribed lower bound and $\hat{x}$ the prescribed upper bound on all x_i. For the topology optimisation problem, we usually set $\check{x} = 1 \times 10^{-3}$, $\hat{x} = 1$ and $\delta = 0.2$.

The success of SAO depends greatly on the quality of the constructed analytical approximations of the objective and constraint functions. One approach is to construct *local* approximations; the simplest approach is probably to make use of a *linear* approximation based on a Taylor series. It is also important that a *trust region* or *move limit* is placed on each new construction point, since the approximation function can only be expected to be accurate over a small region of the allowable search domain. The accuracy of the approximation can be increased by retaining higher order terms of the Taylor series, however, the calculation of the higher order terms can be computationally expensive. An alternative approach to achieve better accuracy is to find *intervening* or *intermediate variables* that would make the approximated function behave more linearly (see, *e.g.*, Vanderplaats (2001) or Haftka and Gürdal (1992)).

In the following we will use the reciprocal (see, *e.g.*, Vanderplaats (2001)) and exponential approximation, although other approximation functions are also possible, *e.g.*, quadratic, refer to Groenwold and Etman (2008) for details.

3.1.1 Primal approximate subproblems

We will restrict the number of approximations used to a very small selection, namely the well known (a) linear, (b) reciprocal and (c) exponential approximations. As mentioned, these approximations are based on a truncated first-order (linear) Taylor series expansion. The reciprocal and exponential approximations make use of intervening variables, in fact, reciprocal intervening variables are just a special case of exponential intervening variables. The three types of approximations mentioned above give different results in terms of the final objective function value and also the final design, see Section 3.6

Thus, in order to construct, say, the exponential approximate subproblem, we simply have to substitute the original problem's objective function (3.1.1) with one of the following approximations.

3.1.1.1 Linear approximation

In order to construct a linear primal approximate subproblem of (3.1.1), we generate a first order Taylor series of the objective function $f(\boldsymbol{x})$ at $\boldsymbol{x}^{\{k\}}$ and get the **linear approximation** as

$$\tilde{f}_L(\boldsymbol{x}) = f(\boldsymbol{x}^{\{k\}}) + \sum_{i=1}^{n} \frac{\partial f(\boldsymbol{x}^{\{k\}})}{\partial x_i}(x_i - x_i^{\{k\}}). \tag{3.1.3}$$

It is of course not necessary to approximate the single constraint on volume, as it is already *linear*...

3.1.1.2 Reciprocal approximation

For the reciprocal approximation we define the intervening variables, $y_i = y_i(x_i), \quad i = 1,2,\ldots,n$, and rewrite the linear approximation, (3.1.3), in terms of the intervening variables as follows,

$$\tilde{f}_I(\boldsymbol{y}) = f(\boldsymbol{y}^{\{k\}}) + \sum_{i=1}^{n} \frac{\partial f(\boldsymbol{y}^{\{k\}})}{\partial y_i}(y_i - y_i^{\{k\}}). \tag{3.1.4}$$

Using the *chain rule*,

$$\frac{\partial f(y)}{\partial y} = \frac{\partial f(y)}{\partial x}\frac{\partial x}{\partial y},$$

we can write (3.1.4) in terms of the *original* variables as

$$\tilde{f}_I(\boldsymbol{x}) = f(\boldsymbol{x}^{\{k\}}) + \sum_{i=1}^{n}\left(\frac{\partial f(\boldsymbol{x}^{\{k\}})}{\partial x_i}\Big/\frac{dy_i}{dx_i}\right).\left(y_i(x_i) - y_i(x_i^{\{k\}})\right). \tag{3.1.5}$$

Now, using the popular *reciprocal* intervening variable, $y_i(x_i) = \dfrac{1}{x_i}$, and substituting into (3.1.5) we get the **reciprocal approximation**:

$$\begin{aligned}\tilde{f}_R(\boldsymbol{x}) &= f(\boldsymbol{x}^{\{k\}}) + \sum_{i=1}^{n}\left(\frac{\partial f(\boldsymbol{x}^{\{k\}})}{\partial x_i}\right)(x_i^{\{k\}})^2\left(\frac{1}{x_i^{\{k\}}} - \frac{1}{x_i}\right)\\ &= f(\boldsymbol{x}^{\{k\}}) + \sum_{i=1}^{n}\left(\frac{\partial f(\boldsymbol{x}^{\{k\}})}{\partial x_i}\right)\left(\frac{x_i^{\{k\}}}{x_i}\right)\left(x_i - x_i^{\{k\}}\right).\end{aligned} \tag{3.1.6}$$

Note that the only difference between (3.1.6) and (3.1.3) is the $\dfrac{x_i^{\{k\}}}{x_i}$ term (the reciprocal).

3.1.1.3 Exponential approximation

Here we make use of the *exponential* intervening variables, $y_i(x_i) = x_i^{a_i}$, and from (3.1.5) we write the **exponential approximation** as

$$\begin{aligned}\tilde{f}_E(\boldsymbol{x}) &= f(\boldsymbol{x}^{\{k\}}) + \sum_{i=1}^{n}\left(\frac{\partial f(\boldsymbol{x}^{\{k\}})}{\partial x_i}\right)\left(\frac{(x_i^{1-a_i^{\{k\}}})^{\{k\}}}{a_i^{\{k\}}}\right)\left(x_i^{a_i^{\{k\}}} - (x_i^{a_i^{\{k\}}})^{\{k\}}\right)\\ &= f(\boldsymbol{x}^{\{k\}}) + \sum_{i=1}^{n}\left(\frac{\partial f(\boldsymbol{x}^{\{k\}})}{\partial x_i}\right)\left(\frac{x_i^{\{k\}}}{a_i^{\{k\}}}\right)\left[\left(\frac{x_i}{x_i^{\{k\}}}\right)^{a_i^{\{k\}}} - 1\right].\end{aligned} \tag{3.1.7}$$

The convexity of (3.1.7) depends on the value of the exponent $a_i^{\{k\}}$, which may be determined in a number of ways. In this thesis, we will use the method proposed by Fadel *et al.*, namely to use first-order gradient information at the previously visited point, $\boldsymbol{x}^{\{k-1\}}$ (for $k \geq 1$), by enforcing $\nabla\tilde{f}_E(\boldsymbol{x}^{\{k-1\}}) = \nabla f(\boldsymbol{x}^{\{k-1\}})$, as shown below.

Since (3.1.7) is separable, we can write

$$\begin{aligned}\frac{\partial \tilde{f}_E(\boldsymbol{x})}{\partial x_i} &= a_i^{\{k\}}\left(\frac{x_i^{a_i^{\{k\}}-1}}{(x_i^{\{k\}})^{a_i^{\{k\}}}}\right)\left(\frac{x_i^{\{k\}}}{a_i^{\{k\}}}\right)\left(\frac{\partial f(\boldsymbol{x}^{\{k\}})}{\partial x_i}\right)\\ &= \left(\frac{x_i}{x_i^{\{k\}}}\right)^{a_i^{\{k\}}-1}\left(\frac{\partial f(\boldsymbol{x}^{\{k\}})}{\partial x_i}\right).\end{aligned} \tag{3.1.8}$$

We see that for $x_i = x_i^{\{k\}}$ in (3.1.7) and (3.1.8),

$$\tilde{f}_E(\boldsymbol{x}^{\{k\}}) = f(\boldsymbol{x}^{\{k\}})$$

and also that

$$\frac{\partial \tilde{f}_E(\boldsymbol{x}^{\{k\}})}{\partial x_i} = \frac{\partial f(\boldsymbol{x}^{\{k\}})}{\partial x_i},$$

thus, our approximation and its derivative is precise in the point $x_i^{\{k\}}$ compared to the original function, and it serves as a 'sanity' check since we want to enforce $\boldsymbol{\nabla}\tilde{f}(\boldsymbol{x}^{\{k-1\}}) = \boldsymbol{\nabla} f(\boldsymbol{x}^{\{k-1\}})$. If we now put $\boldsymbol{x} = \boldsymbol{x}^{\{k-1\}}$ in (3.1.8), we get

$$\frac{\partial \tilde{f}_E(\boldsymbol{x}^{\{k-1\}})}{\partial x_i} = \left(\frac{x_i^{\{k-1\}}}{x_i^{\{k\}}}\right)^{a_i^{\{k\}}-1} \left(\frac{\partial f(\boldsymbol{x}^{\{k\}})}{\partial x_i}\right), \tag{3.1.9}$$

and as mentioned above, we now enforce

$$\frac{\partial \tilde{f}_E(\boldsymbol{x}^{\{k-1\}})}{\partial x_i} = \frac{\partial f(\boldsymbol{x}^{\{k-1\}})}{\partial x_i},$$

which yields the exponent (from (3.1.9)),

$$a_i^{\{k\}} = 1 + \frac{A}{B}, \tag{3.1.10}$$

where

$$A = \log_e \left(\frac{\partial f(\boldsymbol{x}^{\{k-1\}})}{\partial x_i} \Big/ \frac{\partial f(\boldsymbol{x}^{\{k\}})}{\partial x_i}\right),$$
$$B = \log_e(x_i^{\{k-1\}}/x_i^{\{k\}}).$$

ToPy utilises this method of determing or "auto-tuning" the value of a for each iteration. However, it is still possible to set the value of a to a fixed value if required, although in the software we actually make use of η, the so-called damping factor, in order to be consistent with Sigmund's 99-line code. See Appendix A for details.

3.1.2 Dual approximate subproblems

Instead of solving the *primal* approximate reciprocal or exponential subproblem, a *dual* approximate subproblem may be formulated and solved instead.

First, the *Lagrangian* function is constructed (for each iteration k),

$$L^{\{k\}}(\boldsymbol{x},\lambda) = \tilde{f}(\boldsymbol{x}^{\{k\}}) + \lambda g(\boldsymbol{x}^{\{k\}}), \tag{3.1.11}$$

where the single λ represents the Lagrangian multiplier (recall that we only have a single constraint on volume in the classical minimum compliance problem). Also note that the side constraints present in (3.1.1) *are not included* in (3.1.11), instead, the side constraints are accommodated via the closed convex set over which the Falk dual is defined, as will be explained in the paragraphs that follow.

Saddle point theorem: If the primal approximate subproblem is *strictly convex*, the stationary saddle point $(\boldsymbol{x}^*,\lambda^*)$ defines the global minimiser of $L^{\{k\}}(\boldsymbol{x},\lambda)$, with $\boldsymbol{x}^*$ the solution to the associated primal approximate subproblem, *i.e.*, $L^{\{k\}}(\boldsymbol{x}^*,\lambda^*) = \tilde{f}^{\{k\}}(\boldsymbol{x}^*)$, if and only if the Lagrange multipliers satisfy the *KKT* conditions.

Duality: The *dual* function is defined as

$$\zeta(\lambda^{\{k\}}) = \min_{\boldsymbol{x}} \; L^{\{k\}}(\boldsymbol{x},\lambda), \tag{3.1.12}$$

where we have temporarily *excluded* the side constraints. However, if we make use of *Falk's dual formulation*, we can reconstruct the 'normal' dual function (3.1.12) as

$$\gamma(\lambda^{\{k\}}) = \min_{\boldsymbol{x}\in C} \; L^{\{k\}}(\boldsymbol{x},\lambda), \tag{3.1.13}$$

where C is some closed convex set which is introduced to insure the well conditioning of the problem (see Falk (1967) and Haftka and Gürdal (1992) for details). For example, we may conveniently select $C = \{\boldsymbol{x} : \check{x}_i \le x_i \le \hat{x}_i, \quad i = 1,2,\ldots,n\}$, since $\check{x}_i$ and $\hat{x}_i$ represents a closed and bounded set. In an alternative form, we can therefore write the dual function (3.1.13) as

$$\gamma(\lambda^{\{k\}}) = \min_{\boldsymbol{x}} \{L^{\{k\}}(\boldsymbol{x},\lambda) : \check{x}_i \le x_i \le \hat{x}_i\}, \tag{3.1.14}$$

where the side constraints are *now included* as part of the dual function. Using the Falk dual function (3.1.14), the *dual* approximate subproblem can finally be formulated as,

$$\begin{aligned} &\max_{\lambda} \quad \gamma(\lambda^{\{k\}}), \\ &\text{subject to} \quad \lambda \ge 0. \end{aligned} \tag{3.1.15}$$

If the primal approximate problem is *strictly convex*, the solution to the dual (3.1.15) will be identical to that of the primal, again, refer to Falk (1967) for details.

Application to separable problems: In most cases it does not make sense to solve (3.1.15) instead of the primal, since we have more computational work to perform since we obtain a nested optimisation problem. However, if the objective function and constraints are *separable*, the maximisation of (3.1.15) and the minimisation of (3.1.14) become simple to execute (see Appendix C for some simple examples of the method).

An optimisation problem is called separable if the objective function as well as the constraints are separable, *i.e.*, each can be written as a sum of functions of the individual variables $x_1, x_2, \ldots, x_n$, *e.g.*, $f(\boldsymbol{x}) = f_1(x_1) + f_2(x_2) + \ldots + f_n(x_n)$ and $g(\boldsymbol{x}) = g_1(x_1) + g_2(x_2) + \ldots + g_n(x_n)$. This means that we can write the Lagrangian, (3.1.11), as

$$L^{\{k\}}(\boldsymbol{x}, \lambda) = L_1^{\{k\}}(\lambda, x_1) + L_2^{\{k\}}(\lambda, x_2) + \ldots + L_n^{\{k\}}(\lambda, x_n).$$

Each one of the functions $L_r^{\{k\}}$ can then be minimised separately with regards to $\boldsymbol{x}$, by setting the derivative of each equal to zero,

$$\begin{aligned} &\frac{\partial \tilde{f}^{\{k\}}(\boldsymbol{x})}{\partial x_i} + \lambda \frac{\partial g^{\{k\}}(\boldsymbol{x})}{\partial x_i} = 0, \quad i = 1, 2, \ldots, n, \\ &\frac{\partial \tilde{f}^{\{k\}}(\boldsymbol{x})}{\partial x_i} \Big/ \frac{\partial g^{\{k\}}(\boldsymbol{x})}{\partial x_i} = -\lambda, \end{aligned} \tag{3.1.16}$$

and we therefore have $\boldsymbol{x}$ as a function of λ, *i.e.*,

$$x_i^* = x_i^*(\lambda) : \check{x}_i \leq x_i \leq \hat{x}_i, \quad i = 1, 2, \ldots, n. \tag{3.1.17}$$

Finally, we write the *dual approximate subproblem*, as

$$\begin{aligned} \max_{\lambda} \quad & \gamma^{\{k\}}(\lambda) = \tilde{f}^{\{k\}}(\boldsymbol{x}(\lambda)) + \lambda g^{\{k\}}(\boldsymbol{x}(\lambda)), \\ \text{subject to} \quad & \lambda \geq 0. \end{aligned} \tag{3.1.18}$$

The saddle point $(\boldsymbol{x}^*, \lambda^*)$ is found by maximising the dual approximate subproblem, (3.1.18), using (3.1.17). The dual approximate subproblem therefore requires the determination of λ only, and any suitable one-dimensional minimisation method may be used, *e.g.*, bisection, golden section or a polynomial approximation method, if we have one only one constraint (and therefore only one λ).

If we had more than one constraint we could make use of a *conjugate gradient* method such as *Polak-Rebiére* or a *BFGS* search to determine the various λ's. In any case, it reduces to an unconstrained optimisation problem that can be solved with any unconstrained optimisation method able to handle the non-negativity constraint on λ.

3.1.2.1 Reciprocal subproblem

Using the reciprocal approximation (3.1.6), the objective function of the dual approximate subproblem (3.1.18) has the form

$$\gamma^{\{k\}}(\lambda) = f(\boldsymbol{x}^{\{k\}}) + \sum_{i=1}^{n} \left(\frac{\partial f(\boldsymbol{x}^{\{k\}})}{\partial x_i} \right) \left(\frac{x_i^{\{k\}}}{x_i(\lambda)} \right) \left(x_i(\lambda) - x_i^{\{k\}} \right) + \frac{\lambda}{\nu_0} \sum_{i=1}^{n} \nu_i x_i(\lambda) - \bar{\nu}, \tag{3.1.19}$$

and from the stationary conditions for (3.1.19), we find

$$x_i^2(\lambda) = - \left((x_i^{\{k\}})^2 \frac{\partial f(\boldsymbol{x}^{\{k\}})}{\partial x_i} \right) \Big/ \left(\lambda \frac{\partial g(\boldsymbol{x}^{\{k\}})}{\partial x_i} \right). \tag{3.1.20}$$

Recall that the x_i's are bound (see (3.1.2)) and introducing $\beta_i(\lambda) = x_i^2(\lambda)$, we obtain

$$x_i(\lambda) = \begin{cases} \check{x}_i & \text{if} \quad \beta_i^{1/2}(\lambda) \le \check{x}_i, \\ \beta_i^{1/2}(\lambda) & \text{if} \quad \check{x}_i \le \beta^{1/2}(\lambda) \le \hat{x}_i, \\ \hat{x}_i & \text{if} \quad \beta_i^{1/2}(\lambda) \ge \hat{x}_i, \end{cases} \tag{3.1.21}$$

for $i = 1, 2, \ldots, n$ and with

$$\check{x}_i \leftarrow \max\left(x_i - \delta, \rho_{\min}\right)$$
$$\hat{x}_i \leftarrow \min\left(x_i + \delta, 1\right)$$

with δ a prescribed positive move limit and $x_{i_{\min}} = \rho_{\min}$ for all i. Thus, we have a simple and effective method of determining the updated (new) design variables, $\boldsymbol{x}(\lambda)$.

3.1.2.2 Exponential subproblem

As before, but using the exponential approximation (3.1.7), the objective function of the dual approximate subproblem (3.1.18) has the form

$$\gamma^{\{k\}}(\lambda) = f(\boldsymbol{x}^{\{k\}}) + \sum_{i=1}^{n} \left(\frac{\partial f(\boldsymbol{x}^{\{k\}})}{\partial x_i} \right) \left(\frac{x_i^{\{k\}}}{a_i^{\{k\}}} \right) \left[\left(\frac{x_i(\lambda)}{x_i^{\{k\}}} \right)^{a_i^{\{k\}}} - 1 \right] + \frac{\lambda}{\nu_0} \sum_{i=1}^{n} \nu_e x_i(\lambda) - \bar{\nu}. \tag{3.1.22}$$

From the stationary conditions, we find

$$x_i^{1-a_i^{\{k\}}}(\lambda) = - \left((x_i^{\{k\}})^{1-a_i^{\{k\}}} \frac{\partial f(\boldsymbol{x}^{\{k\}})}{\partial x_i} \right) \Big/ \left(\lambda \frac{\partial g(\boldsymbol{x}^{\{k\}})}{\partial x_i} \right), \tag{3.1.23}$$

and introducing $\beta_i(\lambda) = x_i^{1-a_i^{\{k\}}}(\lambda)$, we obtain

$$x_i(\lambda) = \begin{cases} \check{x}_i & \text{if} \quad \beta_i^{1/(1-a_i^{\{k\}})}(\lambda) \leq \check{x}_i, \\ \beta_i^{1/(1-a_i^{\{k\}})}(\lambda) & \text{if} \quad \check{x}_i \leq \beta^{1/(1-a_i^{\{k\}})}(\lambda) \leq \hat{x}_i, \\ \hat{x}_i & \text{if} \quad \beta_i^{1/(1-a_i^{\{k\}})}(\lambda) \geq \hat{x}_i, \end{cases} \tag{3.1.24}$$

for $i = 1, 2, \ldots, n$ and with

$$\begin{aligned} \check{x}_i &\leftarrow \max\left(x_i - \delta, \rho_{\min}\right) \\ \hat{x}_i &\leftarrow \min\left(x_i + \delta, 1\right) \end{aligned}$$

with δ a prescribed positive move limit and $x_{i_{\min}} = \rho_{\min}$ for all i, as for the reciprocal approximation above. The exponent $a_i^{\{k\}}$ is restricted to the range $a_{\min} \leq a_i^{\{k\}} \leq \epsilon_e < 0$, with $\epsilon_e < 0$ to ensure that the approximate primal subproblem is convex. Limit $a_{\min}$ may be selected rather arbitrarily as its purpose is to prevent very large negative exponents. That said, many exponents are expected to be in the vicinity of -1 anyway. For the first iteration ($k = 0$) we set $a_i^{\{0\}} = -1$, *i.e.*, we start with a reciprocal approximation.

The exponential approximation is expected to be more accurate then the reciprocal approximation since it has variable curvatures (it uses function evaluations in two points), whereas the reciprocal approximation has a fixed curvature (only one point is used), see Groenwold and Etman (2008) as well.

3.1.3 Summary of the dual SAO method

The SAO method can be briefly summarised in the following steps:

- Construct a local linear approximate primal problem of the original ('real') problem.
- Substitute the linear variables with intervening variables.
- Replace the intervening variables with either reciprocal or exponential variables (other types, like quadratic variables, are also possible).
- Construct the dual problem of the latter using the Falk dual if the problem is *strictly convex and separable*.
- Solve the Falk dual optimisation problem.

3.2 Optimality criterion (OC) method

Another popular approach for solving the 'classic' topology optimisation problem is the OC method, which is an expression of the KKT conditions at the optimality of the Lagrangian of (2.3.1). The term 'OC' is used loosely here as it *also* refers the heuristic updating scheme for the design variables (see below). For brevity's sake, we show the basics of the OC methods here, for a detailed overview, see for example Bendsøe (1989).

The heuristic updating scheme for the design variables, as previously presented by Bendsøe (1995), is formulated as

$$x_i^{\text{new}} = \begin{cases} \check{x}_i & \text{if} \quad x_i B_i^{\eta} \leq \check{x}_i, \\ x_i B_i^{\eta} & \text{if} \quad \check{x}_i \leq x_i B_i^{\eta} \leq \hat{x}_i, \\ \hat{x}_i & \text{if} \quad x_i B_i^{\eta} \geq \hat{x}_i, \end{cases} \tag{3.2.1}$$

for $i = 1, 2, \ldots, n$ and with

$$\begin{aligned} \check{x}_i &\leftarrow \max\left(x_i - \delta, \rho_{\min}\right) \\ \hat{x}_i &\leftarrow \min\left(x_i + \delta, 1\right) \end{aligned}$$

with δ a prescribed positive move limit, $\eta(= 1/2)$ is a numerical damping coefficient (introduced to overcome the oscillatory behaviour when (3.2.1) is solved) and $x_{i_{\min}} = \rho_{\min}$ for all i. Equation (3.2.1) is then solved iteratively until some stopping criteria is satisfied, *e.g.*, until $\| \boldsymbol{x}^{\text{new}} - \boldsymbol{x} \| < \epsilon$. The B_i are found from the stationary conditions (OC) as

$$B_i = -\left(\frac{\partial f(\boldsymbol{x})}{\partial x_i}\right) \Big/ \left(\lambda \frac{\partial g(\boldsymbol{x})}{\partial x_i}\right). \tag{3.2.2}$$

The λ may be found using any suitable 1D algorithm, *e.g.*, bisectioning or golden section. The derivatives of compliance are obtained as

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i} = -p x_i^{p-1} \boldsymbol{q}_i \boldsymbol{K}_e \boldsymbol{q}_i, \tag{3.2.3}$$

if we assume that the loads $\boldsymbol{r}$ are independent of the design variables. The derivatives of the constraint are obtained as

$$\frac{\partial g(\boldsymbol{x})}{\partial x_i} = \frac{v_i}{v_0} \tag{3.2.4}$$

if a mesh of equally sized elements are used. For minimum compliance, we set $\eta = 0.5$; this completes the simple and efficient approach of Bendsøe.

3.3 The relationship between SAO and OC

As shown by Groenwold and Etman (2008), a rudimentary dual SAO method based on *exponential* intervening variables and the 'direct' OC method, combined with the heuristic updating scheme presented by Bendsøe, yields identical optimisation methods, as shown below.

3.3.1 Equivalence

We start by rewriting (3.2.1) in the following form,

$$x_i^{\text{new}}(\lambda) = \begin{cases} \check{x}_i & \text{if} \quad B_i(\lambda) \leq \check{x}_i^{1/\eta}, \\ B_i(\lambda) & \text{if} \quad \check{x}_i^{1/\eta} \leq B_i(\lambda) \leq \hat{x}_i^{1/\eta}, \\ \hat{x}_i & \text{if} \quad B_i(\lambda) \geq \hat{x}_i^{1/\eta}, \end{cases} \tag{3.3.1}$$

where we have simply eliminated (by division) the x_i's and we have raised the conditional parts to the power $1/\eta$ (note that $x_i > 0$ and $B_i > 0$). Also, we need to redefine (3.2.2) as

$$B_i(\lambda) = -\left(x_i^{1/\eta}\frac{\partial f(\boldsymbol{x})}{\partial x_i}\right) \Big/ \left(\lambda\frac{\partial g(\boldsymbol{x})}{\partial x_i}\right). \tag{3.3.2}$$

From Section 3.1 we rewrite (3.1.24) as

$$x_i^{\text{new}}(\lambda) = \begin{cases} \check{x}_i & \text{if} \quad \beta_i(\lambda) \le \check{x}_i^{(1-a_i)}, \\ \beta_i(\lambda) & \text{if} \quad \check{x}_i^{(1-a_i)} \le \beta(\lambda) \le \hat{x}_i^{(1-a_i)}, \\ \hat{x}_i & \text{if} \quad \beta_i(\lambda) \ge \hat{x}_i^{(1-a_i)}, \end{cases} \tag{3.3.3}$$

with

$$\beta_i(\lambda) = -\left(x_i^{(1-a_i)}\frac{\partial f(\boldsymbol{x})}{\partial x_i}\right) \Big/ \left(\lambda\frac{\partial g(\boldsymbol{x})}{\partial x_i}\right). \tag{3.3.4}$$

We have omitted the superscript $\{k\}$ for the sake of notational clarity and to be consistant with (3.2.1). Comparing (3.3.1) and (3.3.2) with (3.3.3) and (3.3.4), it is clear that the two update methods are indeed identical, with

$$1/\eta = 1 - a_i. \tag{3.3.5}$$

If we substitute the popular choice for the damping factor $\eta = 0.5$, we find $a_i = -1$, being equivalent to SAO with reciprocal intervening variables (see Sections 3.1.2.1 and 3.1.2.2).

3.4 Grey-scale filter (GSF)

As shown by Groenwold and Etman (2009), the following heuristic method may be used to obtain the equivalent of volumetric penalisation into an OC-like statement, or equivalently, SAO, without the difficulties associated with a concave volume constraint function:
Use a standard SIMP to define the first and second density measures, which are introduced as $\mu_{1i}(x_i) = x_i^p$ and $\mu_{2i}(x_i) = x_i^d, (0 < d \le 1)$ respectively. Hence, the objective function can be represented as

$$f(\boldsymbol{x}) = \sum_{i=1}^{n} \mu_{1i}(x_i)\boldsymbol{q}_i^T \boldsymbol{K}_e \boldsymbol{q}_i$$

and the volume constraint as

$$g(\boldsymbol{x}) = \frac{v_e}{v_0}\sum_{i=1}^{n} \mu_{2i}(x_i) - \bar{v} \le 0.$$

In the classic SIMP method, $d = 1$ and we have $\mu_{2i}(x_i) = x_i$.

This method results in familiar SAO subproblems, each consisting of a strictly convex objective function, subject to a linear constraint. Hence, the subproblems are strictly convex. In addition, using exponential intervening variables,

$$y_i(x_i) = x_i^{a_i} = x_i^{1-1/\eta}, \tag{3.4.1}$$

with $0 < \eta < 1$, results in approximate SAO objective functions that exhibit strict monotonicities with respect to the design variables x_i. In light of this, we may modify the 'standard' OC updating method (3.2.1) as follows:

$$x_i^{\text{new}} = \begin{cases} \check{x}_i & \text{if} \quad \Gamma(x_i B_i^\eta) \leq \check{x}_i, \\ \Gamma(x_i B_i^\eta) & \text{if} \quad \check{x}_i \leq \Gamma(x_i B_i^\eta) \leq \hat{x}_i, \\ \hat{x}_i & \text{if} \quad \Gamma(x_i B_i^\eta) \geq \hat{x}_i, \end{cases} \tag{3.4.2}$$

where $\Gamma(x_i B_i^\eta) = (x_i B_i^\eta)^q$ (with $q \geq 1$) indicates a non-linear grey-scale filter (GSF) or intermediate density filter (Groenwold and Etman, 2009). The specific form of the filter was partly chosen to ensure that a value of $q = 1$ will result in the 'standard' OC update method. We make the substitution $\Gamma(x_i B_i^\eta) = (x_i B_i^\eta)^q$, and replace (3.4.2) with the new update formula

$$x_i^{\text{new}} = \begin{cases} \check{x}_i & \text{if} \quad (x_i B_i^\eta)^q \leq \check{x}_i, \\ (x_i B_i^\eta)^q & \text{if} \quad \check{x}_i \leq (x_i B_i^\eta)^q \leq \hat{x}_i, \\ \hat{x}_i & \text{if} \quad (x_i B_i^\eta)^q \geq \hat{x}_i, \end{cases} \tag{3.4.3}$$

where (3.4.3) is the GSF. The GSF does not destroy the convexity of the subproblems, it merely modifies the 'output' of the updating rule. For the proof, the reader is again referred to Groenwold and Etman (2009).

The working principle of the filter is best illustrated by means of Figure 3.1: A value of $x_i B_i^\eta = 0.4$ and $q = 1.0$ results in a (linear) volume contribution of 0.4, whereas $q = 3.0$ results in a contribution of less than 0.1. GSF therefore underestimates intermediate densities, driving them to values that represent void (≈ 0) densities.

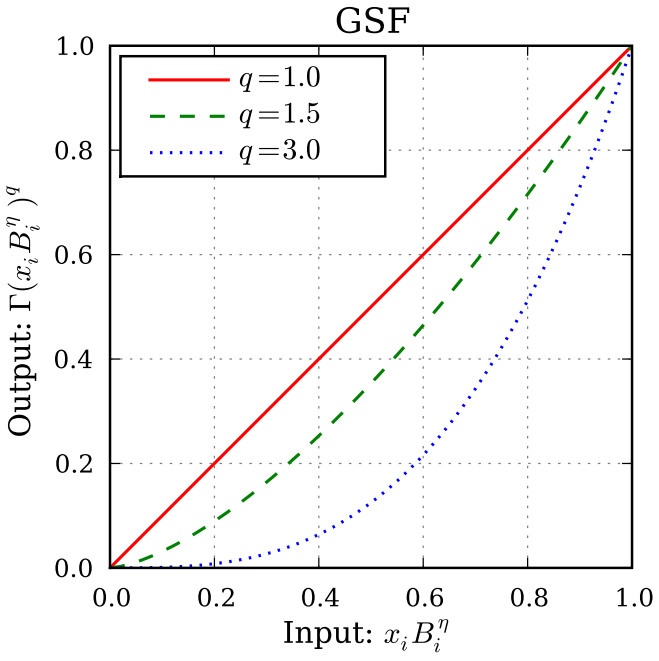


Figure 3.1: Working principle of the grey-scale filter, with $\eta = 0.5$.

The intermediate density variables are driven towards void density values during "inner iterations", *i.e.*, when the Lagrange multiplier λ is updated. Hence, the idea is to underestimate all intermediate density material. For example, design variables with values of $x_i B_i^{\eta} \geq \check{x}_i$ will result in solid density design variables. Note that it frequently occurs that $x_i B_i^{\eta} \gg \check{x}_i$ in simulations of realistic structures, which means that the filter is really only 'filtering' variables with values that range between ≈ 0 and 1 Groenwold and Etman (2009).

3.5 Diagonal quadratic approximations

In structural optimisation, exponential approximations may be written as accurate two-point approximations if the structural response is predominantly monotonically decreasing, thereby rendering them superior to reciprocal approximations. Exponential approximations are however not problem free: If dual methods are used, it is not possible to construct simple *analytical* relationships between the primal and dual variables in dual convex methods if multiple constraints are present, so crucially needed in order to employ the Falk dual. Quadratic approximations is a way to overcome this problem (Groenwold and Etman, 2007).

We can approximate the reciprocal or exponential approximations with quadratic approximations (Groenwold *et al.*, 2008; Groenwold and Etman, 2007). We will herein use a separable *diagonal quadratic* function of the form

$$\tilde{f}(\boldsymbol{x}) = f(\boldsymbol{x}^{\{k\}}) + \sum_{i=1}^{n} \left(\frac{\partial f(\boldsymbol{x}^{\{k\}})}{\partial x_i} \right) (x_i - x_i^{\{k\}}) + \frac{1}{2} \sum_{i=1}^{n} c_{2i}^{\{k\}} \left(x_i - x_i^{\{k\}} \right)^2 . \tag{3.5.1}$$

This incomplete series expansion (ISE) is simply a second order (quadratic) Taylor series expansion without off-diagonal terms. Due to the computational effort of calculating and/or storing the off-diagonal (coupling) terms, we have retained only the *diagonal* Hessian terms, $c_{2i}^{\{k\}}$. $\tilde{f}(\boldsymbol{x})$ is convex if $c_{2i}^{\{k\}} \geq 0 \; \forall \; i$ (it is strictly convex if the inequality holds for all i).

3.5.1 Approximate diagonal Hessian terms

Using the second order Taylor series approximation (3.5.1) to the *exponential* approximation (3.1.7), we can derive the approximate diagonal Hessian terms in 3.5.1. The first and second partial derivatives of (3.1.7) with respect to x_i are

$$\frac{\partial \tilde{f}_E}{\partial x_i} = \left(\frac{x_i}{x_i^{\{k\}}} \right)^{a_i^{\{k\}}-1} \left(\frac{\partial f}{\partial x_i} \right)^{\{k\}} , \tag{3.5.2}$$

and

$$\frac{\partial^2 \tilde{f}_E}{\partial x_i^2} = \left(a_i^{\{k\}} - 1 \right) (x_i)^{a_i^{\{k\}}-2} \left(x_i^{\{k\}} \right)^{1-a_i^{\{k\}}} \left(\frac{\partial f}{\partial x_i} \right)^{\{k\}} . \tag{3.5.3}$$

Now, in the current point x_i, the above reduce to

$$\frac{\partial \tilde{f}_E}{\partial x_i} = \left(\frac{\partial f}{\partial x_i} \right)^{\{k\}} , \tag{3.5.4}$$

and

$$\frac{\partial^2 \tilde{f}_E}{\partial x_i^2} = c_{2i}^{\{k\}} = \frac{\left(a_i^{\{k\}} - 1 \right)}{x_i^{\{k\}}} \left(\frac{\partial f}{\partial x_i} \right)^{\{k\}} , \tag{3.5.5}$$

respectively, where the unknown exponents $a_i^{\{k\}}$ were obtained as (3.1.10) in Section 3.1.1.3.

For problems that are not self-adjoint, *e.g.*, the mechanism synthesis problem, the derivatives $\partial f / \partial x_i$ may become positive. A simple way to overcome this problem is to introduce the following absolute value operator:

$$c_{2i}^{\{k\}} = \frac{\left(1 - a_i^{\{k\}}\right)}{x_i^{\{k\}}} \left| \frac{\partial f}{\partial x_i} \right|^{\{k\}} = \frac{1}{\eta x_i^{\{k\}}} \left| \frac{\partial f}{\partial x_i} \right|^{\{k\}}. \tag{3.5.6}$$

To construct the quadratic Taylor series expansion to the *reciprocal* approximation, we simply set $a_i^{\{k\}} = -1$ (equivalent to $\eta = 0.5$) and get

$$c_{2i}^{\{k\}} = \frac{2}{x_i^{\{k\}}} \left| \frac{\partial f}{\partial x_i} \right|^{\{k\}}. \tag{3.5.7}$$

3.6 Which approximation to use?

There is, unfortunately, no general rule to determine which approximation to use for a specific type of problem. It is, of course, subjective whether a solution is 'best', *e.g.*, a specific result may seem closer to one person's intuition than another person's. That said, for unpenalised problems ($p = 1$), the exponential approximation converges faster to the same final value as the reciprocal approximation.

It is probably easiest (and advisable) to solve a few small problems to determine which approximation gives the 'best' result for a specific problem (*e.g.*, Section 6.3 illustrates a similar approach).

Chapter

4

Finite elements with penalised equilibrium

In this chapter we consider the formulation of a bending-exact element (in terms of displacement) for compliance minimisation in 2D and 3D respectively.

4.1 Background

The standard Q4 element is notorious for its low accuracy in bending dominated conditions. The higher order modes of straining or deformation result in shear locking (see *e.g.*, Cook (1995)) of the elements in a finite element mesh. This in turn results in inaccurate displacement values for the nodes and since the objective function for compliance minimisation is a function of displacement, we may reason that results obtained for bending dominated problems is influenced by this inaccuracy. The question is whether we can obtain lower objective function values if we use bending-exact elements for such problems, instead of the standard Q4 and H8 elements for 2D and 3D problems respectively.

4.2 Summary of 2D element formulation

Section 4.2.1 is taken from De Klerk and Groenwold (2004) and is included here for completeness (if slightly adapted for the current text). In section 4.3, we elaborate and improve on the results in De Klerk and Groenwold (2004) by presenting the symbolic computation of the critical and the optimal parameters. Also, we include the wxMaxima session scripts we use to compute these parameters, see Appendix D.

4.2.1 Penalised equilibrium in assumed stress elements

Assumed stress hybrid elements are renowned for their accurate stress solutions, due to the fact that the stress and displacement trial functions are selected independently. For a 2D 4-node quadrilateral ($Q4$) membrane element, the displacements $\boldsymbol{u}$ are typically interpolated as

$$\boldsymbol{u} = \boldsymbol{N}\boldsymbol{q}, \tag{4.2.1}$$

with $\boldsymbol{N}$ the bi-linear Lagrangian interpolation functions, and $\boldsymbol{q}$ the (unknown) elemental nodal displacements. The stress $\boldsymbol{\sigma}$ is interpolated as

$$\boldsymbol{\sigma} = \boldsymbol{T}\boldsymbol{P}\boldsymbol{\beta}, \tag{4.2.2}$$

with $\boldsymbol{P}$ the stress interpolation matrix, $\boldsymbol{T}$ an optional transformation or constraint matrix, and $\boldsymbol{\beta}$ the (unknown) elemental stress parameters. Selecting $\boldsymbol{T}$ and $\boldsymbol{P}$ is not straight forward; there are no unique optimal formulations. $\boldsymbol{P}$ depends on issues like completeness in the Cartesian coordinate system, which may compete with the limiting principle of Fraeijs du Veubeke (1965). Nevertheless, for the $Q4$ element at least, previous experience has shown that a number of formulations are accurate and robust. Typically, 5 β parameters are used, with

$$\boldsymbol{P} = \begin{bmatrix} 1 & 0 & 0 & \eta & 0 \\ 0 & 1 & 0 & 0 & \xi \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}, \tag{4.2.3}$$

and ξ and η the natural coordinates. The finite elements in ToPy (the software we will develop herein) does not make use of the isoparametric formulation, and we therefore substitute ξ and η with x/a and y/b, where a and b are the half-lengths of the sides of a square FE element, *viz.*, $a = b$ (see Section 5.4 for a discussion of ToPy's limitations).

Accurate stress solutions in the sense of the energy norm do not necessarily imply pointwise accurate stress predictions within elements (Wu and Cheung, 1995). Indeed, most hybrid elements only yield accurate stress predictions at the element centroid, while stress predictions at element edges and in particular element nodes, can be highly inaccurate.

To improve the accuracy of point wise predictions in hybrid elements, a number of formulations have been proposed. This includes pre- and post-treatment proposed by Wu and Cheung (1995), with post-treatment simpler and superior. Post-treatment via penalised equilibrium is outlined as follows:

While distributed body forces may induce important loads on a structure, they can usually be ignored on the element level in stress calculations. Hence element equilibrium is written as

$$\partial\boldsymbol{\sigma} = \boldsymbol{D}^T\boldsymbol{\sigma} = \boldsymbol{0} \text{ in } \Omega \tag{4.2.4}$$

with $\boldsymbol{D}$ the 2D differential operator and Ω the elemental domain. Enforcement of (4.2.4) in the variational formulation yields a functional $\boldsymbol{\Pi}^*$ of the form

$$\boldsymbol{\Pi}^*(\boldsymbol{u},\boldsymbol{\sigma}) \;=\; \boldsymbol{\Pi}(\boldsymbol{u},\boldsymbol{\sigma}) - \alpha\int(\partial\boldsymbol{\sigma})^T(\partial\boldsymbol{\sigma})d\Omega, \quad \alpha \gg 0, \tag{4.2.5}$$

where $\boldsymbol{\Pi}(\boldsymbol{u},\boldsymbol{\sigma})$ represents the potential of the *Hellinger-Reissner* principle. Using matrix notation, and for the sake of convenience setting $\alpha \leftarrow \alpha/2E$, the potential of the assumed stress elements becomes

$$\boldsymbol{\Pi}^*(\boldsymbol{u},\boldsymbol{\sigma}) \;=\; \boldsymbol{\beta}^T\boldsymbol{G}\boldsymbol{q} - \frac{1}{2}\boldsymbol{\beta}^T\left(\boldsymbol{H} + \frac{\alpha}{E}\boldsymbol{H}_p\right)\boldsymbol{\beta} - \boldsymbol{q}^T\boldsymbol{r} \tag{4.2.6}$$

with

$$\boldsymbol{G} = \int_\Omega \boldsymbol{P}^T\boldsymbol{B}\; d\Omega, \tag{4.2.7}$$

$$\boldsymbol{H} = \int_{\Omega} \boldsymbol{P}^T \boldsymbol{C}^{-1} \boldsymbol{P} \; d\Omega, \tag{4.2.8}$$

$$\boldsymbol{r} = \int_{\Omega} \boldsymbol{N}^T \boldsymbol{f} \; d\Omega, \tag{4.2.9}$$

and

$$\boldsymbol{H}_p = \int_{\Omega} (\partial \boldsymbol{P})^T (\partial \boldsymbol{P}) d\Omega, \tag{4.2.10}$$

where $\boldsymbol{P}$ is now understood to represent the matrix product $\boldsymbol{TP}$. From the stationary condition, the elemental stress parameters may be recovered as

$$\boldsymbol{\beta} = \left(\boldsymbol{H} + \frac{\alpha}{E} \boldsymbol{H}_p \right)^{-1} \boldsymbol{G} \; \boldsymbol{q}, \tag{4.2.11}$$

while the force-displacement relationship is obtained as

$$\boldsymbol{G}^T \left(\boldsymbol{H} + \frac{\alpha}{E} \boldsymbol{H}_p \right)^{-1} \boldsymbol{G} \; \boldsymbol{q} = \boldsymbol{r}, \tag{4.2.12}$$

or

$$\boldsymbol{K}\boldsymbol{q} = \boldsymbol{r}. \tag{4.2.13}$$

4.2.1.1 Allowable values of α for hybrid elements

In the penalty formulation, we note that $\alpha \gg 0$ is required. Wu and Cheung (1995) merely report that results converge for $\alpha/E \geq 10000$, but they present no comments on element accuracy in the limit of enforcing $\alpha \gg 0$. They then suggest that the equality represented in this relationship be used for practical implementations.

For elements of regular geometry, penalised equilibrium has no detrimental influence. (A 'regular' geometry is understood to be square or rectangular). However, for elements of irregular geometry, the stiffness associated with the higher order deformation modes completely vanishes as $\alpha \to \infty$.

In an assembly of elements, even very high values of α are normally adequate to prevent communicable mechanisms (spurious modes); in addition, the higher order modes of course have no influence in passing the patch test. Hence, connected elements are unconditionally convergent. Nevertheless, for single elements (or a very small number of elements), the higher order modes may vanish for all practical purposes, resulting in very low stiffness, and therefore significant over-displacement, in bending dominated problems.

4.3 Displacement based element

Even though the enforcement of $\partial \boldsymbol{\sigma} = \boldsymbol{0}$ in Ω seems difficult for the $Q4$ element, it may be attempted to soften the higher order deformation modes by means of introducing elemental parameters, expressed in terms of the element equilibrium equations.

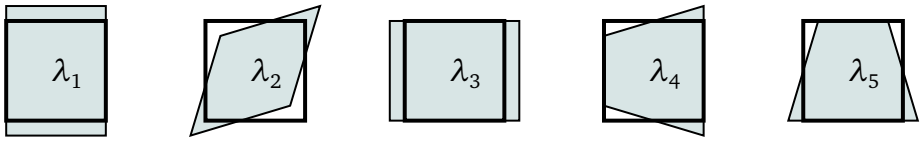


Figure 4.1: Straining modes of a square $Q4$ element.

The straining or deformation modes of a square Q4 element are depicted in Figure 4.1. The (very stiff) higher order modes are to be blamed for the poor performance of this element in bending. For the element, the stresses are obtained from

$$\boldsymbol{\sigma} = \boldsymbol{C}\boldsymbol{\epsilon} = \boldsymbol{C}\boldsymbol{B}\boldsymbol{q}. \tag{4.3.1}$$

Hence

$$\partial\boldsymbol{\sigma} = \partial(\boldsymbol{C}\boldsymbol{B}\boldsymbol{q}) = \partial(\boldsymbol{C}\boldsymbol{B})\boldsymbol{q}^{\ddagger} = \bar{\boldsymbol{B}}\boldsymbol{q} \tag{4.3.2}$$

The potential energy is then constructed as

$$\boldsymbol{\Pi}^*(\boldsymbol{u}) = \frac{1}{2}\int_\Omega \boldsymbol{\epsilon}^T\boldsymbol{C}\boldsymbol{\epsilon}\; d\Omega - \alpha\int_\Omega (\boldsymbol{C}\boldsymbol{\epsilon})^T(\boldsymbol{C}\boldsymbol{\epsilon})\; d\Omega - \int_\Omega \boldsymbol{u}^T\boldsymbol{f}\; d\Omega \tag{4.3.3}$$

with

$$\boldsymbol{\epsilon} = \partial\boldsymbol{u} = \partial\boldsymbol{N}\boldsymbol{q}. \tag{4.3.4}$$

Interpolating for $\boldsymbol{u}$ as

$$\boldsymbol{u} = \boldsymbol{N}\boldsymbol{q}, \tag{4.3.5}$$

(4.3.3) becomes

$$\boldsymbol{\Pi}^*(\boldsymbol{u}) = \frac{1}{2}\boldsymbol{q}^T\int_\Omega \boldsymbol{B}^T\boldsymbol{C}\boldsymbol{B}\; d\Omega\; \boldsymbol{q}\; - \alpha\boldsymbol{q}^T\int_\Omega \bar{\boldsymbol{B}}^T\bar{\boldsymbol{B}}\; d\Omega\; \boldsymbol{q} - \boldsymbol{q}^T\boldsymbol{r}. \tag{4.3.6}$$

The *force-displacement* relationship for the *new* $Q4(\hat{\alpha})$ element, which we call $Q4\alpha5\beta$, is then obtained as

$$(\boldsymbol{K} - \alpha E\bar{\boldsymbol{K}})\boldsymbol{q} = \boldsymbol{r}, \tag{4.3.7}$$

where $\alpha \leftarrow \alpha E/2$. Compare this to the normal force-displacement relationship for the $Q4$ element,

$$\boldsymbol{K}\boldsymbol{q} = \boldsymbol{r}. \tag{4.3.8}$$

As we can see, the difference is the introduction of $\alpha E\bar{\boldsymbol{K}}$, which is introduced in order to 'soften' higher order modes of the $Q4$ element.

Three linearly independant rigid-body motions exist for a 2D element, namely linear movement in each of the two Cartesian directions (*e.g.*, x and y), and rotation *in* the plane. For a 3D solid element, six rigid-body motions exist (three linear and three rotational motions

[‡]There is slight typo in the original text, we have corrected it here.

about x, y and z, respectively). By checking the number of zero eigenvalues of the element stiffness matrix, we can determine if there are as many rigid body motions as we require (Cook *et al.*, 2002, Section 8.10).

For a fully constrained finite element, *i.e.*, no rigid body motion is possible, the $\boldsymbol{K}$ matrix is positive-definite, which implies that all the eigenvalues are positive. On the other hand, for an element that is not fully constrained, the $\boldsymbol{K}$ matrix is positive-semidefinite, which implies that some eigenvalues are positive and others are zero ((Hughes, 2000, Section 1.9), Cook (1995, Section 1.6)).

For elements with 'small' stiffness matrices, typically those of 2D elements, we can calculate the eigenvalues symbolically quite easily (using software like Mathematica (v. 6) or wxMaxima (v. 0.7.1)). The critical α value (α_{crit}) of the $Q4(\hat{\alpha})$ element stiffness matrix is that value of α which introduces an extra (spurious) zero energy mode (an extra zero eigenvalue). Below are the eigenvalues for $Q4(\hat{\alpha})$, obtained using wxMaxima:

$$[-\frac{E}{\nu-1}, \frac{\alpha - 3E^3(\nu-1) + a^2\left(2\nu^2 E - 8\nu E + 6E\right)}{12a^2\left(\nu^3 - \nu^2 - \nu + 1\right)}, \frac{E}{\nu+1}, 0] \qquad [1,2,2,3]. \tag{4.3.9}$$

As we can see, there is only one eigenvalue that contains the α parameter, solving this eigenvalue for a square $Q4(\hat{\alpha})$ element with side lengths a, we get

$$\alpha_{\text{crit}} = \frac{2a^2\nu^2 - 8a^2\nu + 6a^2}{3(\nu+1)E^2}. \tag{4.3.10}$$

Any value smaller than α_{crit} is therefore permissable, however, α should be selected such that it is 'optimal' in some way. For bending dominated problems it makes sense to set the eigenvalues of $Q4(\hat{\alpha})$ equal to those of the assumed stress stress element, 5β-NC (see *e.g.* Di and Ramm (1994)). The eigenvalues of the 5β-NC element are (also using wxMaxima)

$$[-\frac{E}{\nu-1}, \frac{E}{3}, \frac{E}{\nu+1}, 0] \qquad [1,2,2,3]. \tag{4.3.11}$$

Setting corresponding values equal to each other and solving for α gives

$$\alpha = -\frac{2a^2(\nu-1)\left(2\nu^2 - \nu + 1\right)}{3(\nu+1)E^2}. \tag{4.3.12}$$

By inspection, it is easy to see that $\alpha < \alpha_{\text{crit}}$, satisfying our requirement. We now substitute this value into (4.3.7) which gives us the bending-exact element stiffness matrix, $Q4\alpha5\beta$. This element as well as the 5β-NC element is implemented in ToPy as `Q4a5B` and `Q5B` respectively.

4.3.1 3D element formulation

Unfortunately, we were not able to apply the same symbolic method that was used for the 2D element in 3D, our symbolic software (wxMaxima) ran out of memory. Resorting to highly specialised proprietary symbolic software (Mathematica 6 for GNU/Linux) did not help either. In the latter case the software was able to solve the problem, but the result was useless, consisting roughly of four A4 pages of symbolic output...

The formulation requires the stiffness matrix of the 3D version of the 2D 5β-NC hybrid stress element, namely 18β-NC (Di and Ramm, 1994); we have made the 18β-NC element available in ToPy as `H18B`.

4.3.1.1 Testing of 3D elements

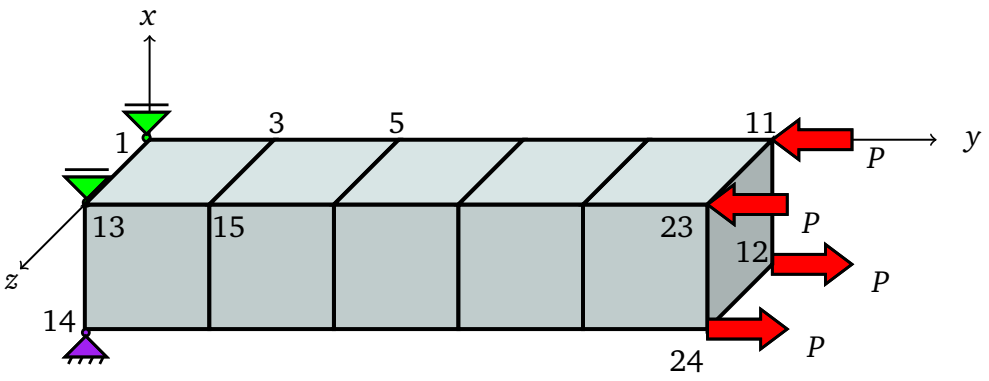


Figure 4.2: Slender 3D beam in pure bending.

Note that for the sake of brevity, we do not include testing of the 2D elements, even though they were tested using the equivalent approach in two-dimensional space. The test routines are included in the source code.

To test the 18β-NC element, we set up a simple pure bending problem as shown in Figure 4.2, where the nodal loads $P = 0.5$ each (*Poisson's ratio* $\nu = \frac{1}{3}$). The analytical result for the deflection of the tip is obtained using the well known formula

$$y_{tip} = \frac{ML^2}{2EI},$$

where $M = 4 \times P \times 0.5 = 1.0$, $I = \frac{1 \times 1^3}{12}$, $L = 5.0$ and $E = 1.0$, which gives

$$y_{tip} = \frac{1 \times 5^2}{2 \times 1 \times 1/12} = 150.0.$$

The numerical output given by ToPy's FEA routine is shown in Table 4.1

Element type	Node nr. 11	Node nr. 12	Node nr. 23	Node nr. 24
$H8$	95.0442	94.283	95.044	94.282
18β-NC	149.842	148.844	149.844	148.844

Table 4.1: Comparison of 3D element performance in pure bending – displacement in the x direction.

Clearly, the 18β-NC element performs much better than the 'standard' $H8$ element for predicting displacements in pure bending situations. The 3D 'heat' finite element were tested in a similar manner.

All finite elements were tested to ensure that they yield results consistent with theory.

Chapter

5

Implementation in Python

In this chapter we briefly cover the implementation details of ToPy (the code), instead of (perhaps) boring the reader with lines and lines of code. For the complete source code, refer to Appendix A.

5.1 Software design

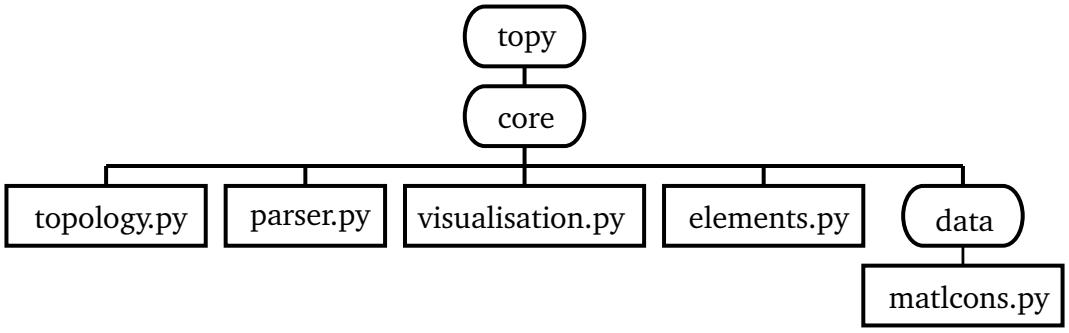


Figure 5.1: Software tree of ToPy (partial).

Figure 5.1 illustrates the main structure of the ToPy package (other directories are omitted for clarity). The rectangular boxes refer to Python files and the rounded boxes to normal file directories. In the sections that follow, we explain the purpose of each Python file.

5.1.1 Problem setup

Instead of defining a topology optimisation problem by changing source code, we create a very simple input file that is *parsed* (to resolve into component parts). The format of the input file, called a ToPy problem definition file (or TPD file for short), can be found in Appendix A.

Once the TPD file is parsed, it is converted to a Python *dictionary* that is a data structure in which you can refer to each *value* by means of a *key* (name). One of the advantages of

using a dictionary is that order is not important and different data types can be mixed. For example, a dictionary can contain strings, numbers, arrays, in fact, even other dictionaries.

Once the dictionary is created, some rudimentary checks are made, for instance, ToPy will pick up typing errors, rigid body motion of the domain, inconsistent loading (if the length of the load vector differs from the vector of nodes to be loaded), and a few other possible problems. In any event, ToPy will most likely not run if a problem definition is incorrect, but it is of course impossible to cater for all possible errors. Finally, we use the dictionary to set up the optimisation problem.

In ToPy, the dictionary contains all the information necessary to setup and solve the problem, it contains values such as the load vector, element type, number of iterations to run, the type of approximation to use, the problem name, stiffness matrix, *etc*.

All of this is handled by **parser.py**.

5.1.2 Topology optimisation

The optimisation consists of the following methods (functions):

5.1.2.1 FEA

ToPy utilises a very efficient and fast sparse solver to perform the FEA. We use a direct solver for 2D problems (SuperLU) and an iterative solver for 3D problems (preconditioned conjugate gradient method), since the latter type of problem either takes a very long time to solve with the direct solver, or ToPy runs out of memory. The source code can be changed to use other solvers quite easily, since the whole FEA routine is only 55 lines of code (26 lines excluding comments). The solvers are called via Pysparse.

The FEA routine is robust in the sense that it will terminate elegantly if a solution is not found or if a prescribed number of iterations is exceeded. These parameters can be set in the source code, nevertheless, we solved > 3 million DoF (slightly more than one million finite elements in 3D) problems with the default settings.

5.1.2.2 Filtering of the design sensitivities

This is Sigmund's heuristic approach, however, our version is partially vectorised and *significantly* faster than the version in the 99-line code. The filter modifies the sensitivity of a specific element based on a weighted average of the sensitivities of the surrounding elements, within a fixed radius.

5.1.2.3 Sensitivity analysis

We determine the objective function value and perform sensitivity analysis (find the derivatives of objective function). We also check what type of problem is solved and apply the suitable algorithm for compliance, heat or mechanism synthesis problems automatically.

5.1.2.4 Updating of design variables and grey-scale filtering

This is where we implement the reciprocal, exponential and diagonal quadratic approximations, as well as the grey-scale filter. We use GSF with with diagonal quadratic approximations even for non-adjoint problems, like mechanism synthesis, which require changes to the updating rule. This has not been done before.

The grey-scale filter, as opposed to Sigmund's heuristic filter, actually manipulates the output of the updating rule by effectively penalising all intermediate or grey material. All of the above is performed by **topology.py**. The interested reader may refer to Appendix A for details.

5.1.3 Visualisation

Our visualisation strategy is simple; instead of printing images to the screen in real time, we create images for 2D problems and geometry files for 3D problems, off-screen. This strategy has an obvious speed advantage, and we may use the data at a later stage to create animations that can be viewed/played in any web browser. See Appendix A for details on how to create animations.

Below are two examples of output produced from a 2D and 3D NumPy (numerical Python) array respectively, using the **visualisation.py** module:

Figure 5.2: 2d ToPy 'logo'

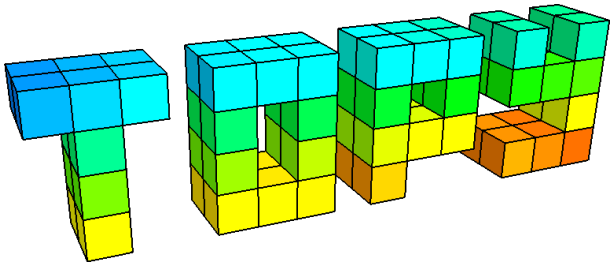


Figure 5.3: 3d ToPy 'logo'

5.2 Elements

The finite elements in ToPy are stored in binary format for speed, *i.e.*, the elements are not created each time a problem is run, it is instead simply loaded. Also, it is a very simple matter of creating and adding ones own elements: **matlcons.py** contains the material constants, and the 'data' directory contains individual Python files for each type of element that is available in ToPy. These can be used as templates to create new elements, and then one only need to add six lines of code to the **elements.py** file.

5.3 Customisation

Due to the design of Python it is very easy to customise ToPy – it is a trivial exercise to change code since no compiling is required, the changes are immediate and the program can be used as soon as the code is changed.

It is also possible to use ToPy from the command line inside a Python shell (like IDLE or better yet, IPython). This has the advantage that one can 'see' what is happening between optimisation iterations. The user may even change, say, the element type that is used between different iterations. It is also an excellent way to explore, test and implement different routines or functions. For more details, refer to Section A.3 in the Appendix.

5.4 Capabilities and limitations

5.4.1 Capabilities

The "out of the box" capabilities and features of ToPy, some of which have already been mentioned, are:

1. Three different problem types can be solved, both in 2D and 3D, namely:
   a) Minimum compliance,
   b) heat conduction and
   c) mechanism synthesis.
2. Implementation of a grey-scale filter to yield predominantly, or even purely, solid-void or black-and-white designs.
3. Continuation of the penalisation factor (p): specification of when to start continuation, the increment value, number of iterations to keep the new value constant after the increment and the maximum value of p.
4. Continuation of the extra penalisation factor (q) used in the GSF: specification of when to start continuation, the increment value, number of iterations to keep the new value constant after the increment and the maximum value of q.

5. Specification of convergence stop criteria value or number of iterations to run.
6. Specification of type of approximation, namely
   a) reciprocal,
   b) exponential (based on previous gradient information or a prescribed value) and
   c) diagonal quadratic approximation to one of the above to approximations.
7. ToPy will terminate after 250 iterations, as a safe guard against infinite loops. This value can easily be changed and set in the source code.

For further details of these and other featrures, see the source code in Appendix A.

5.4.2 Limitations

ToPy has some limitations. The most important ones include the following:

1. Only regularly shaped and square 2D (cubes in 3D) elements are implemented, *i.e.*, not isoparametric elements. This is partly so because the 99-line code only implements a regularly shaped $Q4$ element. The other reason is that it also gives a performance advantage since assembly of the global stiffness matrix is efficient: The elements are "hard coded" (integration to determine the element stiffness matrix was done analytically, stiffness matrices of elements are not determined during assembly of the global stiffness matrix).
2. Only single constraints can be handled, *e.g.*, the single constraint on volume.
3. Geometric non-linear problems cannot be solved (without modifying the source code).
4. Problems in dynamics cannot be solved (without modifying the source code).
5. Multiple load cases (although multiple simultaneous loads can be applied).

5.5 Performance

The results (see Section 6.6) demonstrated that ToPy can solve high dimensionality problems, but how does it compare to the 99-line code, and hence MATLAB (and MATLAB's own sparse module), in terms of speed?

We solve nine 'standard' Messerschmitt-Bölkow-Blohm (MBB) beam problems of increasing dimensionality. The first and smallest domain is $30 \times 10 = 300$ elements and the last and largest domain is $600 \times 200 = 120\,000$ elements. We run each problem for ten iterations and record the time for each problem. The problems are exactly the same and solved on the same computer.

Clearly, for all and specifically for large problems, the Python implementation (ToPy) is superior to MATLAB (version 2008a) in terms of speed (see Figure 5.4). This can be attributed to more efficient implementation of routines, especially the sensitivity filter, but one of the main reasons for the excellent performance is the use of Pysparse to solve the system of equations that arises from the FEA.

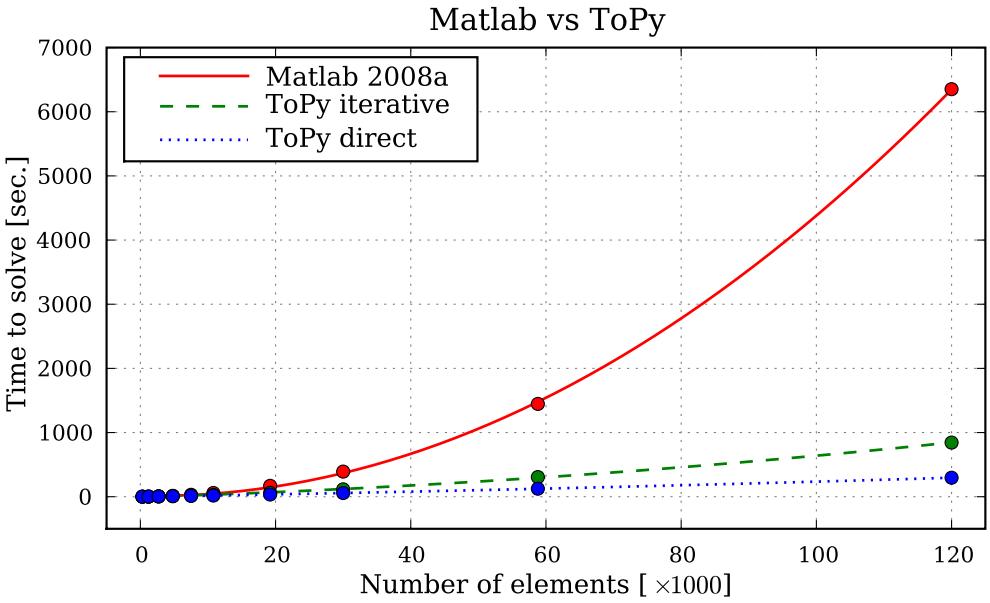


Figure 5.4: Performance comparison.

Chapter

6

Results

We firstly present 2D results for compliance minimisation, heat conduction and compliant mechanism synthesis to demonstrate that we obtain the same results as problems solved with Sigmund's MATLAB codes. (See Bendsøe and Sigmund (2004) for changes to the 99-line code in order to solve heat conduction and compliant mechanism synthesis problems.) MATLAB 5 was used to produce the results.

GSF is then presented by means of a high discretisation (mesh-refined) MBB problem since the effect of GSF is slightly easier to visualise in 2D because of ones familiarity with 'normal' results that contain a large amount of intermediate densities.

Next, we present a simple 3D minimum compliance problem solved with GSF and a range of fixed values for the damping factor, η (refer to Section 3.2 and Bendsøe and Sigmund (2004) for details). We compare the final objective function values to their corresponding damping factor values and then select the 'optimal' η value. We then proceed to use exponential intervening variables for the same problem and compare the result with our 'optimal' η. A few more complex (in terms of geometry, loading and constraints) problems are solved and presented, also using our 'optimal' η.

Finally, we present results for 3D heat and mechanism synthesis problems using diagonal quadratic approximations, as well as two "one million finite elements" solutions.

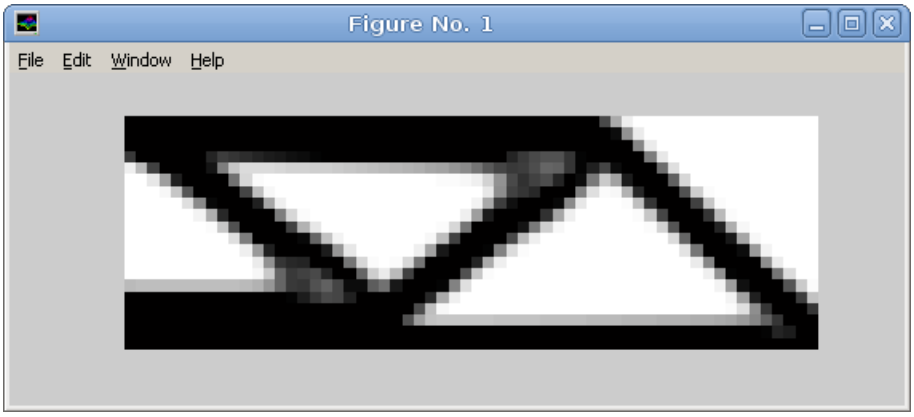
Installation and use of the software is discussed in Appendix A while the input files for some of the problems are listed in Appendix B.

6.1 'Standard' reference problems in 2D

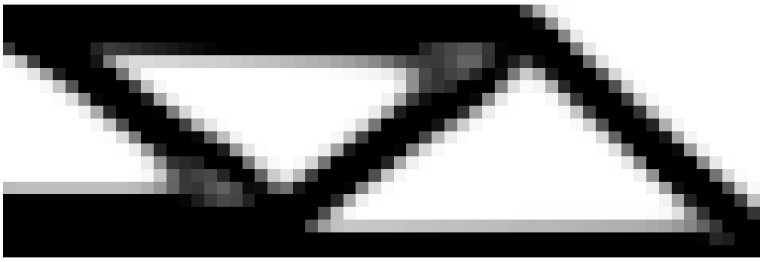
6.1.1 Minimum compliance: MBB beam

Figure 6.1 depicts the well known MBB beam problem, the same 'standard' problem as defined and solved by Sigmund's 99-line MATLAB code. The design domain is a 'rectangle' loaded vertically downward in the centre at the top with the bottom two corners constrained in the vertical direction. We only model half of the domain since we can conveniently exploit the symmetry of the problem. The ToPy problem definition file (TPD file) for this problem is given in Listing B.1 in Appendix B and the (condensed) output of ToPy and MATLAB are shown below. The results are shown in Figure 6.2.

Figure 6.1: 2D mbb beam problem.



(a) MATLAB result



(b) ToPy result

Figure 6.2: 2D mbb beam with reciprocal intervening variables; 60×20 $Q4$ elements.

Condensed output: ToPy

```
================================================================================
ToPy problem definition (TPD) file successfully parsed.
TPD file name: beam_2d_reci.tpd (v2007)
--------------------------------------------------------------------------------
Domain discretisation (NUM_ELEM_X x NUM_ELEM_Y) = 60 x 20
Element type (ELEM_K) = Q4
Filter radius (FILT_RAD) = 1.5
Number of iterations (NUM_ITER) = 100
Problem type (PROB_TYPE) = comp
Problem name (PROB_NAME) = beam_2d_reci
Continuation of penalisation factor (P_FAC) not specified
GSF not active
Damping factor (ETA) = 0.50
No passive elements (PASV_ELEM) specified
No active elements (ACTV_ELEM) specified
================================================================================
```

```
 Iter | Obj. func.   | Vol.  | Change     | P_FAC | Q_FAC | Ave ETA | S-V frac.
------------------------------------------------------------------------------
   1  | 1.007023e+03 | 0.500 | 2.0000e-01 | 3.000 | 1.000 |  0.500  |  0.000
   2  | 5.819346e+02 | 0.500 | 2.0000e-01 | 3.000 | 1.000 |  0.500  |  0.000
   3  | 4.145937e+02 | 0.500 | 2.0000e-01 | 3.000 | 1.000 |  0.500  |  0.115
<snip>
  98  | 2.032089e+02 | 0.500 | 7.9267e-03 | 3.000 | 1.000 |  0.500  |  0.636
  99  | 2.032056e+02 | 0.500 | 7.9487e-03 | 3.000 | 1.000 |  0.500  |  0.634
 100  | 2.032032e+02 | 0.500 | 7.8996e-03 | 3.000 | 1.000 |  0.500  |  0.634
```

Condensed output: Sigmund's `top.m` MATLAB code

```
>> top(60, 20, 0.5, 3, 1.5)
 It.:    1 Obj.:  1007.0226 Vol.:  0.500 ch.:  0.200
 It.:    2 Obj.:   581.9325 Vol.:  0.500 ch.:  0.200
 It.:    3 Obj.:   414.5974 Vol.:  0.500 ch.:  0.200
<snip>
 It.:   98 Obj.:   203.2091 Vol.:  0.500 ch.:  0.008
 It.:   99 Obj.:   203.2013 Vol.:  0.500 ch.:  0.008
 It.:  100 Obj.:   203.2061 Vol.:  0.500 ch.:  0.008
```

6.1.1.1 Discussion

This is the well known result of Sigmund, with material running from the point of load to the support, resulting in a truss-like structure, as one would expect and perhaps intuitively design. There is a significant amount of intermediate density material present, even after 100 iterations (which is equivalent to a 1% or less change in the last two objective function values for this problem).

Also, it is clear that the results are the same, except for some 'round-off' error which may be attributed to a myriad of possibilities, *e.g.*, the use of different algorithms for solving the FE problem, how numbers are stored and the operating system that is used.

6.1.2 Heat conduction: square plate

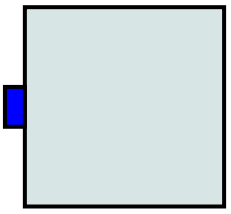


Figure 6.3: 2D heat conduction problem.

The square plate of Figure 6.3 is evenly heated and the temperature at the centre of the left edge is set to zero, *i.e.*, it acts as a heat sink. The problem is to find an optimal distribution of two materials with different isotropic conductivities inside the domain. Again, the TPD file is given in Listing B.2.

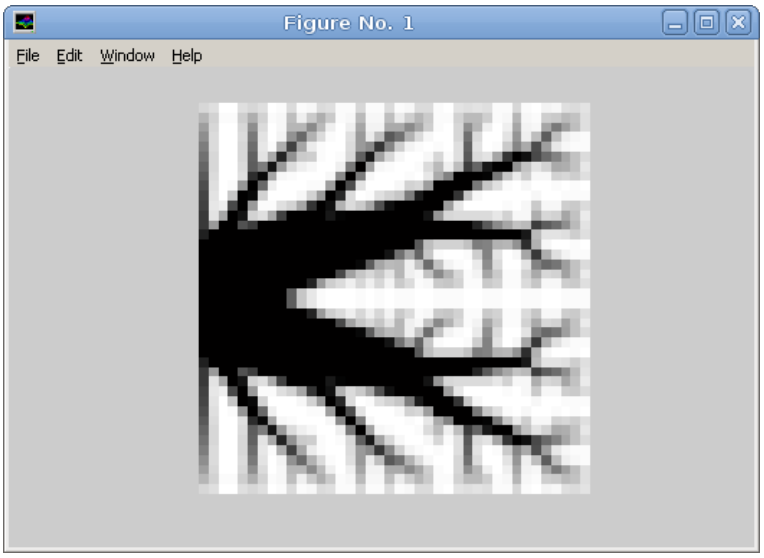
The (condensed) output of both are shown below and the results are shown in Figure 6.4.

Condensed output: ToPy

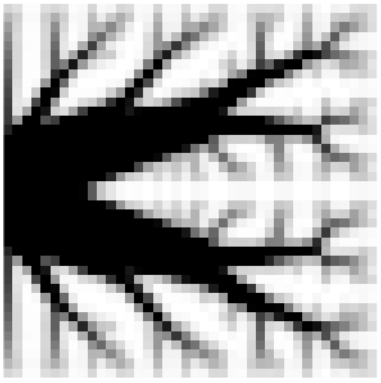
```
================================================================================
ToPy problem definition (TPD) file successfully parsed.
TPD file name: heat_2d_reci.tpd (v2007)
--------------------------------------------------------------------------------
Domain discretisation (NUM_ELEM_X x NUM_ELEM_Y) = 40 x 40
Element type (ELEM_K) = Q4T
Filter radius (FILT_RAD) = 1.2
Number of iterations (NUM_ITER) = 100
Problem type (PROB_TYPE) = heat
Problem name (PROB_NAME) = heat_2d_reci
Continuation of penalisation factor (P_FAC) not specified
GSF not active
Damping factor (ETA) = 0.50
No passive elements (PASV_ELEM) specified
No active elements (ACTV_ELEM) specified
================================================================================
 Iter | Obj. func.   | Vol.  | Change     | P_FAC | Q_FAC | Ave ETA | S-V frac.
--------------------------------------------------------------------------------
   1  | 3.875298e+03 | 0.400 | 2.0000e-01 | 3.000 | 1.000 |  0.500  |  0.000
   2  | 1.569962e+03 | 0.400 | 2.0000e-01 | 3.000 | 1.000 |  0.500  |  0.000
   3  | 1.025087e+03 | 0.400 | 2.0000e-01 | 3.000 | 1.000 |  0.500  |  0.039
<snip>
  98  | 4.477724e+02 | 0.400 | 7.8919e-03 | 3.000 | 1.000 |  0.500  |  0.253
  99  | 4.477672e+02 | 0.400 | 7.1980e-03 | 3.000 | 1.000 |  0.500  |  0.253
 100  | 4.477628e+02 | 0.400 | 6.5722e-03 | 3.000 | 1.000 |  0.500  |  0.254
```

Condensed output: Sigmund's `toph.m` MATLAB code

```
>> toph(40, 40, 0.4, 3, 1.2)
 It.:    1 Obj.:  3875.297769 Vol.: 4.000020e-001 ch.:  0.200000
 It.:    2 Obj.:  1569.950312 Vol.: 4.000012e-001 ch.:  0.200000
 It.:    3 Obj.:  1025.082190 Vol.: 4.000015e-001 ch.:  0.200000
```



(a) MATLAB result



(b) ToPy result

Figure 6.4: 2D square plate with reciprocal intervening variables; 40×40 $Q4T$ elements.

```
<snip>
 It.:   98 Obj.:   447.779095 Vol.: 3.999932e-001 ch.:  0.007901
 It.:   99 Obj.:   447.777280 Vol.: 4.000032e-001 ch.:  0.007177
 It.:  100 Obj.:   447.756415 Vol.: 3.999954e-001 ch.:  0.006574
```

6.1.2.1 Discussion

Comparing (visually) the output and Figure 6.4, we see that the results are the same.

6.1.3 Mechanism synthesis: inverter

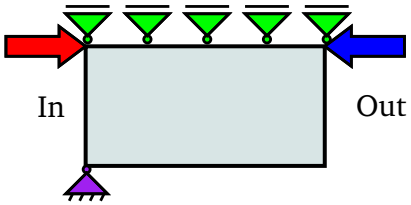


Figure 6.5: 2D inverter problem.

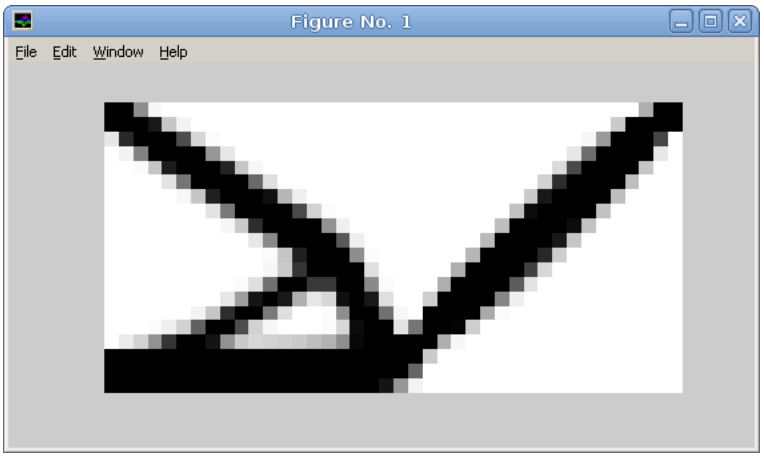
The mechanism of Figure 6.5 produces an inverse displacement. The red arrow (left) indicates input displacement while the blue one indicates the required output displacement. The idea is that as the red arrow 'displaces' material towards the interior of the domain, the material at the blue arrow must also move inwards. Symmetry is once again utilised by suitably constraining the top edge of the domain. As before, the TPD file is given in Listing B.3. The (condensed) output of both are shown below and the results are shown in Figure 6.6:

Condensed output: ToPy

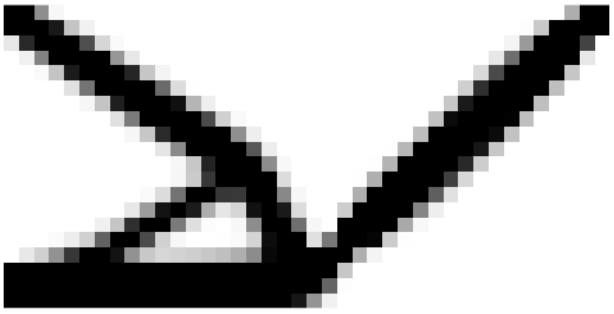
```
================================================================================
ToPy problem definition (TPD) file successfully parsed.
TPD file name: inverter_2d_eta.tpd (v2007)
--------------------------------------------------------------------------------
Domain discretisation (NUM_ELEM_X x NUM_ELEM_Y) = 40 x 20
Element type (ELEM_K) = Q4
Filter radius (FILT_RAD) = 1.2
Number of iterations (NUM_ITER) = 100
Problem type (PROB_TYPE) = mech
Problem name (PROB_NAME) = inverter_2d_eta03
Continuation of penalisation factor (P_FAC) not specified
GSF not active
Damping factor (ETA) = 0.30
No passive elements (PASV_ELEM) specified
No active elements (ACTV_ELEM) specified
================================================================================
 Iter | Obj. func.    | Vol.  | Change      | P_FAC | Q_FAC | Ave ETA | S-V frac.
--------------------------------------------------------------------------------
   1  | 1.174775e-01 | 0.283 | 1.0000e-01 | 3.000 | 1.000 |  0.300  |  0.000
   2  | 1.015754e-02 | 0.295 | 1.0000e-01 | 3.000 | 1.000 |  0.300  |  0.000
   3  | 6.257350e-05 | 0.294 | 1.0000e-01 | 3.000 | 1.000 |  0.300  |  0.225
<snip>
  98  | -1.114243e+00 | 0.300 | 6.6710e-03 | 3.000 | 1.000 |  0.300  |  0.756
  99  | -1.114256e+00 | 0.300 | 6.6098e-03 | 3.000 | 1.000 |  0.300  |  0.756
 100  | -1.114267e+00 | 0.300 | 6.5594e-03 | 3.000 | 1.000 |  0.300  |  0.756
```

Condensed output: Sigmund's `topm.m` MATLAB code

```
>> topm(40, 20, 0.3, 3, 1.2)
 It.:    1 Obj.:     0.117477 Vol.: -1.725000e-002 ch.:  0.100000
 It.:    2 Obj.:     0.010158 Vol.: -5.000000e-003 ch.:  0.100000
```



(a) MATLAB result



(b) ToPy result

Figure 6.6: 2D inverter with ETA=0.3 intervening variables; 40×20 $Q4$ elements.

```
 It.:    3 Obj.:     0.000063 Vol.: -5.525000e-003 ch.:  0.100000
<snip>
 It.:   98 Obj.:    -1.114248 Vol.: 1.443265e-006 ch.:  0.006675
 It.:   99 Obj.:    -1.114260 Vol.: 1.118107e-007 ch.:  0.006616
 It.:  100 Obj.:    -1.114267 Vol.: -1.561612e-006 ch.:  0.006567
```

6.1.3.1 Discussion

Comparing the output and (Figure 6.6), we can again see that the results are the same.

6.2 Minimum compliance problems in 2D with GSF

We present results for 2D problems illustrating the effect of GSF. To *depict* intermediate density material in 3D is a challenging problem in itself, so we have opted to show the GSF results in 2D here, merely for convenience.

6.2.1 MBB beam with GSF

We again solve the MBB beam problem, but in this case we use GSF. The ToPy problem definition file for this problem is listed in Listing B.4 in Appendix B and the result is shown in Figure 6.7. As shown in the listing, `ETA` is fixed at 0.5 (giving reciprocal intervening variables). The penalty factor and the extra penalisation factor (q in Chapter 3, `Q_FAC` in the TPD file) are both ramped after 20 iterations, with p (`P_FAC`) being ramped faster. Each has a maximum value they can attain.

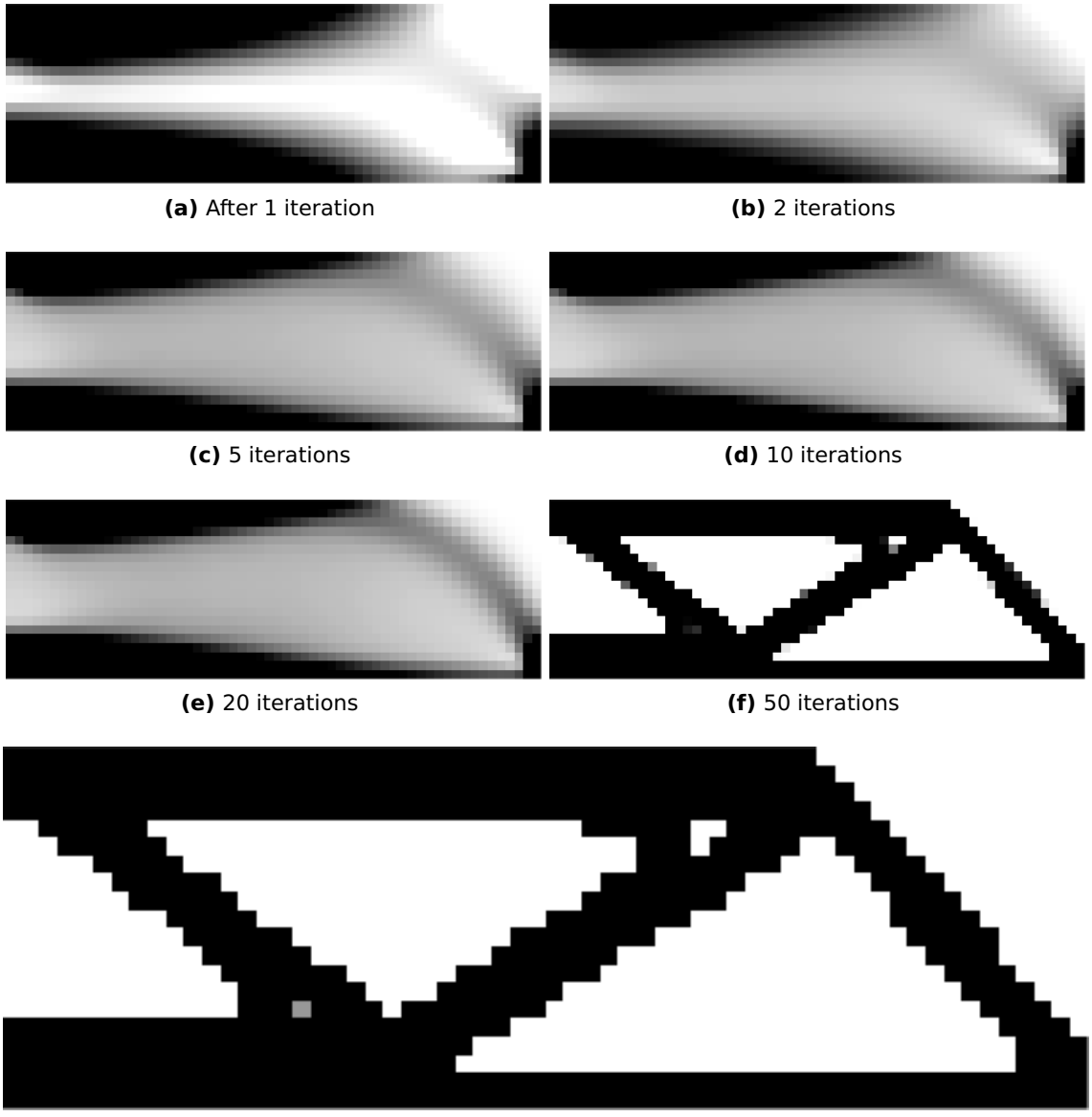
Condensed output

```
================================================================================
ToPy problem definition (TPD) file successfully parsed.
TPD file name: beam_2d_reci_gsf.tpd (v2007)
--------------------------------------------------------------------------------
Domain discretisation (NUM_ELEM_X x NUM_ELEM_Y) = 60 x 20
Element type (ELEM_K) = Q4
Filter radius (FILT_RAD) = 1.5
Number of iterations (NUM_ITER) = 100
Problem type (PROB_TYPE) = comp
Problem name (PROB_NAME) = beam_2d_reci_gsf
GSF active
Damping factor (ETA) = 0.50
No passive elements (PASV_ELEM) specified
No active elements (ACTV_ELEM) specified
================================================================================
 Iter | Obj. func.   | Vol.  | Change     | P_FAC | Q_FAC | Ave ETA | S-V frac.
--------------------------------------------------------------------------------
   1  | 2.517557e+02 | 0.500 | 2.0000e-01 | 1.000 | 1.000 |  0.500  |  0.000
   2  | 1.981605e+02 | 0.500 | 2.0000e-01 | 1.000 | 1.000 |  0.500  |  0.000
   3  | 1.745135e+02 | 0.500 | 1.7882e-01 | 1.000 | 1.000 |  0.500  |  0.182
<snip>
  98  | 1.908648e+02 | 0.500 | 9.0142e-09 | 3.000 | 4.650 |  0.500  |  0.999
  99  | 1.908648e+02 | 0.500 | 5.9517e-10 | 3.000 | 4.700 |  0.500  |  0.999
 100  | 1.908648e+02 | 0.500 | 1.3074e-09 | 3.000 | 4.750 |  0.500  |  0.999

Solid plus void to total elements fraction = 0.99917
100 iterations took 0.730 minutes (0.007 min/iter. or 0.438 sec/iter.)
Average of all ETA's = 0.500 (average of all a's = 1.000)
```

6.2.1.1 Discussion

The result (Figure 6.7) is similar to the result of the standard problem solved without GSF, but resulting in a predominantly solid-void structure. In fact, if one looks closely there is only a single grey element visible. Note the high solid-void fraction (`S-V frac.`) of 0.99917 for the last iteration, as well as a lower objective function value.



(a) After 1 iteration

(b) 2 iterations

(c) 5 iterations

(d) 10 iterations

(e) 20 iterations

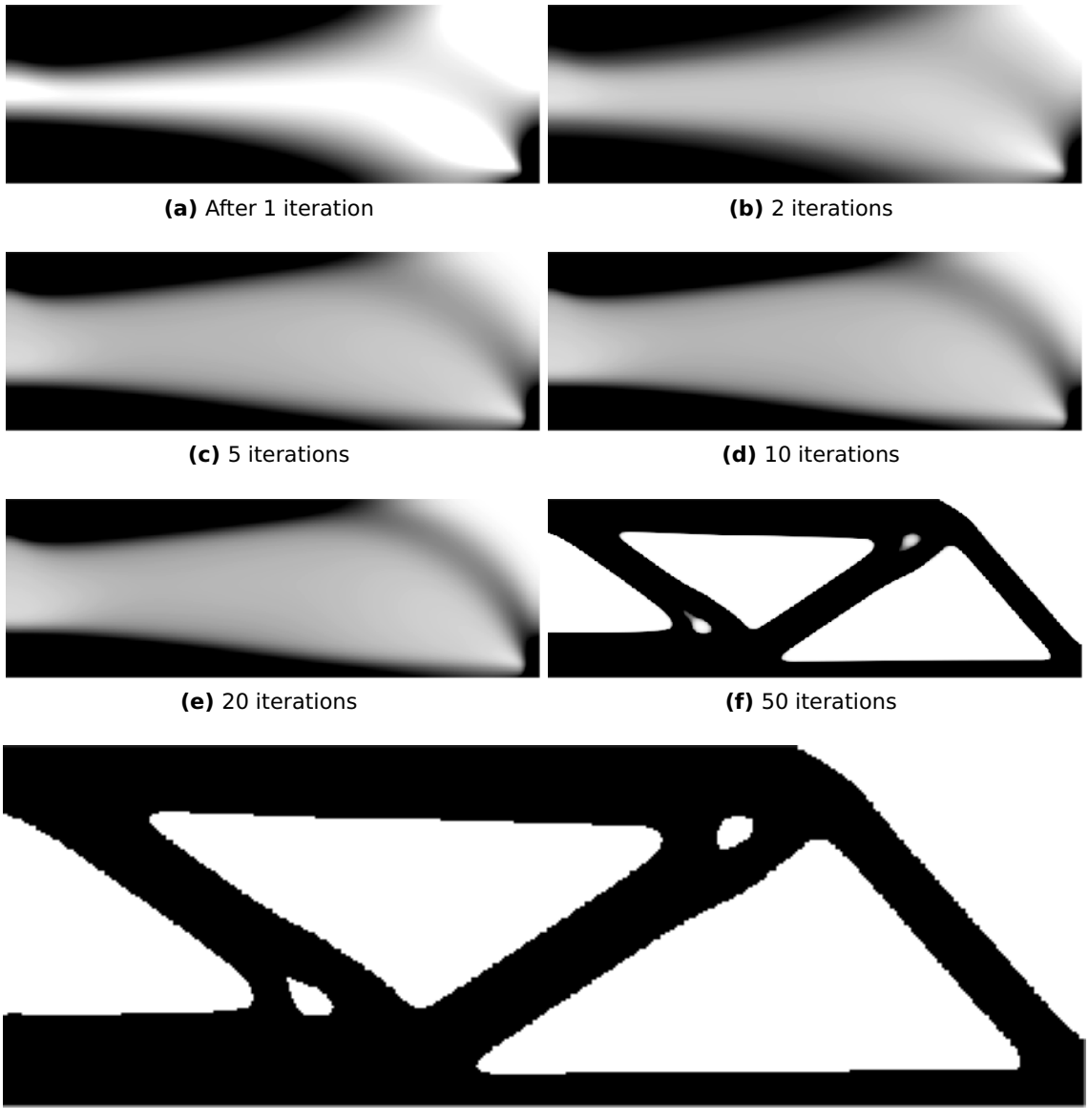
(f) 50 iterations

(g) 100 iterations

Figure 6.7: 2D mbb beam with GSF; 60×20 $Q4$ elements.

6.2.2 MBB beam with GSF; refined mesh

We present the MBB beam problem subjected to GSF and a higher discretisation; increased by a factor of 10 per dimension, thus 600×200 as opposed to 60×20. The result is shown in Figure 6.8. We have to increase the filter radius by the same amount as the increase in number of elements per dimension, namely $10 \times 1.5 = 15$, in order to avoid results that are dependant on the finer mesh discretisation (refinement) and will result in the formation of fine-scale internal structure layout, see Bendsøe and Sigmund (2004, Section 1.3.1).



(a) After 1 iteration

(b) 2 iterations

(c) 5 iterations

(d) 10 iterations

(e) 20 iterations

(f) 50 iterations

(g) 100 iterations

Figure 6.8: 2D mbb beam with GSF; 600×200 $Q4$ elements.

6.2.2.1 Discussion

The result is similar to that of Section 6.2.1, again resulting in a predominantly solid-void structure but of finer 'resolution'. Note the lower small void in this result due to the higher discretisation. The void is not so prominant in the 'coarse' 60×20 since the relatively larger finite elements do a poor job of representing the geometry (shape of the void) in this area. It is the same as comparing two pictures of the same object taken with a high resolution and a low resolution camera.

6.3 Minimum compliance problems in 3D with GSF

6.3.1 Comparison of different methods

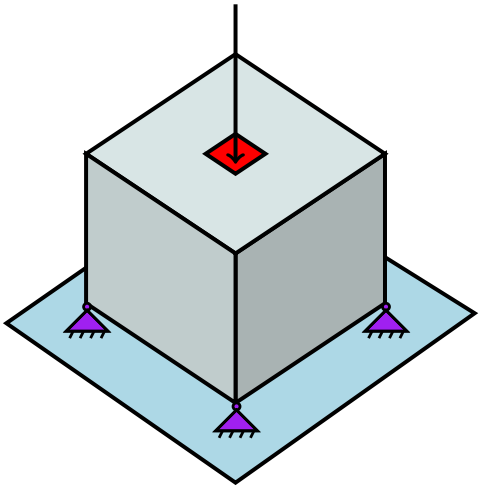


Figure 6.9: 3D trestle problem

The design domain is an equal sided 'cube' loaded in the centre at the top with the bottom four corners fully constrained. For this problem, we use seven values for η, namely $[0.1, 0.3, 0.4, 0.5\ (=\text{reciprocal}), 0.6, 0.7, 0.9]$ and plot the objective function values and the solid-void fraction against their respective values, as shown in Figures 6.10 and 6.11.

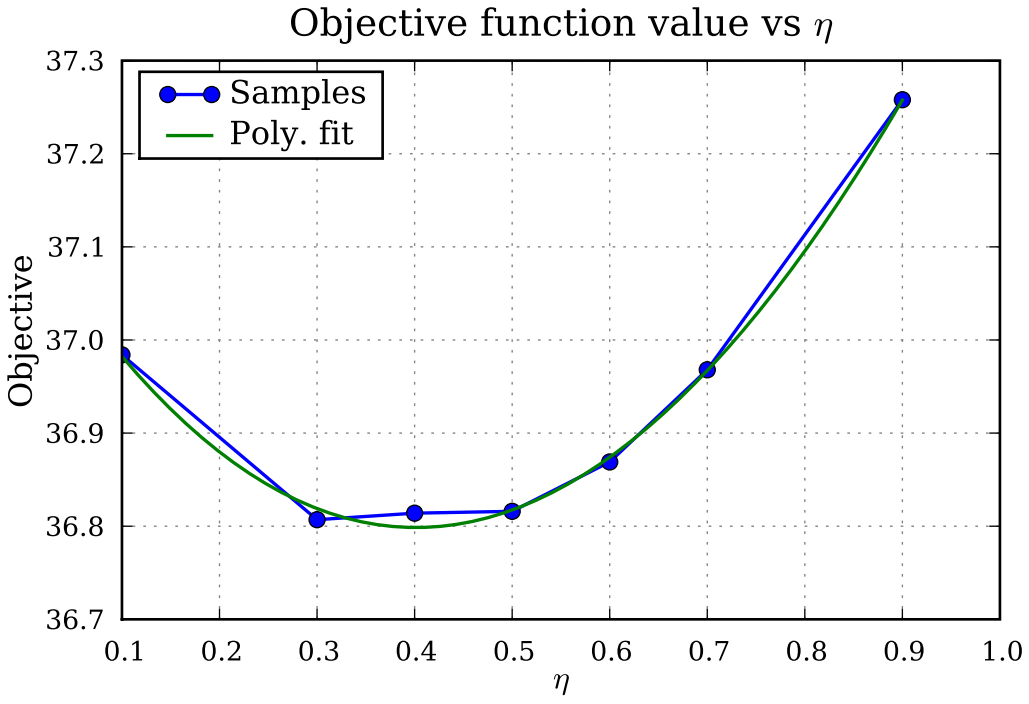


Figure 6.10: Second degree polynomial fit to objective vs η data.

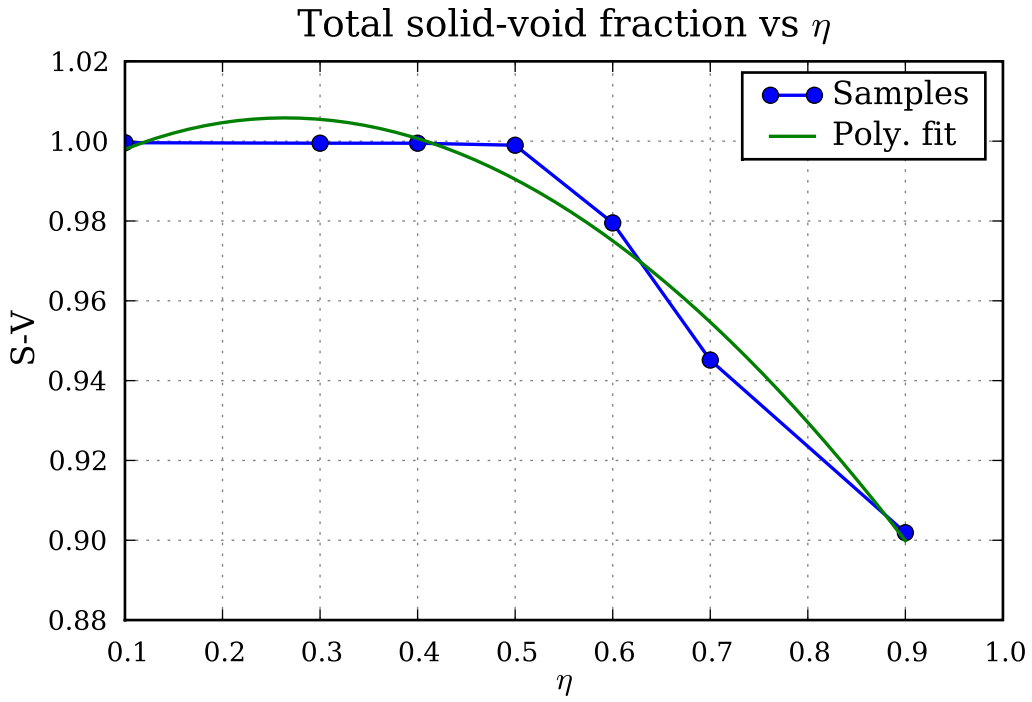


Figure 6.11: Second degree polynomial fit to solid-void fraction vs η data.

The use of $\eta = 0.5$ for structures, which results in a reciprocal approximation, is well documented (see *e.g.*, Haftka and Gürdal (1992, Section 6.1)). While our approach here is most certainly not exhaustive, we notice that $\eta = 0.4$ also gives good (low) objective function values. Also, for the limited amount of iterations, we also get good (≈ 1) values for the solid-void fraction, which will eventually also reach ≈ 1 after enough iterations.

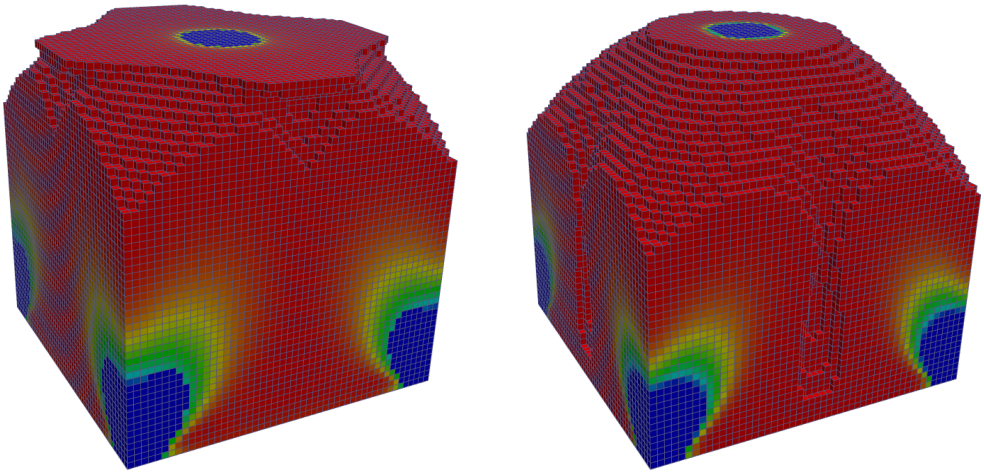
Since Bendsøe and Sigmund (2004) recommends $\eta = 0.5$ for compliance minimisation and heat conduction problems, and $\eta = 0.3$ for mechanism synthesis, we will hereinafter use $\eta = 0.4$ for all 3D problems, since it is a value that 'sits' nicely in the middle, and because of our little 'experiment'.

Performing the same experiment with $\eta =$ exponential gives disappointing results: η (on average) is driven past 0.9, and this gives poor results in light of our findings above. For this specific problem then, the exponential approximation is not recommended.

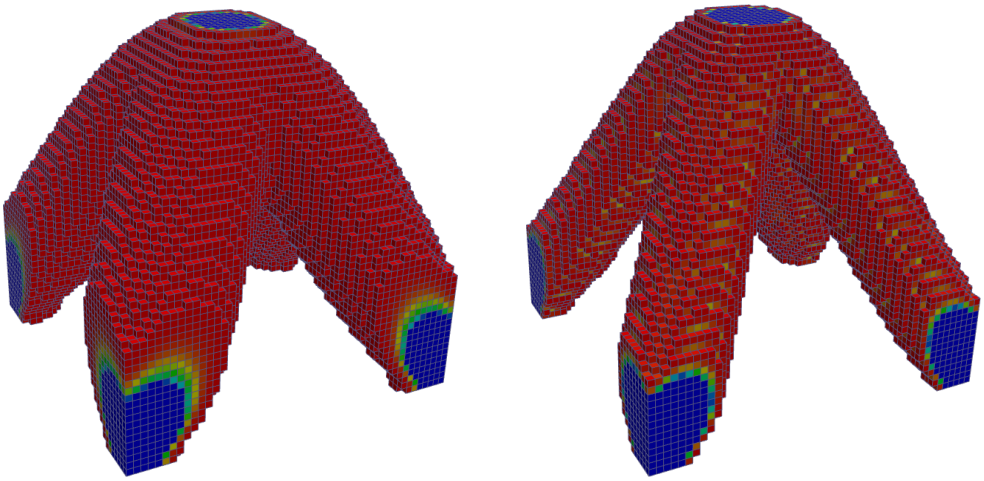
Figure 6.12 shows the final result for $\eta = 0.4$.

6.3.1.1 Discussion

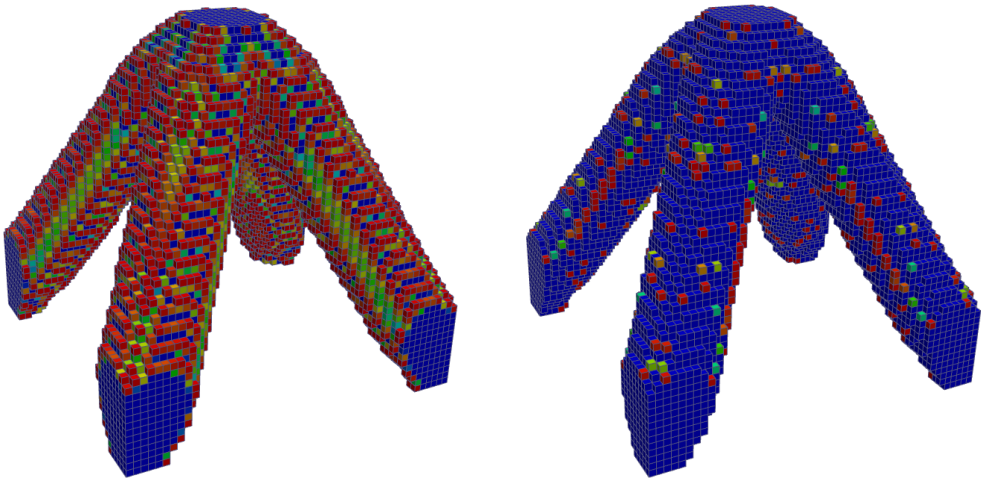
One expects a simple four-legged trestle (*Afr: bokkie*) to form, without any braces between the legs since the four corners are not allowed to move at all.



(a) 5 iterations **(b)** 15 iterations

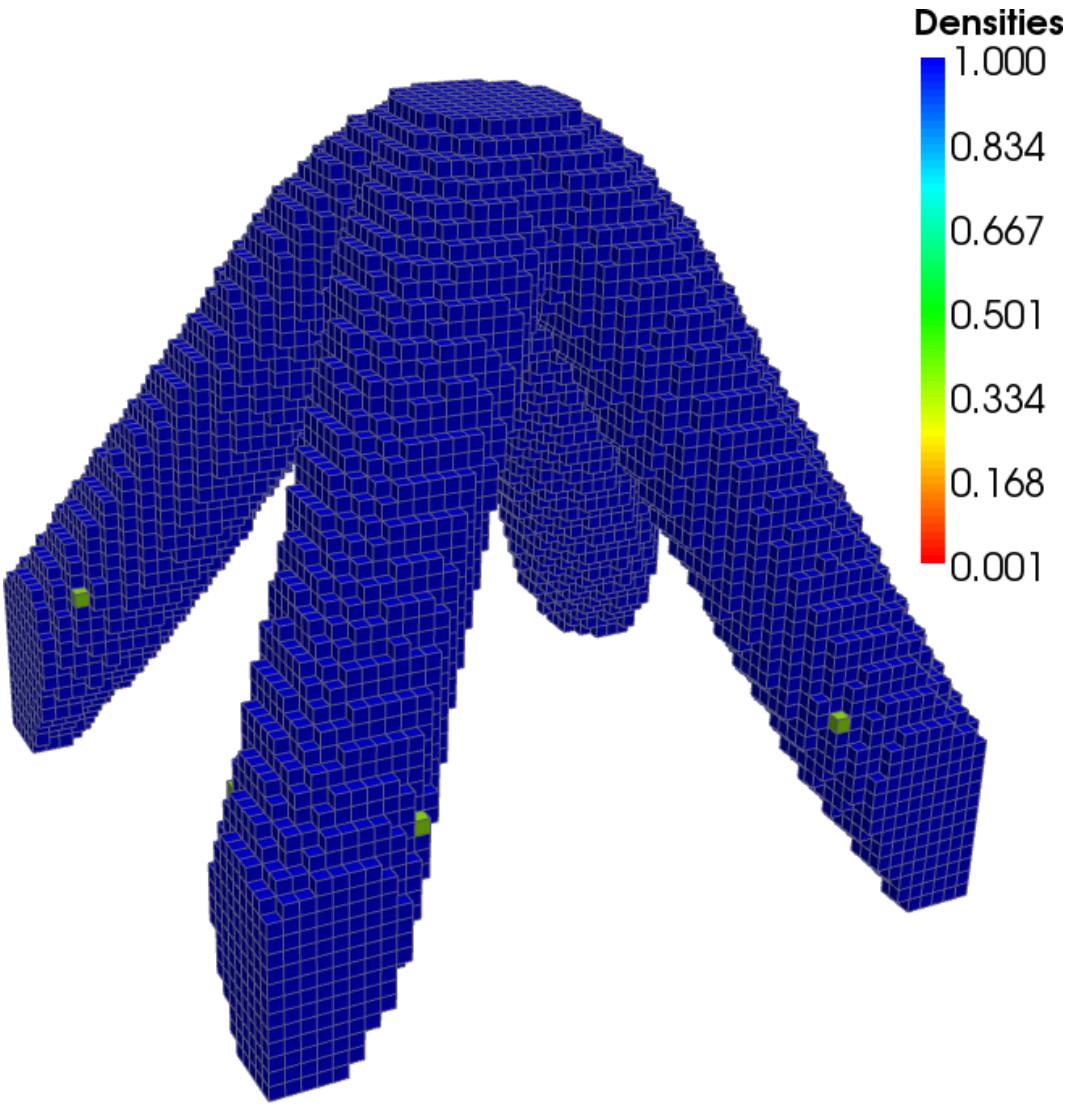


(c) 20 iterations **(d)** 25 iterations



(e) 30 iterations **(f)** 40 iterations

Figure 6.12: 3D trestle with GSF; $51 \times 51 \times 51$ $H8$ elements (page 1/2).



(g) 50 iterations

(h) 50 iterations (auxiliary view)

Figure 6.12: 3D trestle with GSF; $51 \times 51 \times 51$ $H8$ elements (page 2/2).

6.3.2 3D Cantilever

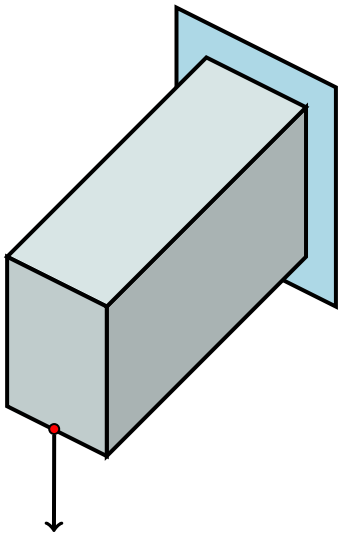


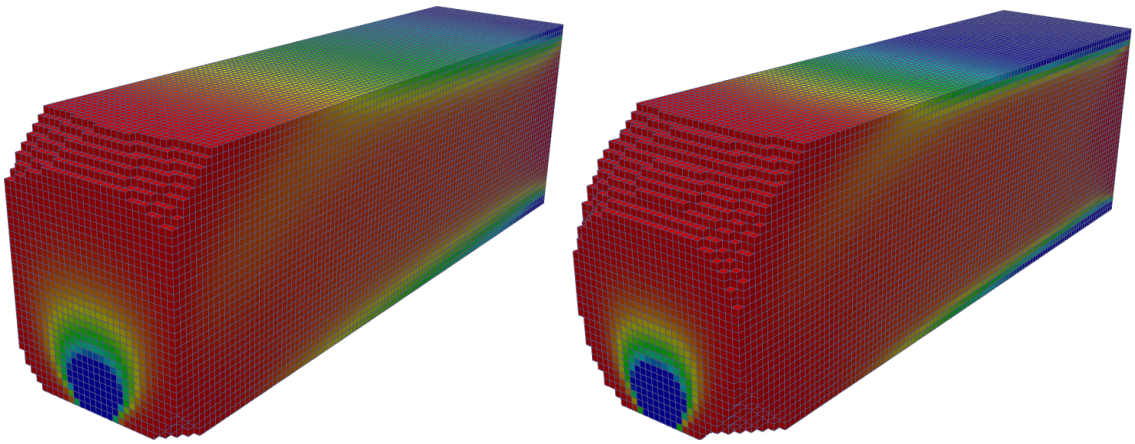
Figure 6.13: 3D cantilever problem.

The design domain is a rectangular prism fully constrained at one end (the wall, furthest from the reader). The domain is loaded vertically downwards in the centre on the lower side of the domain at the free end (closest to the reader). The load is applied to a single node. The ToPy problem definition file for this problem is given in Listing B.5 and the result is shown in Figure 6.14.

6.3.2.1 Discussion

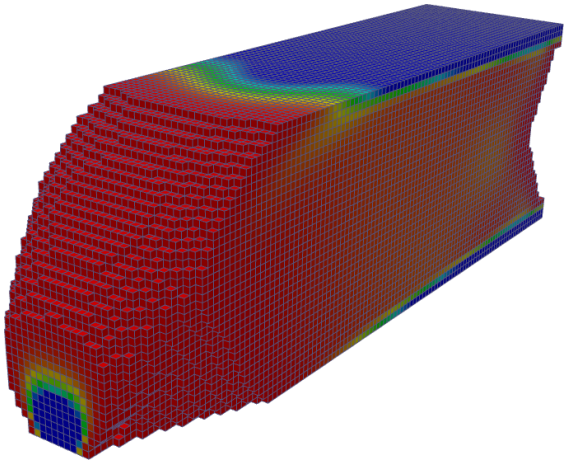
In the immediate vicinity of the applied load, braces are formed and further away from the load, where the local influence of the load is less significant, a castelated I-beam-like section results. The geometry is consistent with a maximum second moment of area, where material is distributed away from the neutral axis.

Also note that the resulting structure consists almost entirely of solid elements (densities equal to 1) as a result of GSF.

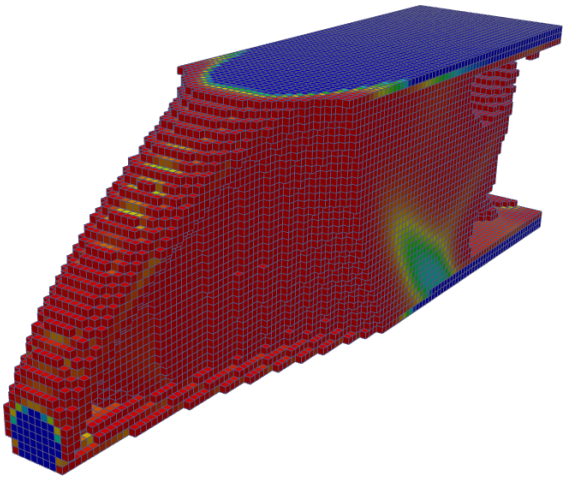


(a) 5 iterations

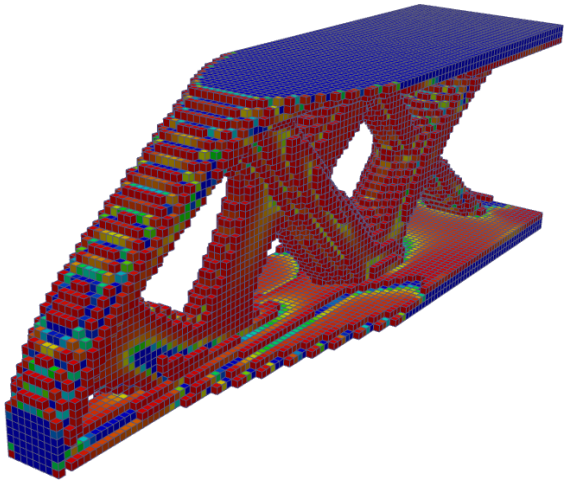
(b) 15 iterations



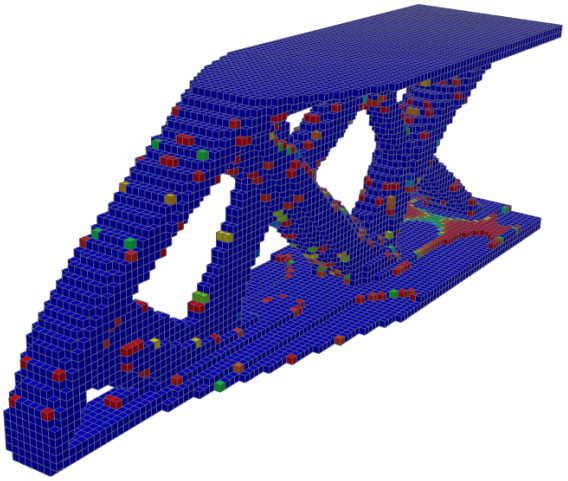
(c) 20 iterations



(d) 25 iterations

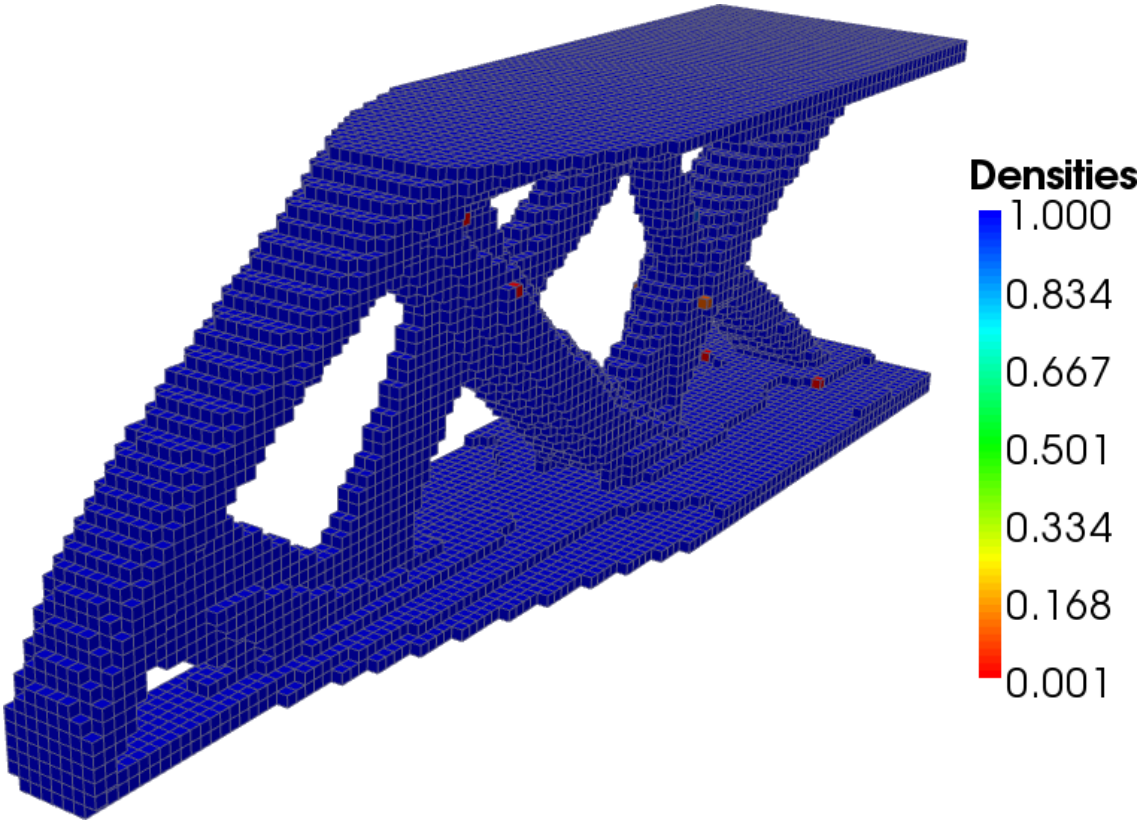


(e) 30 iterations

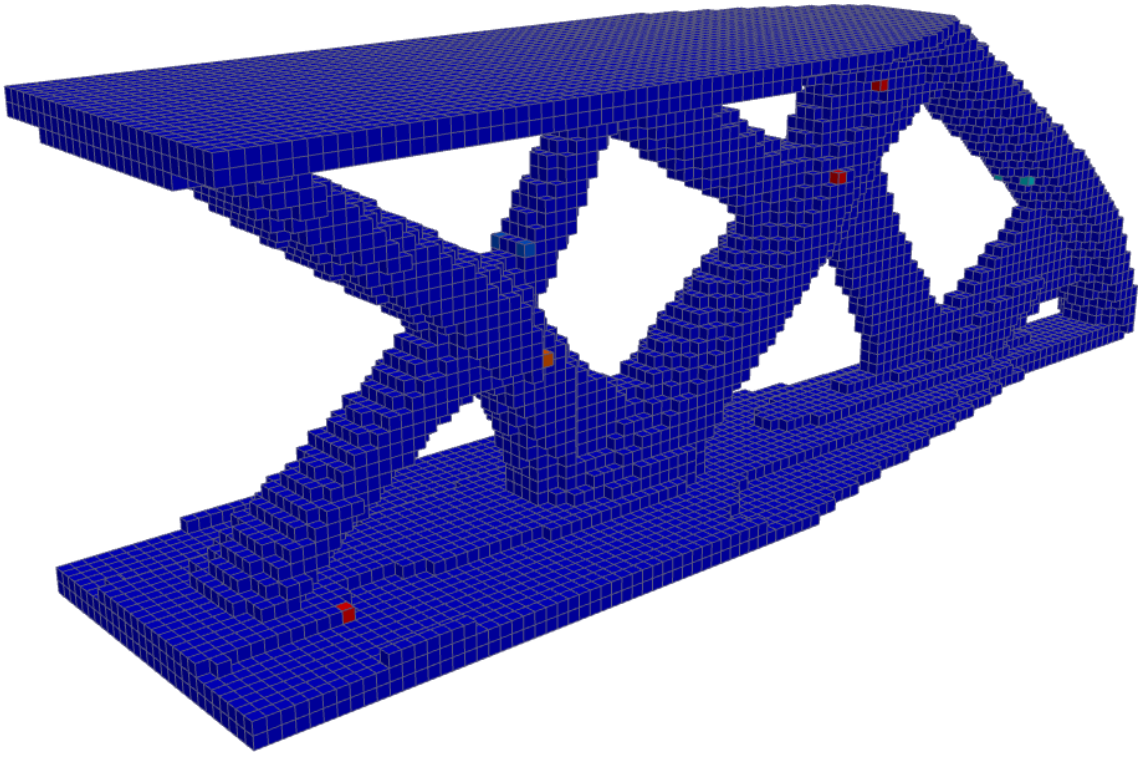


(f) 40 iterations

Figure 6.14: 3D cantilever with GSF; $28 \times 37 \times 111$ $H8$ elements (page 1/2).



(g) 50 iterations



(h) 50 iterations (auxiliary view)

Figure 6.14: 3D cantilever with GSF; $28 \times 37 \times 111$ $H8$ elements (page 2/2).

6.3.3 3D Dogleg

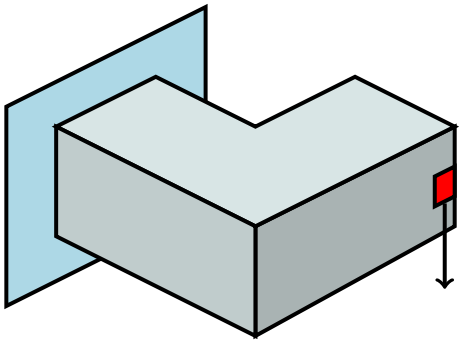


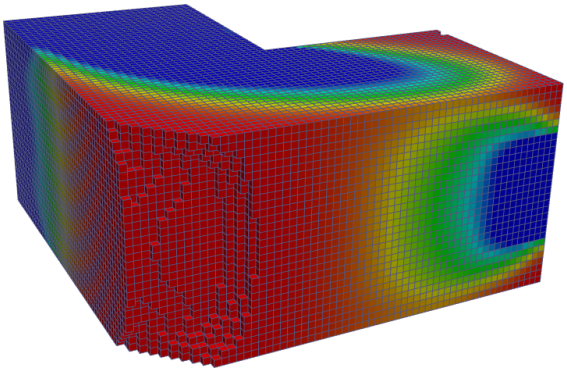
Figure 6.15: 3D dogleg problem.

The design domain is an L-shaped ('dogleg') prism fully constrained at one end (the wall, furthest from the reader). The domain is loaded vertically downwards in the centre on the near side of the domain at the right. Nine nodes forming a 3×3 square is loaded, each node with the same value. The ToPy problem definition file for this problem is given in Listing B.6 and the result is shown in Figure 6.16.

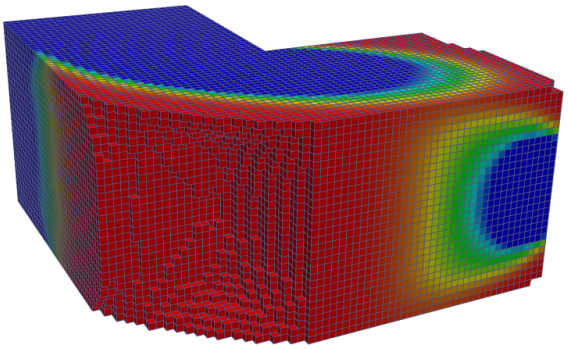
6.3.3.1 Discussion

The geometry protruding perpendicularly from the wall is subjected to torsion, in this volume a tube-like geometry can be observed which is one of the most effective ways of resisting deflection due to torque. As one moves towards the load, the geometry transforms into an I-beam-like structure and eventually tapers down towards a C-beam-like cross-section.

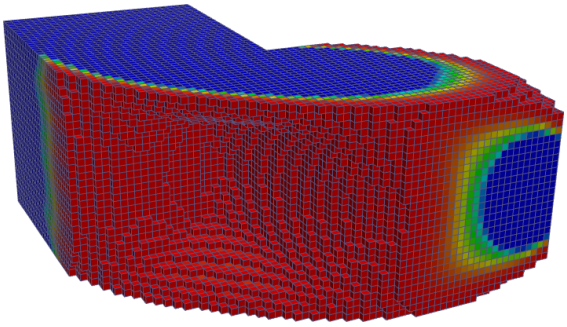
Again, note that the resulting structure consists almost entirely of solid elements.



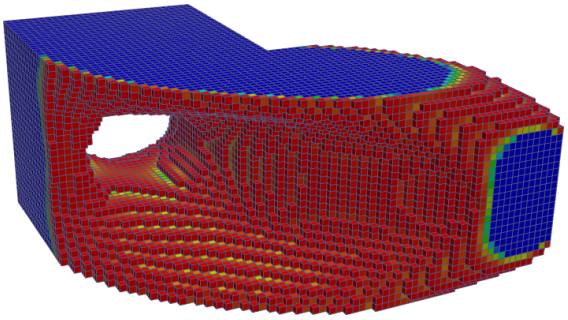
(a) 5 iterations



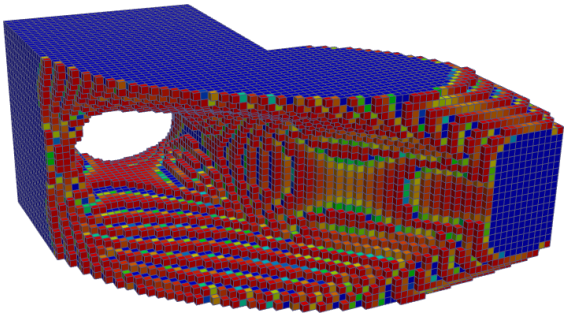
(b) 15 iterations



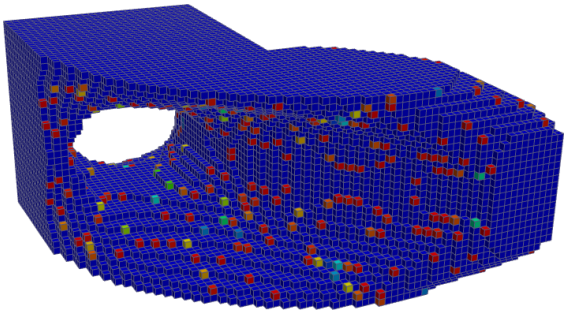
(c) 20 iterations



(d) 25 iterations

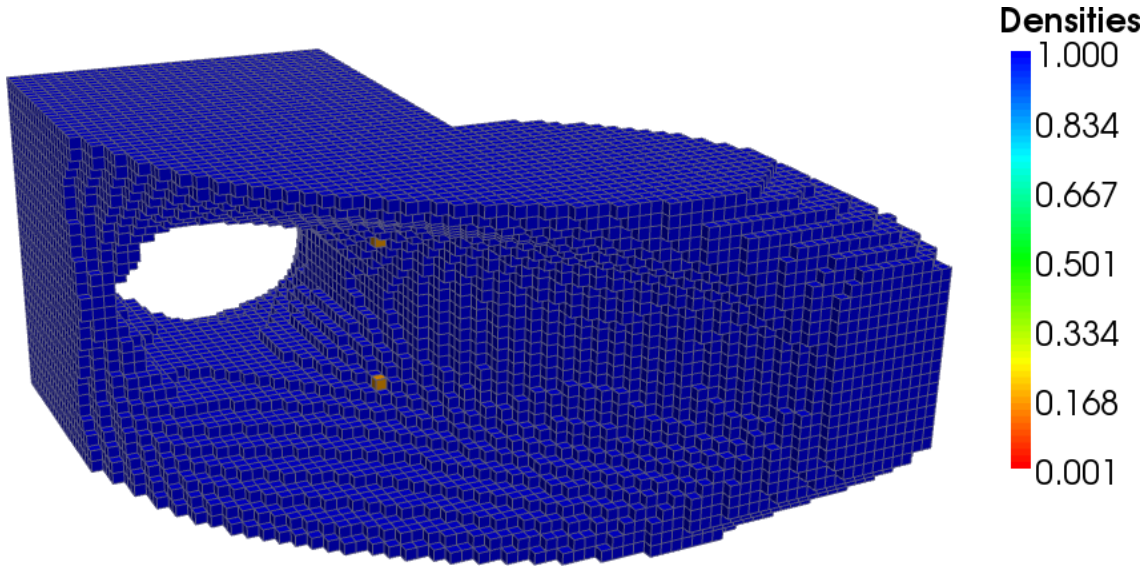


(e) 30 iterations

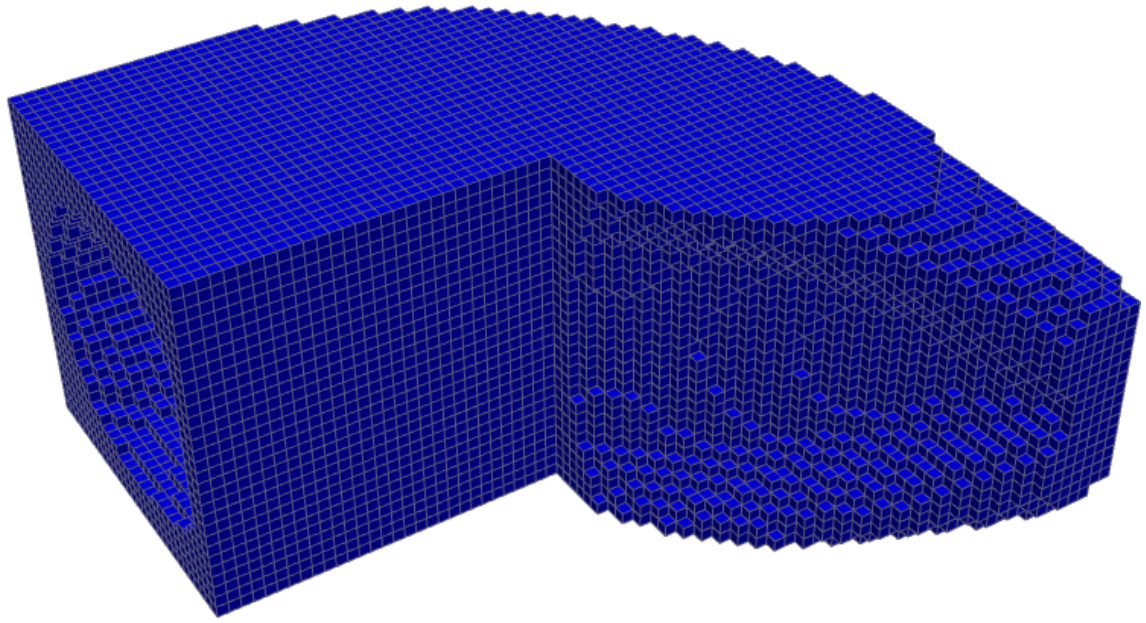


(f) 40 iterations

Figure 6.16: 3D cantilever with GSF; $60 \times 30 \times 60$ $H8$ elements (page 1/2).



(g) 50 iterations



(h) 50 iterations (auxiliary view)

Figure 6.16: 3D dogleg with GSF; $60 \times 30 \times 60$ $H8$ elements (page 2/2).

6.4 Bending dominated problems

In this section we briefly investigate the effect of the (hybrid) bending exact elements of Chapter 4 on compliance minimisation compared to 'standard' bi-linear elements. The mentioned hybrid elements are accuarte with regards to displacement (compared to the 'standard' elements), and since compliance is a function of displacement, it warrants an investigation to determine whether these elements will give subjectively better results.

For the 2D problem, we use the penalised equilibrium element ($Q4\alpha5\beta$), and for 3D the hybrid stress element (18β-NC).

6.4.1 Minimum compliance: 2D T-piece

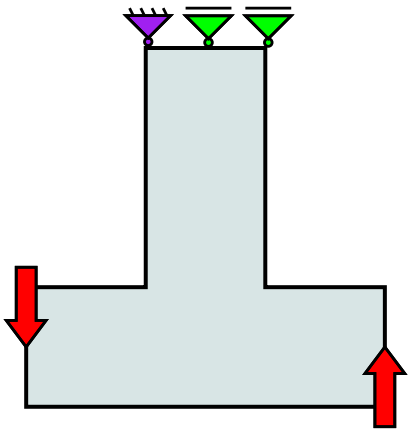


Figure 6.17: 2D t-piece problem.

The design domain is a so-called 'T-piece', obtained by excluding two regions on either side of the middle. The ends of the 'T' are loaded in opposite directions as shown, in order to create a bending dominated problem. In fact, the vertical portion of the domain will experience pure bending. Figures 6.18 and 6.19 shows final results for the $Q4$ and $Q4\alpha5\beta$ element.

6.4.1.1 Discussion

Even though the $Q4\alpha5\beta$ element is bending exact, as opposed to the $Q4$ element which performs poorly in bending dominated problems, the results are indistinguishable to the naked eye. It is highly likely that in this particular example, the mesh discretisation for both problems are fine enough, yielding almost identical results in terms of displacement. In the limit of mesh refinement, both elements will give the same result...

The higher objective function values obtained using the $Q4\alpha5\beta$ element (refer to Table 6.1) can be attributed to the lower stiffness (in bending) of this element – a stiffer element will yield a stiffer structure, and since compliance is the inverse of stiffness, we get lower objective function values. It is however probably more relevant to look at the total change of initial versus final objective function values; we see that the total percentage change is

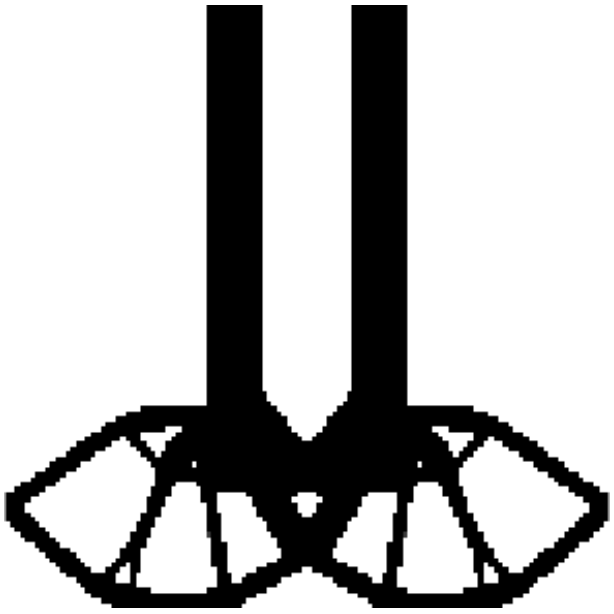


Figure 6.18: 2D T-piece with GSF; 120×120 $Q4$ elements.

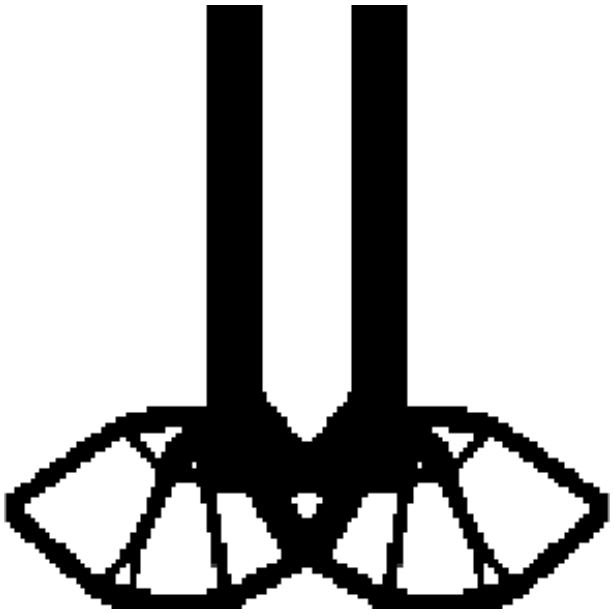


Figure 6.19: 2D T-piece with GSF; 120×120 $Q4\alpha5\beta$ elements.

virtually the same, with the 'stiffer' $Q4$ element performing marginally different than the $Q4\alpha5\beta$ element, for the same reason as above.

Note, once again, how material is distributed away from the symmetry line of the 'T', thereby increasing the second moment of area and minimising compliance (maximising stiffness). The geometry in the lower part of the 'T' is reminiscent of Michell (1904) structures.

Element type	Initial obj. func. val.	Final obj. func. val.	Percentage change	Solid-void fraction
$Q4$	190.3690	343.8942	81	0.99986
$Q4\alpha5\beta$	192.0133	344.8412	80	0.99986

Table 6.1: Comparison of 2D element performance in bending dominated problem.

6.4.2 Minimum compliance: 3D moment arm

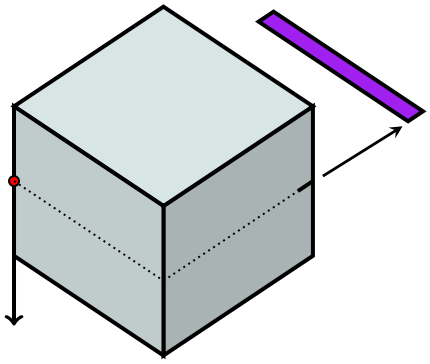


Figure 6.20: 3D moment arm problem.

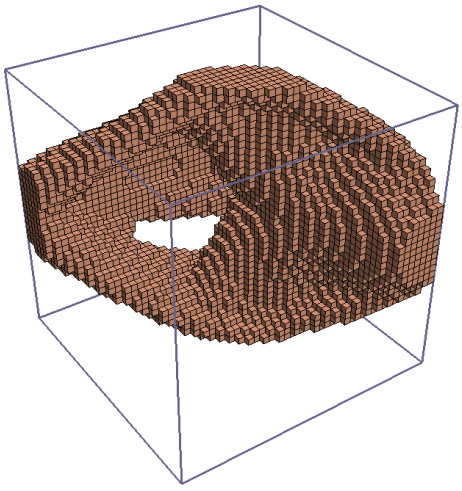
We present the minimum compliance solution to a 3D "moment arm" problem, using the 18β-NC element. The point load at a single node in the middle of the edge will create 'skew' bending of the domain. The domain is constrained by fixing three adjacent rows of nodes lying in the centre plane of the domain, thereby resisting torque. In addition, the outer row is also restrained to avoid rigid body motion, as well as three nodes on the edge in the same plane. The final result is shown in Figure 6.21, with various auxiliary views to aid visualisation. For brevity's sake, we do not show the result for $H8$ element as it is, like the 2D problem above, indistinguishable to the naked eye, compared to the "18β-NC result".

6.4.2.1 Discussion

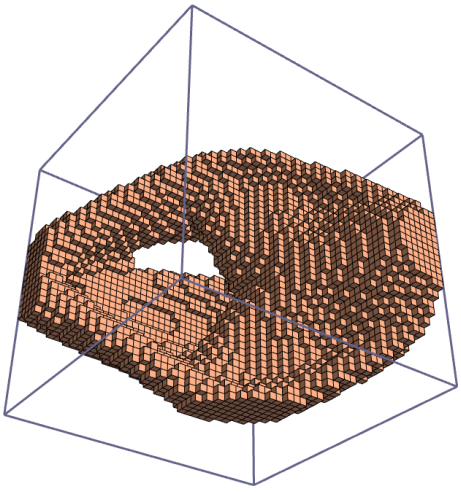
Here too, from Table 6.2, we see that the 'stiffer' $H8$ element performs only slightly different than its hybrid counterpart, and we cite the same reasons as for the 2D problem above.

Element type	Initial obj. func. val.	Final obj. func. val.	Percentage change	Solid-void fraction
$H8$	118.3226	262.6688	122	0.99994
18β-NC	129.8002	285.9374	120	0.99987

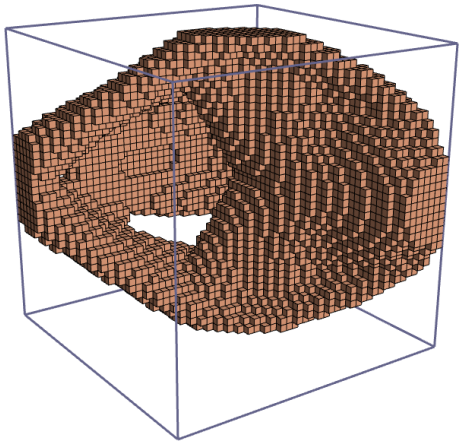
Table 6.2: Comparison of 3D element performance in bending dominated problem.



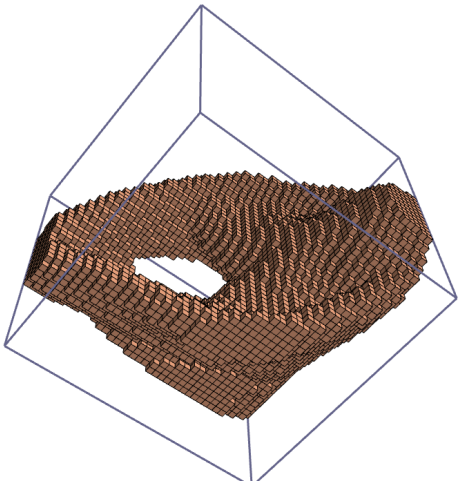
(a) Auxiliary view 1



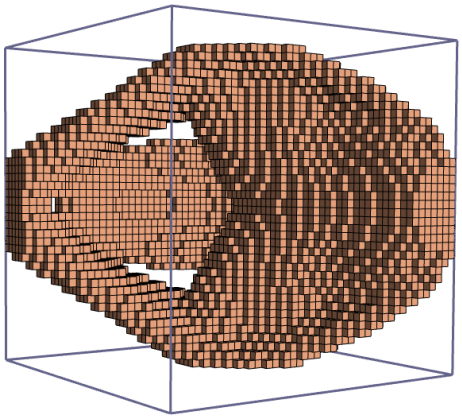
(b) Auxiliary view 4



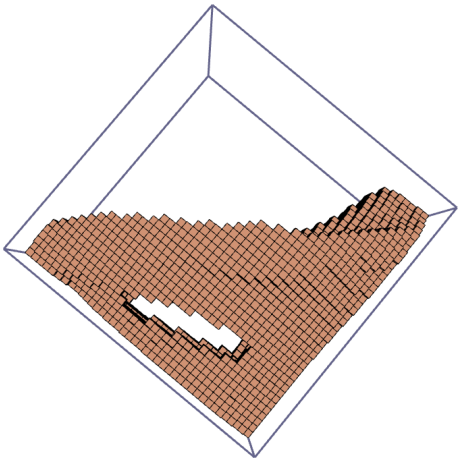
(c) Auxiliary view 2



(d) Auxiliary view 5

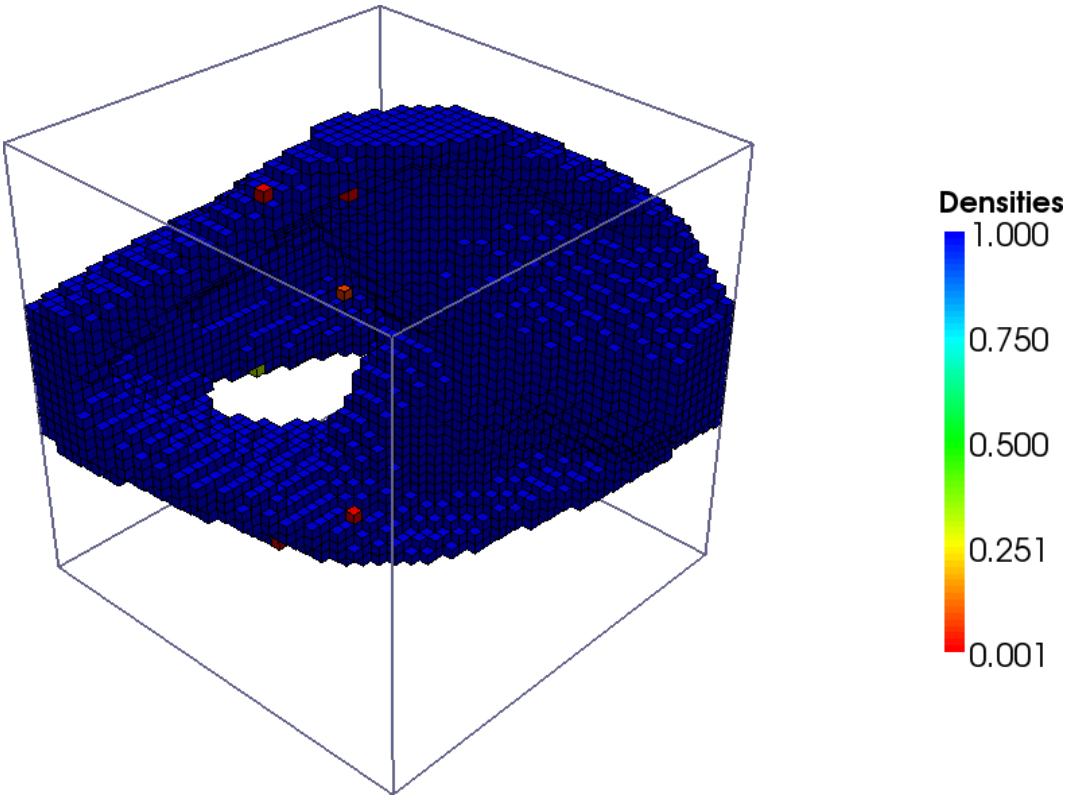


(e) Auxiliary view 3

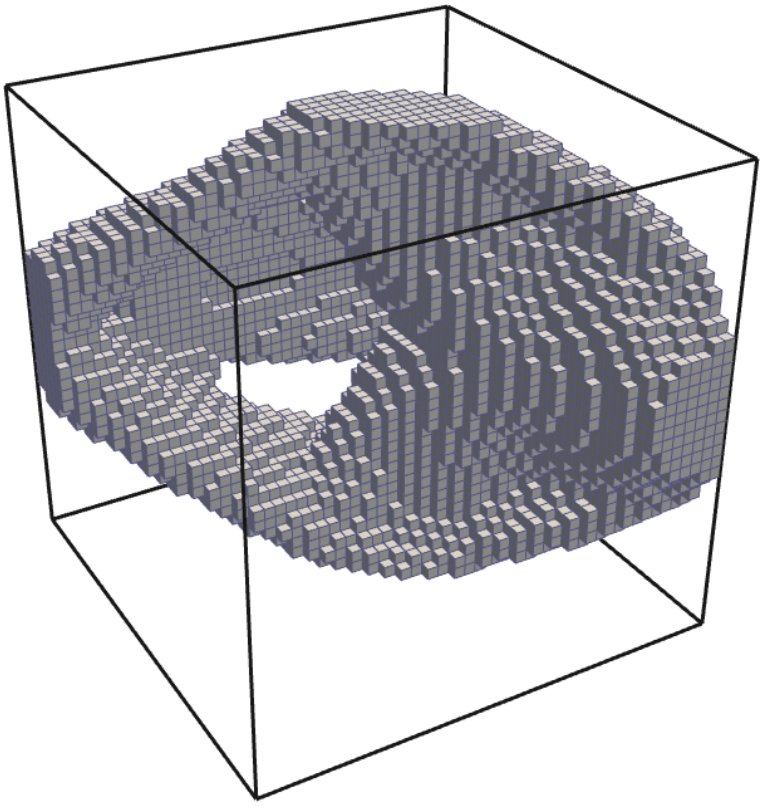


(f) Auxiliary view 6

Figure 6.21: 3D moment arm with GSF; $51 \times 51 \times 51$ 18β-NC elements (page 1/2).



(g) 50 iterations



(h) 50 iterations (solid colour for clarity)

Figure 6.21: 3D moment arm with GSF; $51 \times 51 \times 51$ 18β-NC elements (page 2/2).

6.5 Diagonal quadratic approximations

6.5.1 Heat conduction: solid square plate

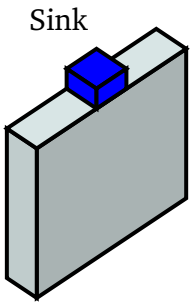
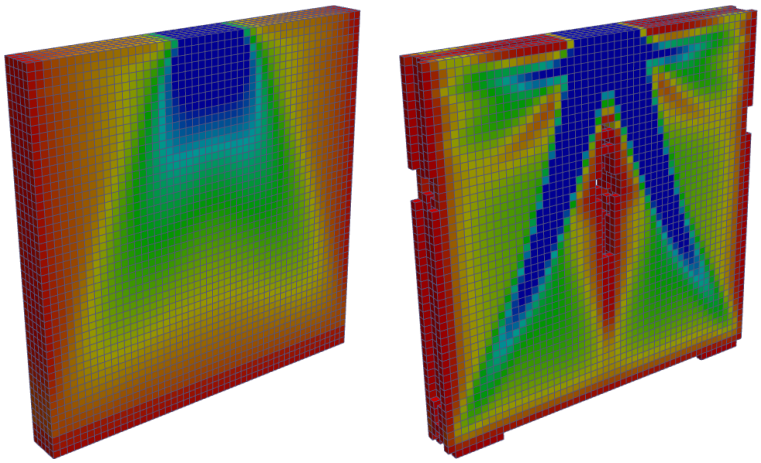


Figure 6.22: 3D heat problem.

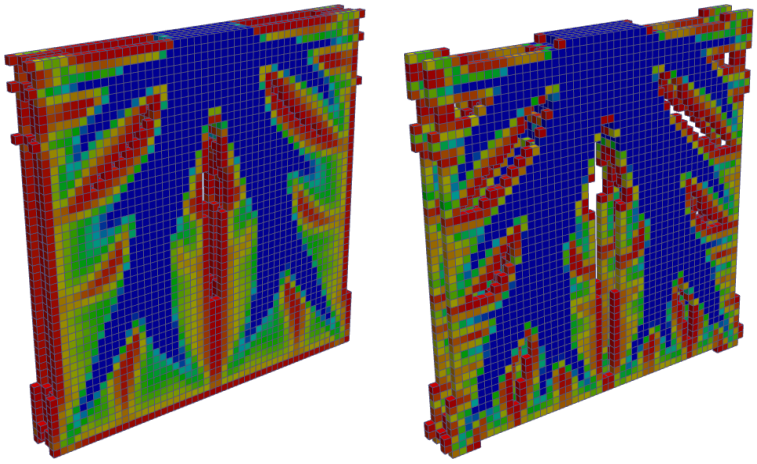
Results for 3D heat conduction problems are challenging to visualise and to present, to say the least. The geometry one gets is usually of such complexity (we can expect this just by looking at the 2D results...) that trying to present it on a two dimensional page is practically impossible. For this reason we will only show a 3D version of the 2D square plate problem, *i.e.*, a square plate with a heat sink on the middle of one face and all nodes are given a thermal load. We set $\eta = 0.4$ as before, but we now make use of quadratic approximation too. The final result is shown in Figure 6.23, for the TPD file see B.7.

6.5.1.1 Discussion

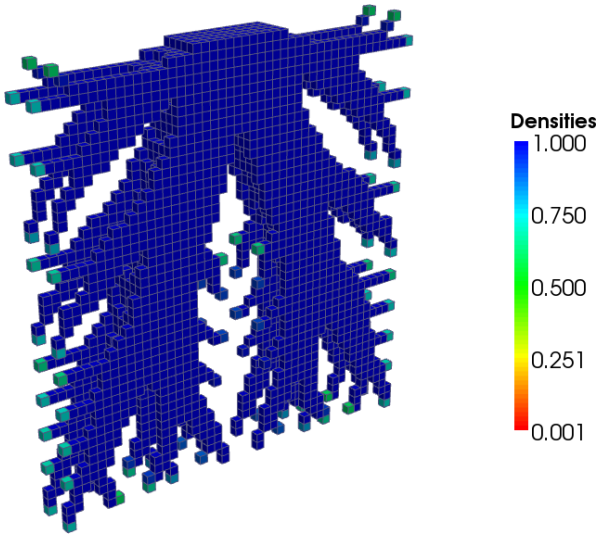
The result is similar to the 2D result of Section 6.1.2.



(a) 15 iterations **(b)** 25 iterations



(c) 35 iterations **(d)** 45 iterations



(e) 100 iterations

Figure 6.23: 3D heat conduction problem, quadratic diagonal approximation with GSF; $60 \times 60 \times 60$ $H8T$ elements.

6.5.2 Mechanism synthesis: inverter

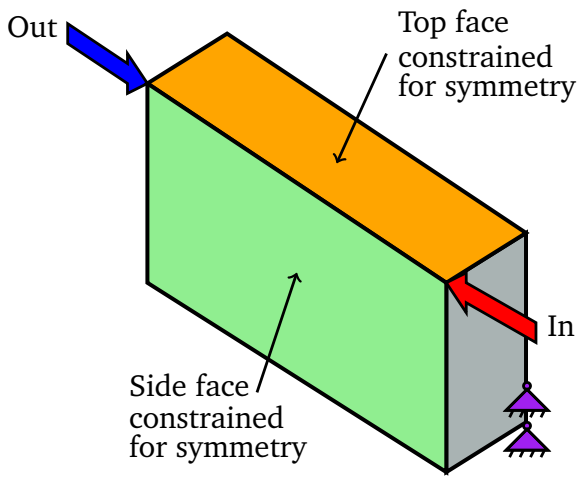
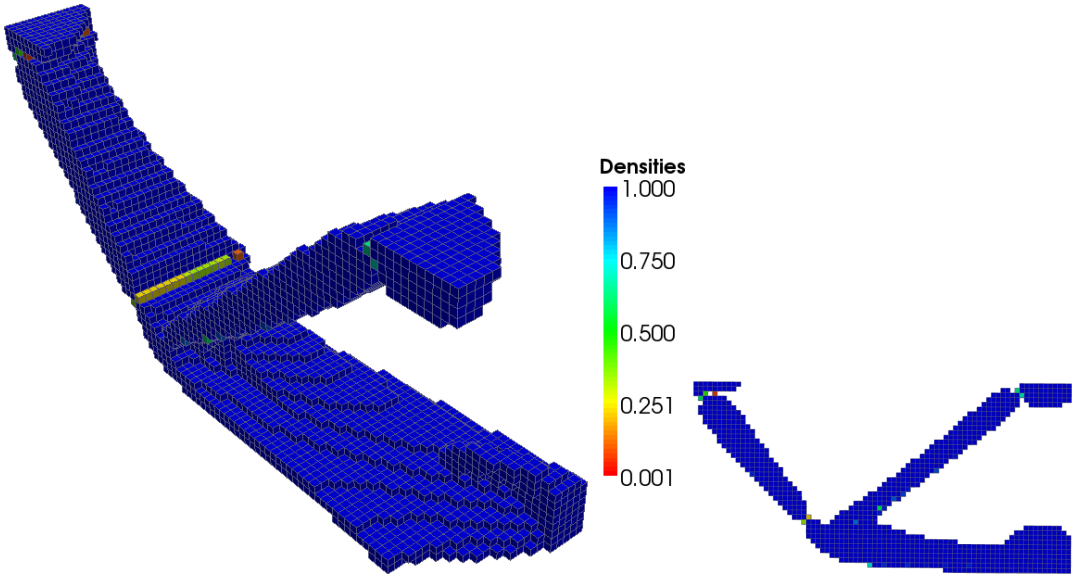


Figure 6.24: 3D mechanism problem.

Three dimensional mechanisms are difficult to depict as well. Strange geometries are obtained in many cases, especially if the domain is not properly constrained – we have found that symmetry constraints are very useful in this regard. We have therefore again chosen a problem similar to the 2D inverter problem, so that we can predict (to some extent) the result. The red (right) arrow indicates the input, and the blue arrow the required output. The forces are parallel to each other and parallel to the top face of the cube, *i.e.*, they lie *in* the plane of the top face. Also, the top and side faces have planar displacement constraints (nodes can move *in* the plane, but not *out* of the plane) in order to model symmetry of the design domain. The final result can be viewed in Figure 6.25.

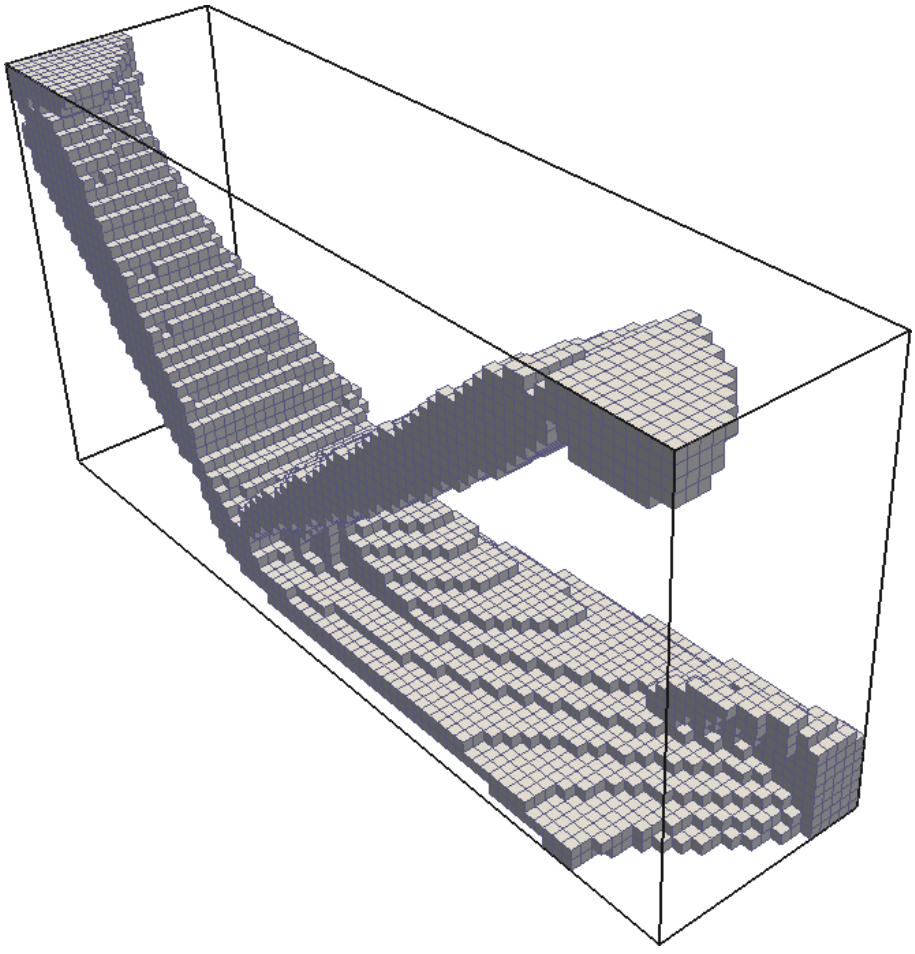
6.5.2.1 Discussion

The result is again similar to the 2D result of section 6.1.3. Note that the locations still having some intermediate density material are also where hinges are required.



(a) 150 iterations

(b) 150 iterations (side view)



(c) 50 iterations (solid colour for clarity)

Figure 6.25: 3D inverter; diagonal quadratic approximation with GSF; $80 \times 40 \times 20$ $H8$ elements.

6.6 High dimensionality problems

ToPy is capable of solving 'large' problems with very high dimensionality, as shown by the two examples below. The first example gives the final solution of a 2D cantilever problem and the second gives the final solution of a 3D torsion bar problem. We optimise for minimum compliance in both cases, with each domain consisting of at least *one million* elements for the 2D and 3D problem respectively.

For these 'large' problems, ToPy's direct sparse solver runs out of memory. We made use of the iterative solver to obtain the results (by making a tiny change in the source code to force the use of the iterative solver). The only real hardware requirement is memory, about 3 Gigabytes on a serial machine is enough.

6.6.1 Minimum compliance: 2D cantilever

Figure 6.26: 2D cantilever problem - 1 000 000 elements (2 005 002 DoF).

The domain in Figure 6.26 is fully constrained on the left with a downward point load in the centre of the domain on the right edge. We show only the final result after 100 iterations and for reciprocal ($\eta = 0.5$) approximation, see Figure 6.27. The choice of the reciprocal approximation here, as opposed to $\eta = 0.4$, is simply because we wish to solve the problem using the 'standard' approach, *i.e.*, not exponential. The result may therefore also serve as a "standard benchmark" for large problems.

Figure 6.27: 2D cantilever with GSF; 2000×500 $Q4$ elements.

Condensed output: ToPy

```
================================================================================
ToPy problem definition (TPD) file successfully parsed.
TPD file name: cantilvr2000x500.tpd (v2007)
--------------------------------------------------------------------------------
Domain discretisation (NUM_ELEM_X x NUM_ELEM_Y) = 2000 x 500
Element type (ELEM_K) = Q4
Filter radius (FILT_RAD) = 50.0
Number of iterations (NUM_ITER) = 100
Problem type (PROB_TYPE) = comp
Problem name (PROB_NAME) = cantilvr2000x500
GSF active
Damping factor (ETA) = 0.50
No passive elements (PASV_ELEM) specified
No active elements (ACTV_ELEM) specified
================================================================================
 Iter | Obj. func.   | Vol.  | Change     | P_FAC | Q_FAC | Ave ETA | S-V frac.
--------------------------------------------------------------------------------
ToPy: Solution for FEA converged after 3357 iterations.
   1  | 5.415890e+02 | 0.500 | 2.0000e-01 | 1.000 | 1.000 |  0.500  |  0.000
ToPy: Solution for FEA converged after 3460 iterations.
   2  | 4.154467e+02 | 0.500 | 2.0000e-01 | 1.000 | 1.000 |  0.500  |  0.000
ToPy: Solution for FEA converged after 3578 iterations.
   3  | 3.585562e+02 | 0.500 | 1.9725e-01 | 1.000 | 1.000 |  0.500  |  0.214
<snip>
ToPy: Solution for FEA converged after 4185 iterations.
  67  | 3.625451e+02 | 0.500 | 2.0000e-01 | 3.000 | 3.100 |  0.500  |  0.999
ToPy: Solution for FEA converged after 4281 iterations.
  68  | 3.625367e+02 | 0.500 | 2.0000e-01 | 3.000 | 3.150 |  0.500  |  1.000
ToPy: Solution for FEA converged after 4177 iterations.
  69  | 3.625310e+02 | 0.500 | 2.0000e-01 | 3.000 | 3.200 |  0.500  |  1.000
<snip>
ToPy: Solution for FEA converged after 0 iterations.
  98  | 3.625222e+02 | 0.500 | 9.8717e-09 | 3.000 | 4.650 |  0.500  |  1.000
ToPy: Solution for FEA converged after 0 iterations.
  99  | 3.625222e+02 | 0.500 | 1.9654e-08 | 3.000 | 4.700 |  0.500  |  1.000
ToPy: Solution for FEA converged after 0 iterations.
 100  | 3.625222e+02 | 0.500 | 1.9478e-08 | 3.000 | 4.750 |  0.500  |  1.000

Solid plus void to total elements fraction = 1.00000
100 iterations took 4068.148 minutes (40.681 min/iter. or 2440.889 sec/iter.)
Average of all ETA's = 0.500 (average of all a's = 1.000)
```

6.6.1.1 Discussion

Probably the most interesting feature of this result is how the top and lower parts taper down towards the point of load, this makes sense given that the bending moment also reduce as one approaches the load. The solution also obtained a solid-void fraction of 1 after only 68 iterations.

6.6.2 Minimum compliance: 3D torsion arm

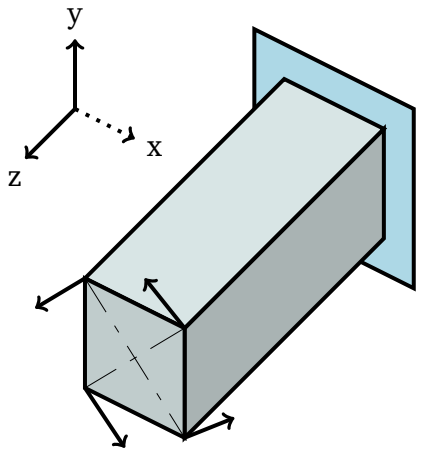


Figure 6.28: 3D torsion arm problem - 1 004 500 elements (3 115 338 DoF).

For the problem depicted in Figure 6.28, we have constrained the domain by 'fixing' it to the wall, but allowing movement in the x direction. Four forces are applied to the corners of the end, causing torsion.

Condensed output: ToPy

```
================================================================================
ToPy problem definition (TPD) file successfully parsed.
TPD file name: torquearm70x70x205.tpd (v2007)
--------------------------------------------------------------------------------
Domain discretisation (NUM_ELEM_X x NUM_ELEM_Y x NUM_ELEM_Z) = 70 x 70 x 205
Element type (ELEM_K) = H8
Filter radius (FILT_RAD) = 1.5
Number of iterations (NUM_ITER) = 50
Problem type (PROB_TYPE) = comp
Problem name (PROB_NAME) = torquearm70x70x205
GSF active
Damping factor (ETA) = 0.40
No passive elements (PASV_ELEM) specified
No active elements (ACTV_ELEM) specified
================================================================================
 Iter | Obj. func.   | Vol.  | Change     | P_FAC | Q_FAC | Ave ETA | S-V frac.
--------------------------------------------------------------------------------
ToPy: Solution for FEA converged after 833 iterations.
   1  | 6.575736e+02 | 0.150 | 2.0000e-01 | 1.000 | 1.000 |  0.400  |  0.000
ToPy: Solution for FEA converged after 881 iterations.
   2  | 3.178322e+02 | 0.150 | 2.0000e-01 | 1.000 | 1.000 |  0.400  |  0.000
ToPy: Solution for FEA converged after 933 iterations.
   3  | 2.230964e+02 | 0.150 | 2.0000e-01 | 1.000 | 1.000 |  0.400  |  0.006
ToPy: Solution for FEA converged after 979 iterations.
<snip>
  47  | 1.446704e+02 | 0.150 | 2.0000e-01 | 3.000 | 2.600 |  0.400  |  1.000
ToPy: Solution for FEA converged after 1610 iterations.
  48  | 1.446668e+02 | 0.150 | 2.0000e-01 | 3.000 | 2.650 |  0.400  |  1.000
ToPy: Solution for FEA converged after 1476 iterations.
```

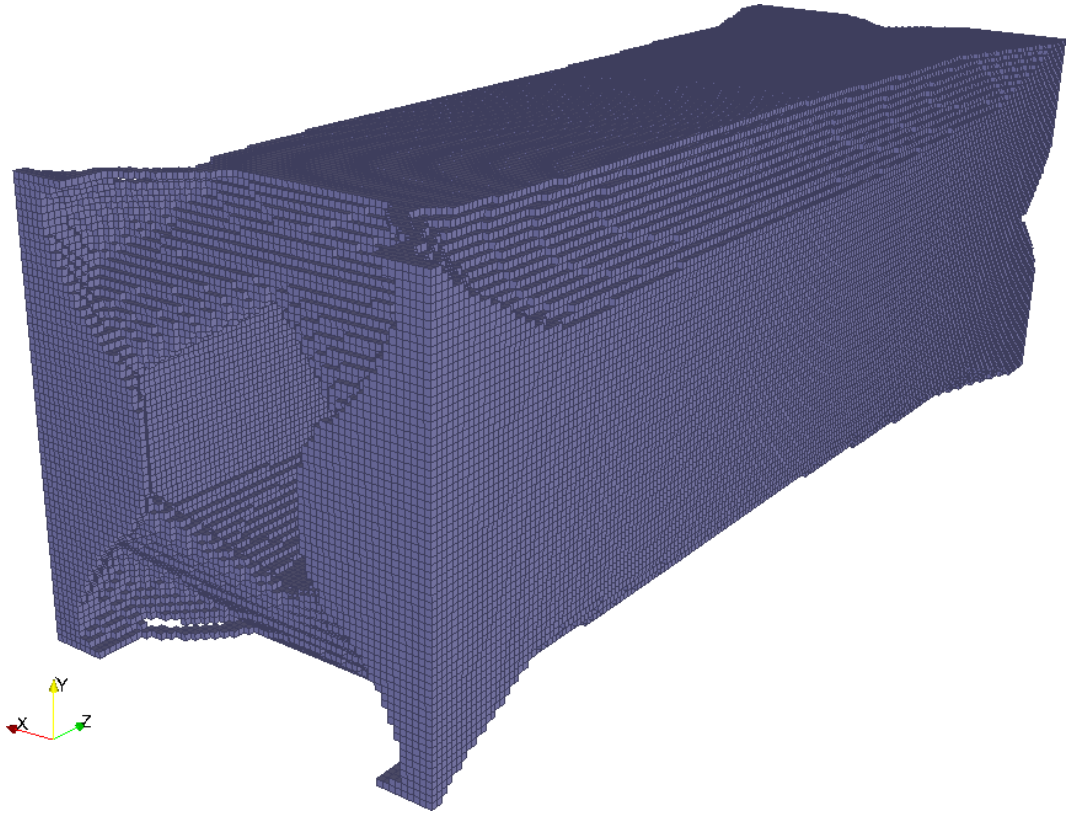
```
  49  | 1.446652e+02 | 0.150 | 1.8493e-01 | 3.000 | 2.700 |  0.400  |  1.000
ToPy: Solution for FEA converged after 1405 iterations.
  50  | 1.446645e+02 | 0.150 | 2.0000e-01 | 3.000 | 2.750 |  0.400  |  1.000

Solid plus void to total elements fraction = 0.99998
50 iterations took 2716.300 minutes (54.326 min/iter. or 3259.560 sec/iter.)
Average of all ETA's = 0.400 (average of all a's = 1.500)
```

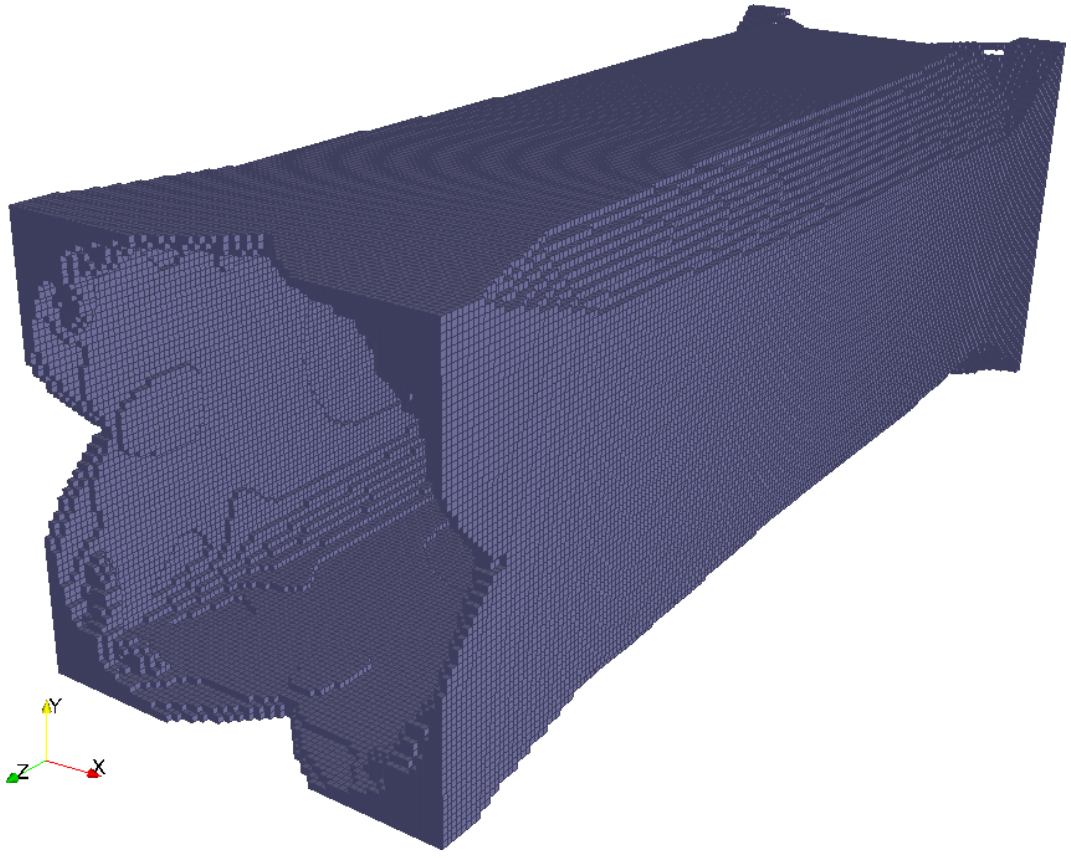
6.6.2.1 Discussion

The result is perhaps not as 'exciting' as one may expect, but it does confirm what we already know: tube-like cross-sections are superior in their torsional stiffness compared to other cross-sections. Also, note how virtually no material is present in the x direction against the wall, because the domain was not constrained in this direction (wall view in Figure 6.29). All the torsion in the domain has to be resisted in the y direction, notice the 'gussets' protruding from the top and bottom sides of the tubing towards the wall in order to counter angular deflection.

Also, note that ratio of DoF's of 3D to 2D is $\frac{3115338}{2005002} = 1.554$; this is also reflected in the solution time per iteration, namely $\frac{3259.560}{2440.889} = 1.335$.

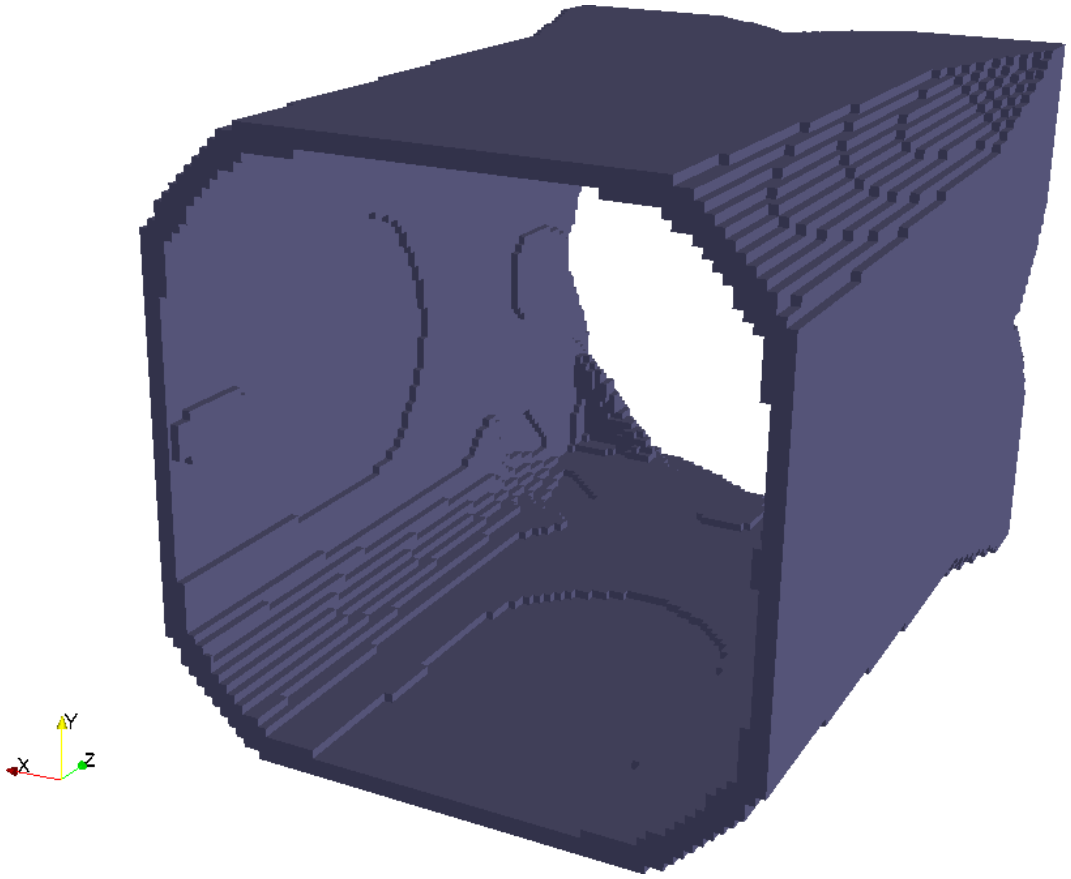


(a) 50 iterations (wall view)

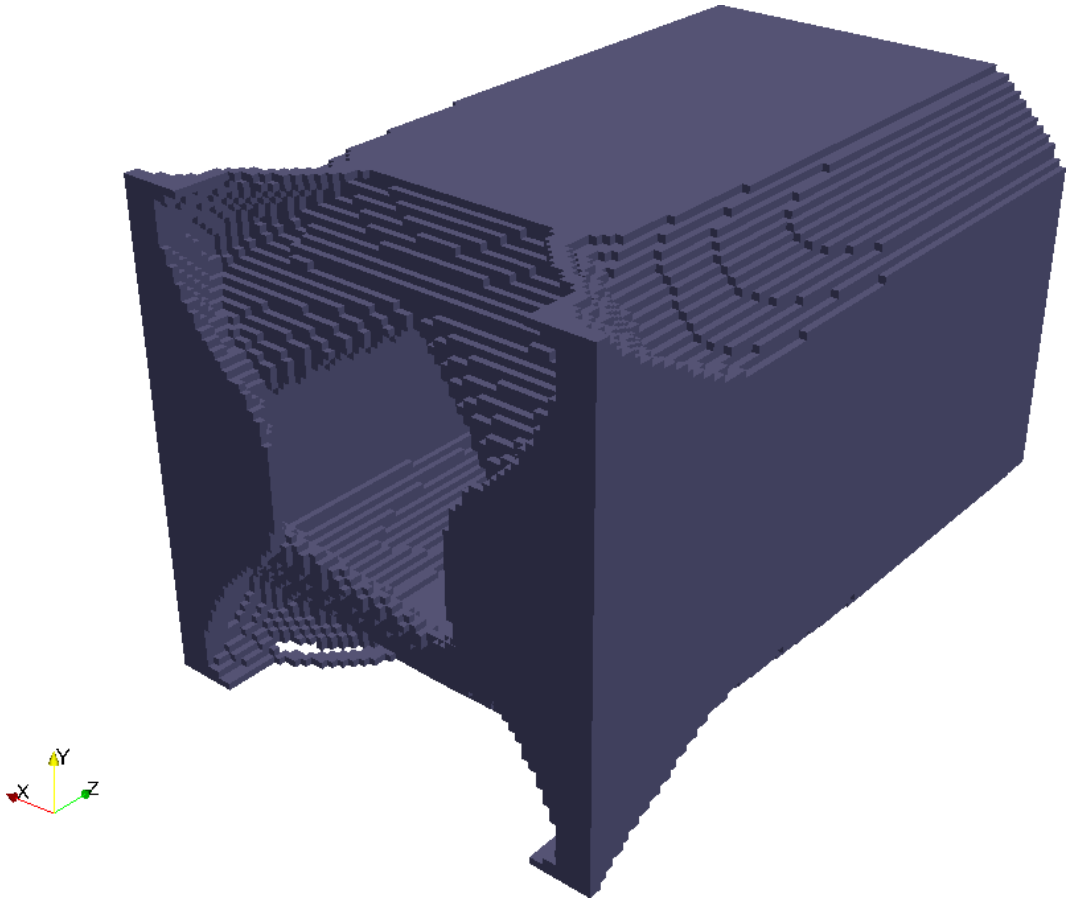


(b) 50 iterations (front view)

Figure 6.29: 3D torque arm with GSF; $70 \times 70 \times 205$ $H8$ elements (page 1/2).



(c) 50 iterations (clipped view 1)



(d) 50 iterations (clipped view 2)

Figure 6.29: 3D torque arm with GSF; $70 \times 70 \times 205$ $H8$ elements (page 2/2).

Chapter

7

Conclusions and future work

Inspired by Sigmund's 99-line MATLAB code for minimum compliance topology optimisation, an OSS version was developed in Python, denoted ToPy. ToPy extends on the 99-line code of Sigmund, in that ToPy can solve different problems, *e.g.*, minimum compliance, heat conduction and mechanism synthesis in 2D or 3D, by simply changing an input file. In other words, no additional coding is required. Also, by using established OSS (Pysparse and its iterative solver) for solving the sparse finite element (FE) systems of equations that need to be solved, the ToPy code provides improved speed and scalability.

Additional enhancements were made by incorporating the gray-scale filtering (GSF) technique of Groenwold and Etman to yield predominantly, or even purely, solid-void or black-and-white designs in 2D and 3D.

In addition, an exponential approximation to the objective function was implemented. This approximation is a generalisation of the reciprocal approximation so popular in structural optimisation; it uses the method proposed by Fadel *et al.* to determine the unknown exponents. Alternatively, fixed exponents may be prescribed, in the spirit of optimality criterion (OC) methods.

As a further generalisation, the diagonal quadratic approximation of the exponential approximation recently proposed by Groenwold and Etman in an SAO setting was also implemented. What is more: the diagonal quadratic approximation to the exponential approximation was successfully used in combination with GSF. This is a novelty of some importance, since Groenwold and Etman had previously suggested that GSF can only be used in combination with strictly monotonic objective functions, like the reciprocal approximation. This approach was successfully used to solve a 3D mechanism synthesis problem, which yielded a result analogous to a 2D result obtained using the usual reciprocal (or equivalently exponential) approach.

Interesting problems for future consideration include using the diagonal quadratic approximation for problems with multiple constraints, *e.g.*, the addition of constraints on stress. This may involve using a general unconstrained optimisation method, such as Polak-Rebiére or a BFGS search, to determine the optimal parameters.

Other possibilities for future work include:

- Developing a custom graphical user interface for pre-processing, and so providing a graphical environment to create TPD files. As an alternative, established OSS pre-processors, such as GMSH or Salome-Platform, may be utilised to the same effect.
- The current designs obtained via ToPy cannot be manufactured without violating the volume constraint. This may be remedied by careful (while not violating the volume constraint) post-processing of the design to extract and smooth the boundary surface, using methods such as *marching cubes* and NURBS or subdivision. This may well render manufacturable designs.
- Parallelisation of the FEA routine to further increase speed and scalability. Various OSS Python solutions exist, such as PyTrilinos and PyMPI.
- The implementation of a sensitivity filter for elements of general shape and size. In addition, this will require the facility for isoparametric finite elements, which will enable the user to optimise existing structures via a CAD interface.

List of References

Beckers, M. (1996). Topology optimization using a dual method with discrete variables. *Structural Multidisciplinary Optimization*, vol. 11, pp. 102–112. (Cited on page 6).

Bendsøe, M.P. (1989). Optimal shape design as a material distribution problem. *Structural Multidisciplinary Optimization*, vol. 1, pp. 193–202. (Cited on pages 5, 7, and 17).

Bendsøe, M.P. (1995). *Optimization of structural topology, shape and material*. Springer, Berlin, Heidelberg, New York. (Cited on page 18).

Bendsøe, M.P. and Kikuchi, N. (1988). Generating optimal topologies in structural design using a homogenization method. *Computer Methods in Applied Mechanics and Engineering*, vol. 71, pp. 197–224. (Cited on page 5).

Bendsøe, M.P. and Sigmund, O. (2004). *Topology Optimization, Theory, Methods and Applications*. 2nd edn. Springer-Verlag, Berlin. (Cited on pages 3, 5, 7, 8, 9, 36, 45, and 48).

Bruns, T.E. (2005). A reevaluation of the SIMP method with filtering and an alternative formulation for solid-void topology optimization. *Structural Multidisciplinary Optimization*, vol. 30, pp. 428–436. (Cited on page 6).

Cook, R.D. (1995). *Finite Element Modeling for Stress Analysis*. 1st edn. John Wiley & Sons, Inc., New York. (Cited on pages 24 and 28).

Cook, R.D., Malkus, D.S., Plesha, M.E. and Witt, R.J. (2002). *Concepts and Applications of Finite Element Analysis*. 4th edn. John Wiley & Sons, Inc., New York. (Cited on page 28).

De Klerk, A. and Groenwold, A. (2004 July). On the robustness of the Q4 membrane element. In: Zingoni, A. (ed.), *Progress in Structural Engineering, Mechanics and Computation*, pp. 961–966. Cape Town, South Africa. (Cited on page 24).

Di, S. and Ramm, E. (1994). On alternative hybrid stress 2d and 3d elements. *Engineering Computations*, vol. 11, pp. 49–68. (Cited on page 28).

Falk, J.E. (1967). Lagrange multipliers and nonlinear programming. *Journal of Mathematical Analysis and Applications*, vol. 19, pp. 141–159. (Cited on page 14).

Fraeijs du Veubeke, B. (1965). Displacement and equilibrium models in the finite element method. In: Zienkiewicz, O.C. and Holister, G.C. (eds.), *Stress Analysis*, pp. 145–197. John Wiley & Sons, Inc., Chichester. (Cited on page 25).

Groenwold, A.A. and Etman, L.F.P. (2007). A quadratic approximation for topology optimization. *International Journal for Numerical Methods in Engineering*. Submitted. (Cited on page 22).

Groenwold, A.A. and Etman, L.F.P. (2008). On the equivalence of optimality criterion and sequential approximate optimization methods in the classical topology layout problem. *International Journal for Numerical Methods in Engineering*, vol. 73, pp. 297–316. (Cited on pages 11, 17, and 18).

Groenwold, A.A. and Etman, L.F.P. (2009). A simple heuristic for gray-scale suppression in optimality criterion-based topology optimization. *Structural Multidisciplinary Optimization*. To appear. (Cited on pages 6, 19, 20, and 21).

Groenwold, A.A., Etman, L.F.P. and Wood, D.W. (2008). Approximated approximations for SAO. *Computer Methods in Applied Mechanics and Engineering*. Submitted. (Cited on page 22).

Haftka, R.T. and Gürdal, Z. (1992). *Elements of Structural Optimization*. 3rd edn. Kluwer Academic Publishers Group, The Netherlands. (Cited on pages 11, 14, and 48).

Hetland, M.L. (2005). *Beginning Python: From Novice to Professional*. 1st edn. Apress, Berkeley, CA. (Cited on page 90).

Hughes, T.J.R. (2000). *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. 1st edn. Dove Publications, Inc., New York. (Cited on page 28).

Long, C.S., Loveday, P.W. and Groenwold, A.A. (2008). Effects of planar element formulation and numerical integration order on checkerboard material layouts. *Structural Multidisciplinary Optimization*, vol. DOI 10.1007/s00158-008-0345-1. Submitted. (Cited on page 5).

Michell, A.G.M. (1904). The limit of economy of material in frame structures. *Philosophical Magazine*, vol. 8, pp. 589–597. (Cited on page 58).

Mills, A.F. (1995). *Heat and Mass Transfer*. 1st edn. Richard D. Irwin, Inc., Chicago. (Cited on page 8).

Rossow, M.P. and Taylor, J.E. (1973). A finite element method for the optimal design of variable thickness sheets. *AIAA*, vol. 11, pp. 1566–1569. (Cited on page 5).

Rozvany, G.I.N. (2000 May). Problem classes, solution strategies and unified terminilogy of fe-based topology optimization. In: Rozvany, G.I.N. and Olhoff, N. (eds.), *Topology Optimization of Structures and Composite Continua*, vol. 7. NATO, Kluwer Academic Publishers Group, The Netherlands. (Cited on page 1).

Rozvany, G.I.N. (2009). A critical review of established methods of structural topology optimization. *Structural Multidisciplinary Optimization*, vol. 37, pp. 217–237. Published online 21 February 2008 (http://www.springerlink.com/content/t1123m1267501484). (Cited on page 5).

Rozvany, G.I.N. and Zhou, M. (1991). Applications of COC method in layout optimization. In: Eschenauer, H., Mattheck, C. and Olhoff, N. (eds.), *Proc. Engineering Optimization in Design Processes*, pp. 59–70. Springer-Verlag, Berlin. (Cited on pages 5 and 7).

Rozvany, G.I.N., Zhou, M. and Birker, T. (1992). Generalized shape optimization without homogenization. *Structural Multidisciplinary Optimization*, vol. 4, pp. 250–254. (Cited on page 5).

Sigmund, O. (2001). A 99 line topology optimization code written in Matlab. *Structural Multidisciplinary Optimization*, vol. 21, pp. 120–127. (Cited on pages 1 and 7).

Sigmund, O. (2007). Morphology-based black and white filters for topology optimization. *Structural Multidisciplinary Optimization*, vol. 33, pp. 401–424. (Cited on page 6).

Svanberg, K. and Werme, M. (2005). A hierarchical neighborhood search method for topology optimization. *Structural Multidisciplinary Optimization*, vol. 29, pp. 325–340. (Cited on page 6).

Mathematica (v 6). Compute and visualise CAS. [Accessed 2008].
Available at: `http://www.wolfram.com` (Cited on page 28).

wxMaxima (v 071). GUI front-end to maxima CAS. [Accessed 2008]. Available at: http://wxmaxima.sourceforge.net (Cited on pages 28 and 130).

Vanderplaats, G.N. (2001). *Numerical Optimization Techniques for Engineering Design*. 3rd edn. VR&D, Inc., Colorado Springs. (Cited on pages 3 and 11).

Wang, M., Zhou, S. and Ding, H. (2004). Nonlinear diffusions in topology, optimization. *Structural Multidisciplinary Optimization*, vol. 28, pp. 262–276. (Cited on page 6).

Wu, C.-C. and Cheung, Y.K. (1995). On optimization approaches of hybrid stress elements. *Finite Elements in Analysis and Design*, vol. 21, pp. 111–128. (Cited on pages 25 and 26).

Zhou, M. and Rozvany, G.I.N. (1991). The COC method. Part II. Topological, geometrical and generalized shape optimization. *Computer Methods in Applied Mechanics and Engineering*, vol. 40, pp. 1–26. (Cited on page 6).

Zhou, M., Shyy, Y.K. and Thomas, H.L. (2001). Checkerboard and minimum member size control in topology optimization. *Structural Multidisciplinary Optimization*, vol. 21, pp. 152–158. (Cited on page 6).

Appendices

Appendix

A

Software installation, use and source code

This chapter covers the software, ToPy (topology optimization with Python), that was developed by the author for solving the problems as explained in Chapter 2, and also how to use ToPy. We also give a list other useful software that can be used in conjunction with ToPy.

This appendix should be read together with Chapter 5.

A.1 Installation and use of ToPy

A.1.1 Download

ToPy can be obtained from the author, William Hunter, at the following e-mail address:

willemjagter@gmail.com.

At the time of writing, the latest version is `ToPy-0.1.0.tar.gz`.

A.1.2 Installation

First make sure you have all the required software to use ToPy, see Section A.2 a little further down.

The standard way to install any Python package is as follows: Copy and save the tarball (usually a '.tar.gz' file) to a directory of your choice and do the following:

A.1.2.1 GNU/Linux instructions

Launch a terminal and `cd` to the directory where you saved the tarball:

1. Unpack the tarball:

   ```
   tar -xvvf ToPy-0.1.0.tar.gz
   ```

2. Install ToPy:

   ```
   sudo python setup.py install
   ```

If no errors were reported, you successfully installed the package.

A.1.2.2 Windows instructions

First *unzip* the tarball to a directory of your choice using something like WinZip. Launch a terminal (command line prompt) by right-clicking on the 'Start' button, selecting "Run command" and typing `cmd.exe`, then `cd` to the directory where you unzipped the tarball and type:

```
python setup.py install
```

On a Microsoft Windows operating system, you might have to set your `PYTHONPATH` environment variable as well. There are a couple of ways of doing this, but the easiest is probably to edit the `autoexec.bat` file directly, which you can find in the root (top) directory of the C drive (assuming you have a relatively standard setup). Using any editor, *e.g.* Notepad, and say you want to add the directory `C:\python\myprogs` to the file, then type:

```
set PYTHONPATH=%PYTHONPATH%;C:\python\myprogs
```

A.2 ToPy requirements

All software required to use ToPy is available and in the public domain.

In order to use ToPy, Python 2.5 and a few other related software packages are required. Python should already be installed if you are using a GNU/Linux machine. On Microsoft Windows you will most probably have to install it first. Python can be downloaded from `http://www.python.org/`.

A.2.1 Scientific Python software

These should be available as binaries on most platforms, select and install the following packages:

1. NumPy, "provides convenient and fast N-dimensional array manipulation" (see `http://www.scipy.org/`).

2. Pysparse, "a sparse matrix library for Python" (see `sourceforge.net/projects/pysparse/`).

3. Matplotlib is a "Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments" (see `http://matplotlib.sourceforge.net/`).

4. PyVTK "provides tools for manipulating Visualization Toolkit (VTK) files in Python" (see `http://cens.ioc.ee/projects/pyvtk/`).

A.2.1.1 Linux instructions

On Ubuntu 8.04 LTS, a Debian based distribution of Linux, simply type the following command in a terminal (or alternatively make use of a graphical user interface such as Synaptic Package Manager and select and install the following three packages):

```
sudo apt-get install python-numpy python-matplotlib python-pyvtk
```

You can also choose to build NumPy and Matplotlib from source, which will give you the latest versions since the ones in your distribution's repository may be outdated (the development of NumPy and Matplotlib is rapid). ToPy was tested against the versions in the Ubuntu 8.04 LTS repositories as well as the following (latest, at the time of writing) versions:

NumPy version 1.2.1

Matplotlib version 0.98.3

Note the above version of NumPy contains API changes that will cause Pysparse to give a warning when solving 3D problems, this does not affect the results or performance whatsoever. Future versions of Pysparse will remedy this.

In order to compile easily from source, type the following command to drag in all the required dependencies:

```
sudo apt-get build-dep python-numpy python-matplotlib
```

Download and unpack the source tarballs for NumPy and Matplotlib, change to the numpy-1.2.1 (the latest version at the time of writing this) directory and type:

```
python setup.py build
```

If no errors were reported, then type:

```
sudo python setup.py install --prefix=/usr/local
```

You can omit '-prefix=/usr/local' part, in which case the package will be installed in the standard location, either will work. Now do the same for Matplotlib (unpack, build and install).

For sparse matrix computations and solving the large sparse systems, we make use of a package called Pysparse, that gives us access to SuperLU and UMFpack as well as iterative solvers. This package has to be compiled from source:

1. Install ATLAS (Automatically Tuned Linear Algebra Software) development headers:

   ```
   sudo apt-get install atlas3-base-dev
   ```

2. Download Pysparse, unpack, build and install:

   ```
   python setup.py build
   sudo python setup.py install --prefix=/usr/local
   ```

   Note: On Ubuntu systems you might have to edit the 'setup.py' file, by looking for and editing the following line to tell Pysparse where the ATLAS library lives:

   ```
   ...
   # default settings
   library_dirs_list= ['/usr/lib/atlas']
   ...
   ```

Other GNU/Linux distributions will have a very similar installation procedure.

A.2.1.2 Windows instructions

All packages are available as 'installers' from the same download URL's, simply download and double-click...

A.2.2 Other

A.2.2.1 Viewing 3D results (VTK files)

ParaView (or any other viewer that supports Kitware's VTK legacy format) may be used in order to view the 3D results as produced by ToPy. At the time of writing this, it is available for download at http://www.paraview.org. ParaView 3.2.1 as well as version 3.4.0 were used to view the results. ParaView is extremely useful for visualising scientific data in general. Below in Figure A.1 is a screenshot of ParaView on GNU/Linux.

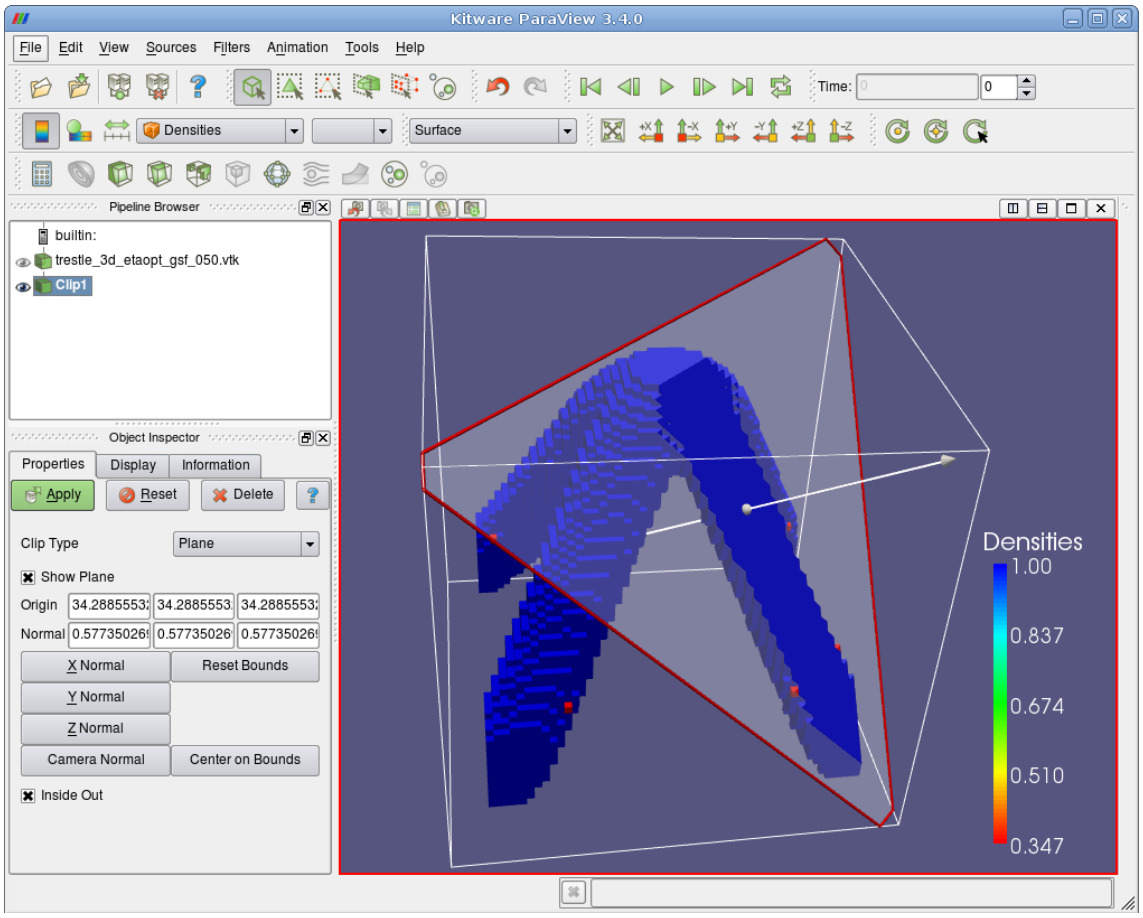


Figure A.1: ParaView in action — clipping a 3D trestle leg to inspect the interior for voids.

Alternatively, you may choose to use MayaVi1 or MayaVi2 – "MayaVi is a general purpose, cross-platform tool for 2D and 3D scientific data visualization". It can be downloaded from `http://mayavi.sourceforge.net`. On Ubuntu, to install MayaVi1;

```
sudo apt-get install mayavi
```

or for MayaVi2:

```
sudo apt-get install mayavi2
```

You can also use easy_install.

Further, any viewer that can read legacy VTK files will work.

A.2.2.2 Creating animated GIF's

To create GIF animations, use ImageMagick's *convert* utility ImageMagick can be installed as follows (or downloaded from `www.imagemagick.org`):

```
sudo apt-get install imagemagick
```

Once you have opened a sequence of VTK files with ParaView, you can play an animation and save it as an AVI or as a sequence of PNG's (for 2D problems ToPy creates PNG images by default). Convert the PNG's to an animated GIF by typing the following command in a terminal:

```
convert -delay 35 *.png <some file name>.gif
```

The *convert* approach is less error prone than the AVI one, it also produces cross-platform, smaller (by about 60% in most cases) and better quality animations. Then simply open the GIF file with any web standard compliant internet browser (*e.g.*, Opera or Mozilla Firefox).

A.3 How to use ToPy

To use ToPy is easy since no programming is actually required by the user to define the problem, instead, a ToPy problem definition file (a TPD file) is created which is then parsed and solved by ToPy.

A.3.1 Example

The listing below is for the 'standard' 2D 60x20 MBB beam, 10 iterations, reciprocal approximation (`ETA = 0.5`):

```
[ToPy Problem Definition File v2007]

# Author: William Hunter
# The 'classic' 60x20 2d mbb beam, as per Ole Sigmund's 99 line code.

PROB_TYPE  : comp
PROB_NAME  : beam_2d_reci
ETA        : 0.5  # reciprocal approx.
DOF_PN     : 2
VOL_FRAC   : 0.5
FILT_RAD   : 1.5
ELEM_K     : Q4
NUM_ELEM_X : 60
NUM_ELEM_Y : 20
NUM_ELEM_Z : 0
FXTR_NODE_X: 1|21
FXTR_NODE_Y: 1281
LOAD_NODE_Y: 1
LOAD_VALU_Y: -1

NUM_ITER   : 10

P_FAC      : 3
```

The listing above may be saved as, *e.g.*, `beam_2d_reci_10.tpd`, then type the following command to run ToPy and to solve the problem—in a terminal:

```
python optimise.py beam_2d_reci_10.tpd
```

Assuming you installed all the prerequisites and of course ToPy itself, you should start seeing output within a moment. ToPy will perform some rudimentary checking of the TPD file, mostly checking for the minimum required amount of keywords, typos and inconsistent loading and constraints, but it is obviously not fool proof.

The `optimise.py` file is a Python script that calls the relevant ToPy functions. This file can be modified by the user if he/she wishes. The contents of `optimise.py` is shown below:

```
#!/usr/bin/env python

# Author: William Hunter
# This script handles both number of iterations and change stop criteria runs


# Import required modules:
from sys import argv

from time import time

from numpy import array

from matplotlib import pyplot as pp

import topy


# Set up ToPy:
t = topy.Topology()
t.load_tpd_file(argv[1])
t.set_top_params()

# Create empty list for later use:
etas_avg = []

# Optimising function:
def optimise():
    t.fea()
    t.sens_analysis()
    t.filter_sens_sigmund()
    t.update_desvars_oc()
    # Below this line we print and create images:
    if t.nelz:
        topy.create_3d_geom(t.desvars, prefix=t.probname, \
        iternum=t.itercount, time='none')
    else:
        topy.create_2d_imag(t.desvars, prefix=t.probname, \
        iternum=t.itercount, time='none')
    print '%4i   | %3.6e | %3.3f | %3.4e | %3.3f | %3.3f |  %1.3f   |  %3.3f '\
    % (t.itercount, t.objfval, t.desvars.sum()/(t.nelx * t.nely * nelz), \
    t.change, t.p, t.q, t.eta.mean(), t.svtfrac)
    # Build a list of average etas:
    etas_avg.append(t.eta.mean())

# Create (plot) initial design domain:
if t.nelz:
    #create_3d_geom(t.desvars, prefix=t.probname, iternum=0, time='none')
    nelz = t.nelz
else:
```

```
    #create_2d_imag(t.desvars, prefix=t.probname, iternum=0, time='none')
    nelz = 1 #  else we divide by zero

# Start optimisation runs, create rest of design domains:
print '%5s | %11s | %5s | %10s | %5s | %5s | %7s | %5s ' % ('Iter', \
'Obj. func.  ', 'Vol. ', 'Change    ', 'P_FAC', 'Q_FAC', 'Ave ETA', 'S-V frac.')
print '-' * 79
ti = time()

# Try CHG_STOP criteria, if not defined (error), use NUM_ITER for iterations:
try:
    while t.change > t.chgstop:
        optimise()
except AttributeError:
    for i in range(t.numiter):
        optimise()
te = time()

# Print solid-void ratio info:
print '\nSolid plus void to total elements fraction = %3.5f' % (t.svtfrac)
# Print iteration info:
print t.itercount, 'iterations took %3.3f minutes (%3.3f min/iter. \
or %3.3f sec/iter.)'\
%((te - ti) / 60, (te - ti) / 60 / t.itercount, (te - ti) / t.itercount)
print 'Average of all ETA\'s = %3.3f (average of all a\'s = %3.3f)' \
% (array(etas_avg).mean(), 1/array(etas_avg).mean() - 1)
```

The terminal output for the first 10 iterations is shown below:

```
================================================================================
ToPy problem definition (TPD) file successfully parsed.
TPD file name: beam_2d_reci_10_iters.tpd (v2007)
--------------------------------------------------------------------------------
Domain discretisation (NUM_ELEM_X x NUM_ELEM_Y) = 60 x 20
Element type (ELEM_K) = Q4
Filter radius (FILT_RAD) = 1.5
Number of iterations (NUM_ITER) = 10
Problem type (PROB_TYPE) = comp
Problem name (PROB_NAME) = beam_2d_reci
Continuation of penalisation factor (P_FAC) not specified
GSF not active
Damping factor (ETA) = 0.50
No passive elements (PASV_ELEM) specified
No active elements (ACTV_ELEM) specified
================================================================================
 Iter | Obj. func.   | Vol.  | Change     | P_FAC | Q_FAC | Ave ETA | S-V frac.
--------------------------------------------------------------------------------
   1  | 1.007023e+03 | 0.500 | 2.0000e-01 | 3.000 | 1.000 |  0.500  |  0.000
   2  | 5.819346e+02 | 0.500 | 2.0000e-01 | 3.000 | 1.000 |  0.500  |  0.000
   3  | 4.145937e+02 | 0.500 | 2.0000e-01 | 3.000 | 1.000 |  0.500  |  0.115
   4  | 3.457323e+02 | 0.500 | 2.0000e-01 | 3.000 | 1.000 |  0.500  |  0.153
   5  | 3.232380e+02 | 0.500 | 1.9465e-01 | 3.000 | 1.000 |  0.500  |  0.175
   6  | 3.090768e+02 | 0.500 | 2.0000e-01 | 3.000 | 1.000 |  0.500  |  0.193
   7  | 2.986085e+02 | 0.500 | 1.7186e-01 | 3.000 | 1.000 |  0.500  |  0.212
   8  | 2.886144e+02 | 0.500 | 1.9676e-01 | 3.000 | 1.000 |  0.500  |  0.239
   9  | 2.798081e+02 | 0.500 | 1.3974e-01 | 3.000 | 1.000 |  0.500  |  0.258
  10  | 2.714769e+02 | 0.500 | 1.3511e-01 | 3.000 | 1.000 |  0.500  |  0.296
```

```
Solid plus void to total elements fraction = 0.29583
10 iterations took 0.076 minutes (0.008 min/iter. or 0.456 sec/iter.)
Average of all ETA's = 0.500 (average of all a's = 1.000)
```

A.3.2 TPD file format

The TPD file can be created in any text editor. Here follows verbatim the entries of a template TPD file. The template also explains the format in detail.

```
[ToPy Problem Definition File v2007]

# ==============================================================================
# Author: William Hunter <willemjagter@gmail.com>
# Copyright (C) 2008, William Hunter.
#
# Template file explaining the ToPy Problem Definition file format.
#
# The file *must* start with the identifier '[ToPy Input File v2007]' and then
# a blank line.
#
# Comments may be placed after the the hash (#) character like these lines and
# inline comments are also supported.
#
# Order of keywords (parameters) may be random, but keywords need to be first
# in sequence, e.g:
# KEYWORD1: some value
# KEYWORD2: another value
# KEYWORD3: yet another value
# KEYWORD4: etc.
#
# In this template file, all available keywords (parameters) are shown. See
# 'minimal.tpd' in this directory for a file with a minimum mandatory
# amount of entries.
#
# There is no restriction on the use of whitespace, but <TAB>s
# will create problems, thus, make sure your editor replaces <TAB>s with
# <SPACE>s (the ToPy parser will warn you if it finds <TAB>s anyway).
#
# NOTE: This template does not actually define a valid problem!
# ==============================================================================

# ====================
# === Problem type ===
# ====================
# Is one of 'comp' = minimum compliance problem,
#           'heat' = heat conduction problem or
#           'mech' = mechanism design (syntehesis) problem.
# The keyword-value case doesn't matter, i.e., comp = CoMp = COMP.
PROB_TYPE: comp #  Solve a minimum compliance problem.
# Problem name:
PROB_NAME: really_cool_problem  # Output files will have this name.

# ====================================
# === Problem parameters (keywords) ===
# ====================================
VOL_FRAC: 0.5  #  The volume fraction to be used.
FILT_RAD: 1.5  #  Filter radius.

# Use one of the following:
NUM_ITER: 100  #  Number of iterations to run.
CHG_STOP: 0.01 #  Change stop value, checks the change in obj. function value.
```

```
P_FAC : 3    #  Start value of penalty factor (p).
P_MAX : 3.5  #  Max value of P_FAC.
P_INCR: 0.02 #  Increment value of P_FAC.
P_CON : 25   #  Number of iterations to keep P_FAC constant after increment.
Q_FAC : 1    #  Start value of extra penalty factor (q) for GSF.
Q_MAX : 5    #  Analoguous to P_MAX.
Q_INCR: 0.08 #  Analoguous to P_INCR.
Q_CON : 20   #  Analoguous to P_CON.

ETA   : 0.5   #  0.001, use 0.5 for reciprocal approximation.
ETA   : exp   #  Use exponential approximation, eta is 'auto-tuned'
APPROX: dquad #  Use diagonal quadratic approximation, ETA must be specified.

# ============================
# === Finite Element Types ===
# ============================
# Nodes are numbered as follows, although this is not important to the user.
#
# 2D: Y             3D: Y
#     |                 |
#   4-|-3             4-|-3
#   | +-|---X        /| +-|---X
#   1---2            / 1/--2
#                   8--/7 /
#                   | / |/
#                   5/--6
#                   /
#                  Z
#
ELEM_K: Q4 #  Other 2D: Q5B, Q4a5B, Q4T.   3D: H8, H18B, H8T.

# ========================================
# === Discretisation of the design domain ===
# ========================================
# 2D: Y             3D: Y
#     |                 |
#     +---X             +---X
#                      /
#                     Z
#
# 1---5---9
# | 1 | 5 |
# 2---6---10
# | 2 | 6 |
# 3---7---11
# | 3 | 7 |
# 4---8---12
#
# Numbering of nodes and elements is from top to bottom, columnwise, starting
# at one (1).
# For 3D, the X-Y plane is numbered first, then in the Z-direction.
NUM_ELEM_X: 60 #  Number of elements in the X-direction.
NUM_ELEM_Y: 20 #  Number of elements in the (negative) Y-direction.
# Set the following keyword to 0 if not necessary for your problem, i.e., 2D:
NUM_ELEM_Z: 10

# ================================
# === Translational constraints ===
# ================================
```

```
# Node number(s) and/or 'start|stop|step' notation may be used for multiple
# nodes, ";" may be used to separate ranges. NOTE: Do not end a line with a ";"
# FXTR_NODE_X = FiX TRanslation of NODE in the X direction
FXTR_NODE_X: 1|21               #  Node 1 to 21, step size 1 is implied.
FXTR_NODE_Y: 1281               #  Lower right corner for 60x20 problem
FXTR_NODE_Z: 1; 4|13|3; 18|22|2 #  Nodes 1, 4, 7, 10, 13, 18, 20, 22.

# =============
# === Loads ===
# =============
# Node number(s) and/or 'start|stop|step' notation may be used for multiple
# nodes, ";" may be used to separate ranges. NOTE: Do not end a line with a ";"
# A load value (VALU) *must* be specified for the nodes you choose. Use + or -
# to set the direction of the load (+ is 'up', - is 'down).
# Set the *node number(s)* that's loaded, *not* the degrees of freedom, that's
# taken care of by ToPy. Assign values for the corresponding load size(s).
# Also note the use of the "@" below, which is rather convenient for the user.
LOAD_NODE_X: 1; 4; 9 #  Load nodes 1, 4 and 9.
LOAD_NODE_Y: 1       #  Upper left corner -- node number = 1 (always).
LOAD_NODE_Z: 20|32|3 #  Load node 20 to 32 in steps of 3.
LOAD_VALU_X: 0.75    #  Simply omit a line if not necessary for your problem.
LOAD_VALU_Y: -1      #  Value of the load = 1, direction negative Y (down).
LOAD_VALU_Z: 1@10    #  Value of the load = 1 at 10 nodes in Z direction.


# ================================================
# === Passive (void) and active (solid) elements ===
# ================================================
# List the *element* numbers you want to affect.
PASV_ELEM: 10|19; 30|39; 50|59; 70|79; 90|99
ACTV_ELEM: 1|1181|20; 1181|1200


# ============================================
# === Mechanism design (synthesis) specific ===
# ============================================
LOAD_NODE_X_OUT: 841 #  Node number(s) at which you require the output
LOAD_VALU_X_OUT: -1  #  Value of output at specified OUT node.


# ================================
# === Heat conduction specific ===
# ================================
# NOTE: For heat conduction problems, only use *_X keywords, i.e., no Y or Z
# dimensions, since heat problems are one-dimensional i.t.o. degrees of freedom
# (temperature is a scalar value).
# However, you still have to specify the following for 2D problems:
NUM_ELEM_Z : 0
# And don't forget this, for example:
DOF_PN     : 1
ELEM_K     : Q4T

# That's it, easy peasy :-)
```

A.4 Program structure

The program structure consists of four modules (a module is simply a Python file) and is listed below in order of complexity. The code is well commented and fairly easy to follow.

One needs some knowledge of Python to *fully* understand the code, however, even without *any* Python knowledge, one will be able to follow the code. There is a plethora of Python tutorials and even books available on the WWW, as well as published books, *e.g.*, Hetland (2005) is highly recommended.

In Python, a directory containing code is called a package. The ToPy directory, which contains four modules (Python files), may therefore be called the ToPy package.

ToPy consists of about a 1414 lines of code (including comments). A 1000–line program is generally considered as a small program.

A.4.1 Topology module (topology.py) description and listing

Of all the modules, this is the only one that contains a *class*. A class is handy, it enables us to create an object that "knows stuff" and can "do stuff". This module does all the numerical work, *i.e.*, optimisation and in particular, the FEA.

```
"""
# ============================================================================
# A class to optimise the topology of a design domain for defined boundary
# conditions. Data is read from an input file, see 'examples' directory.
#
# Author: William Hunter <willemjagter@gmail.com>
# Date: 01-06-2007
# Last change: 15-12-2008 (spell check)
# Copyright (C) 2008, William Hunter.
# ============================================================================
"""

from __future__ import division

from string import lower

from numpy import arange, array, dot, floor, indices, maximum, minimum, ones,\
put, setdiff1d, setmember1d, sqrt, where, zeros, zeros_like, append, empty,\
abs, take, log, allclose, clip, log2, equal

from pysparse import superlu, itsolvers, precon

from parser import tpd_file2dict

__all__ = ['Topology']


MAX_ITERS = 250

SOLID, VOID = 1.000, 0.001 #  Upper and lower bound value for design variables
KDATUM = 0.1 #  Reference stiffness value of springs for mechanism synthesis
```

```
# Constants for exponential approximation:
A_LOW = -3 #  Lower restriction on 'a' for exponential approximation
A_UPP = -1e-5 #  Upper restriction on 'a' for exponential approximation



# =======================
# === ToPy error class ===
# =======================
class ToPyError(Exception):
    """
    Base class for exceptions in this module.

    """
    pass #  Use default __init__ of Exception

# =======================
# === ToPy base class ===
# =======================
class Topology:
    """
    A class to optimise the topology of a design domain for defined boundary
    values. Data is read from an input file (see 'examples' folder).

    """
    def __init__(self):
        self.topydict = {} #  Store tpd file data in dictionary
        self.pcount = 0 #  Counter for continuation of p
        self.qcount = 0 #  Counter for continuation of q for GSF
        self.itercount = 0 #  Internal counter
        self.change = 1
        self.svtfrac = None

    # ======================
    # === Public methods ===
    # ======================
    def load_tpd_file(self, fname):
        """
        Load a ToPy problem definition (TPD) file, return a dictionary:

        INPUTS:
            fname -- file name of tpd file.

        EXAMPLES:
            >>> import topy
            >>> t = topy.Topology()
            >>> # load a ToPy problem definition file:
            >>> t.load_tpd_file('filename.tpd')

        """
        self.tpdfname = fname
        self.topydict = tpd_file2dict(fname)

    def set_top_params(self):
        """
        Set topology optimisation problem parameters (you must already have
        instantiated a Topology object).
```

```
        EXAMPLES:
            >>> import topy
            >>> t = topy.Topology()
            >>> # load a ToPy problem definition file:
            >>> t.load_tpd_file('filename.tpd')
            >>> # now set the problem parameters:
            >>> t.set_top_params()

        You can now access all the class methods or instance variables. To see
        the dictionary of parameters, just type

            >>> t.topydict

        You can also reset some parameters without reloading the file, but use
        with caution. For example, you can change the filer radius (FILT_RAD)
        as follows:

            >>> t.topydict['FILT_RAD'] = 1.5

        Remember to reset the problem parameters again, whereafter you can
        solve the problem.

            >>> t.set_top_params()

        See also: load_tpd_file

        """
        print '-' * 79
        # Set all the mandatory minimum amount of parameters that constitutes
        # a completely defined topology optimisation problem:
        if not self.topydict:
            raise ToPyError('You must first load a TPD file!')
        self.probtype = self.topydict['PROB_TYPE'] #  Problem type
        self.probname = self.topydict['PROB_NAME'] #  Problem name
        self.volfrac = self.topydict['VOL_FRAC'] #  Volume fraction
        self.filtrad = self.topydict['FILT_RAD'] #  Filter radius
        self.p = self.topydict['P_FAC'] #  'Standard' penalisation factor
        self.dofpn = self.topydict['DOF_PN'] #  DOF per node
        self.e2sdofmapi = self.topydict['E2SDOFMAPI'] #  Elem to structdof map
        self.nelx = self.topydict['NUM_ELEM_X'] #  Number of elements in X
        self.nely = self.topydict['NUM_ELEM_Y'] #  Number of elements in Y
        self.nelz = self.topydict['NUM_ELEM_Z'] #  Number of elements in Z
        self.fixdof = self.topydict['FIX_DOF'] #  Fixed dof vector
        self.loaddof = self.topydict['LOAD_DOF'] #  Loaded dof vector
        self.loadval = self.topydict['LOAD_VAL'] #  Loaded dof values
        self.Ke = self.topydict['ELEM_K'] #  Element stiffness matrix
        self.K = self.topydict['K'] #  Global stiffness matrix
        if self.nelz:
            print 'Domain discretisation (NUM_ELEM_X x NUM_ELEM_Y x \
NUM_ELEM_Z) = %d x %d x %d' % (self.nelx, self.nely, self.nelz)
        else:
            print 'Domain discretisation (NUM_ELEM_X x NUM_ELEM_Y) = %d x %d'\
            % (self.nelx, self.nely)

        print 'Element type (ELEM_K) =', self.topydict['ELEM_TYPE']
        print 'Filter radius (FILT_RAD) =', self.filtrad

        # Check for either one of the following two, will take NUM_ITER if both
        # are specified.
```

```
try:
    self.numiter = self.topydict['NUM_ITER'] #  Number of iterations
    print 'Number of iterations (NUM_ITER) = %d' % (self.numiter)
except KeyError:
    self.chgstop = self.topydict['CHG_STOP'] #  Change stop criteria
    print 'Change stop value (CHG_STOP) = %.3e (%.2f%%)' \
    % (self.chgstop, self.chgstop * 100)
    self.numiter = MAX_ITERS

# All DOF vector and design variables arrays:
# This needs to be recoded at some point, perhaps. I originally
# (incorrectly) thought I could do everything just by looking at DOF
# per node, not so, Cartesian dimension also plays a role.
# Thus, the 'if'* below is a hack for this to work, and it does...
if self.dofpn == 1:
    if self.nelz == 0: #  *had to this
        self.e2sdofmapi = self.e2sdofmapi[0:4]
        self.alldof = arange(self.dofpn * (self.nelx + 1) * \
        (self.nely + 1))
        self.desvars = zeros((self.nely, self.nelx)) + self.volfrac
    else:
        self.alldof = arange(self.dofpn * (self.nelx + 1) * \
        (self.nely + 1) * (self.nelz + 1))
        self.desvars = zeros((self.nelz, self.nely, self.nelx)) + \
        self.volfrac
elif self.dofpn == 2:
    self.alldof = arange(self.dofpn * (self.nelx + 1) * (self.nely +
    1))
    self.desvars = zeros((self.nely, self.nelx)) + self.volfrac
else:
    self.alldof = arange(self.dofpn * (self.nelx + 1) *\
    (self.nely + 1) * (self.nelz + 1))
    self.desvars = zeros((self.nelz, self.nely, self.nelx)) + \
    self.volfrac
self.df = zeros_like(self.desvars) #  Derivatives of obj. func. (array)
self.freedof = setdiff1d(self.alldof, self.fixdof) #  Free DOF vector
self.r = zeros_like(self.alldof).astype(float) #  Load vector
self.r[self.loaddof] = self.loadval #  Assign load values at loaded dof
self.rfree = self.r[self.freedof] #  Modified load vector (free dof)
self.d = zeros_like(self.r) #  Displacement vector
self.dfree = zeros_like(self.rfree) #  Modified load vector (free dof)
# Determine which rows and columns must be deleted from global K:
self._rcfixed = where(setmember1d(self.alldof, self.fixdof), 0, 1)

# Print this to screen, just so that the user knows what type of
# problem is being solved:
print 'Problem type (PROB_TYPE) = ' + self.probtype
print 'Problem name (PROB_NAME) = ' + self.probname

# Set extra parameters if specified:
# (1) Continuation parameters for 'p':
try:
    self._pmax = self.topydict['P_MAX']
    self._phold = self.topydict['P_HOLD']
    self._pincr = self.topydict['P_INCR']
    self._pcon = self.topydict['P_CON']
except KeyError: #  If they're None
    print 'Continuation of penalisation factor (P_FAC) not specified'
    self._pmax = 1
```

```
        self._pcon = self.numiter #  'p' stays constant for all iterations
        self._phold = self.numiter
    # (2) Extra penalisation factor (q) and continuation parameters:
    try:
        self.q = self.topydict['Q_FAC']
        print 'GSF active'
    except KeyError:
        print 'GSF not active'
        self.q = 1
    try:
        self._qmax = self.topydict['Q_MAX']
        self._qhold = self.topydict['Q_HOLD']
        self._qincr = self.topydict['Q_INCR']
        self._qcon = self.topydict['Q_CON']
    except KeyError:  #  If they're None
        self._qmax = self.q
        self._qcon = self.numiter #  'q' stays constant for all iterations
        self._qhold = self.numiter
    # (3) Exponential approximation of eta:
    try:
        self.eta = float(self.topydict['ETA']) * ones(self.desvars.shape)
        print 'Damping factor (ETA) = %3.2f' % (self.eta.mean())
    except ValueError:
        if self.topydict['ETA'] == 'exp':
            #  Initial value of exponent for comp and heat problems:
            self.a = - ones(self.desvars.shape)
            if self.probtype == 'mech':
                #  Initial value of exponent for mech problems:
                self.a = self.a * 7 / 3
            self.eta = 1 / (1 - self.a)
            print 'Damping factor (ETA) = exp'
    # (4) Diagonal quadratic approximation:
    try:
        self.approx = lower(self.topydict['APPROX'])
    except KeyError:
        self.approx = None
    if self.approx == 'dquad':
        print 'Using diagonal quadratic approximation (APPROX = dquad)'
    # (5) Set passive elements:
    self.pasv = self.topydict['PASV_ELEM']
    if self.pasv.any():
        print 'Passive elements (PASV_ELEM) specified'
    else:
        print 'No passive elements (PASV_ELEM) specified'
    # (6) Set active elements:
    self.actv = self.topydict['ACTV_ELEM']
    if self.actv.any():
        print 'Active elements (ACTV_ELEM) specified'
    else:
        print 'No active elements (ACTV_ELEM) specified'

    # Set parameters for compliant mechanism synthesis, if they exist:
    if self.probtype == 'mech':
        if self.topydict['LOAD_DOF_OUT'].any() and \
        self.topydict['LOAD_VAL_OUT'].any():
            self.loaddofout = self.topydict['LOAD_DOF_OUT']
            self.loadvalout = self.topydict['LOAD_VAL_OUT']
        else:
            raise ToPyError('Not enough input data for mechanism \
```

```
synthesis!')

            self.rout = zeros_like(self.alldof).astype(float)
            self.rout[self.loaddofout] = self.loadvalout
            self.rfreeout = self.rout[self.freedof]
            self.dout = zeros_like(self.rout)
            self.dfreeout = zeros_like(self.rfreeout)
            ksin = ones(self.loaddof.shape, dtype='int') * KDATUM
            ksout = ones(self.loaddofout.shape, dtype='int') * KDATUM
            maskin = ones(self.loaddof.shape, dtype='int')
            maskout = ones(self.loaddofout.shape, dtype='int')
            if len(ksin) > 1:
                self.K.update_add_mask_sym([ksin, ksin], self.loaddof, maskin)
                self.K.update_add_mask_sym([ksout, ksout], self.loaddofout, \
                maskout)
            else:
                self.K.update_add_mask_sym([ksin], self.loaddof, maskin)
                self.K.update_add_mask_sym([ksout], self.loaddofout, maskout)
        print '=' * 79

    def fea(self):
        """
        Performs a Finite Element Analysis given the updated global stiffness
        matrix [K] and the load vector {r}, both of which must be in the
        modified state, i.e., [K] and {r} must represent the unconstrained
        system of equations. Return the global displacement vector {d} as a
        NumPy array.

        EXAMPLES:
            >>> t.fea()

        See also: set_top_params

        """
        if not self.topydict:
            raise ToPyError('You must first load a TPD file!')
        if self.itercount >= MAX_ITERS:
            raise ToPyError('Maximum internal number of iterations exceeded!')

        Kfree = self._updateK(self.K.copy())

        if self.dofpn < 3 and self.nelz == 0: #  Direct solver
            Kfree = Kfree.to_csr() #  Need CSR for SuperLU factorisation
            lu = superlu.factorize(Kfree)
            lu.solve(self.rfree, self.dfree)
            if self.probtype == 'mech':
                lu.solve(self.rfreeout, self.dfreeout)  # mechanism synthesis
        else: #  Iterative solver for 3D problems
            Kfree = Kfree.to_sss()
            preK = precon.ssor(Kfree) #  Preconditioned Kfree
            (info, numitr, relerr) = \
            itsolvers.pcg(Kfree, self.rfree, self.dfree, 1e-8, 8000, preK)
            if info < 0:
                print 'PySparse error: Type:', info,', at', numitr, \
'iterations.'
                raise ToPyError('Solution for FEA did not converge.')
            else:
                print 'ToPy: Solution for FEA converged after', numitr, \
'iterations.'
```

```
            if self.probtype == 'mech':  # mechanism synthesis
                (info, numitr, relerr) = \
                itsolvers.pcg(Kfree, self.rfreeout, self.dfreeout, 1e-8, \
                8000, preK)
                if info < 0:
                    print 'PySparse error: Type:', info,', at', numitr, \
'iterations.'
                    raise ToPyError('Solution for FEA of adjoint load case \
                    did not converge.')

        # Update displacement vectors:
        self.d[self.freedof] = self.dfree
        if self.probtype == 'mech':  # 'adjoint' vectors
            self.dout[self.freedof] = self.dfreeout
        # Increment internal iteration counter
        self.itercount += 1

    def sens_analysis(self):
        """
        Determine the objective function value and perform sensitivity analysis
        (find the derivatives of objective function). Return the design
        sensitivities as a NumPy array.

        EXAMPLES:
            >>> t.sens_analysis()

        See also: fea

        """
        if not self.topydict:
            raise ToPyError('You must first load a TPD file!')
        tmp = self.df.copy()
        self.objfval  = 0.0 #  Objective function value
        if self.nelz == 0: #  2D problem
            for ely in xrange(self.nely):
                for elx in xrange(self.nelx):
                    e2sdofmap = self.e2sdofmapi + self.dofpn *\
                                (ely + elx * (self.nely + 1))
                    qe = self.d[e2sdofmap]
                    qeTKeqe = dot(dot(qe, self.Ke), qe)
                    if self.probtype == 'comp':
                        self.objfval += (self.desvars[ely, elx] ** self.p) *\
                        qeTKeqe
                        tmp[ely, elx] = - self.p * self.desvars[ely, elx] **\
                        (self.p - 1) * qeTKeqe
                    elif self.probtype == 'heat':
                        self.objfval += (VOID + (1 - VOID) * \
                        self.desvars[ely, elx] ** self.p) * qeTKeqe
                        tmp[ely, elx] = - (1 - VOID) * self.p * \
                        self.desvars[ely, elx] ** (self.p - 1) * qeTKeqe
                    elif self.probtype == 'mech':
                        self.objfval = self.d[self.loaddofout]
                        qeout = self.dout[e2sdofmap]
                        tmp[ely, elx] = self.p * self.desvars[ely, elx]\
                        ** (self.p - 1) * dot(dot(qe, self.Ke), qeout)
        else: #  3D problem
            for elz in xrange(self.nelz):
                for ely in xrange(self.nely):
                    for elx in xrange(self.nelx):
```

```
                        e2sdofmap = self.e2sdofmapi + self.dofpn *\
                                    (ely + elx * (self.nely + 1) + elz *\
                                    (self.nelx + 1) * (self.nely + 1))
                        qe = self.d[e2sdofmap]
                        qeTKeqe = dot(dot(qe, self.Ke), qe)
                        if self.probtype == 'comp':
                            self.objfval += (self.desvars[elz, ely, elx] **\
                            self.p) * qeTKeqe
                            tmp[elz, ely, elx] = - self.p * self.desvars[elz, \
                            ely, elx] ** (self.p - 1) * qeTKeqe
                        elif self.probtype == 'heat':
                            self.objfval += (VOID + (1 - VOID) * \
                            self.desvars[elz, ely, elx] ** self.p) * qeTKeqe
                            tmp[elz, ely, elx] = - (1 - VOID) *  self.p * \
                            self.desvars[elz, ely, elx] ** (self.p - 1) * \
                            qeTKeqe
                        elif self.probtype == 'mech':
                            self.objfval = self.d[self.loaddofout].sum()
                            qeout = self.dout[e2sdofmap]
                            tmp[elz, ely, elx] = self.p * \
                            self.desvars[elz, ely, elx] ** (self.p - 1) * \
                            dot(dot(qe, self.Ke), qeout)
        self.df = tmp

    def filter_sens_sigmund(self):
        """
        Filter the design sensitivities using Sigmund's heuristic approach.
        Return the filtered sensitivities.

        EXAMPLES:
            >>> t.filter_sens_sigmund()

        See also: sens_analysis

        """
        if not self.topydict:
            raise ToPyError('You must first load a TPD file!')
        tmp = zeros_like(self.df)
        rmin = int(floor(self.filtrad))
        if self.nelz == 0:
            U, V = indices((self.nelx, self.nely))
            for i in xrange(self.nelx):
                umin = maximum(i - rmin - 1, 0)
                umax = minimum(i + rmin + 2, self.nelx + 1)
                for j in xrange(self.nely):
                    vmin = maximum(j - rmin - 1, 0)
                    vmax = minimum(j + rmin + 2, self.nely + 1)
                    u = U[umin: umax, vmin: vmax]
                    v = V[umin: umax, vmin: vmax]
                    dist = self.filtrad - sqrt((i - u) ** 2 + (j - v) ** 2)
                    sumnumr = (maximum(0, dist) * self.desvars[v, u] *\
                               self.df[v, u]).sum()
                    sumconv = maximum(0, dist).sum()
                    tmp[j, i] = sumnumr / (sumconv * self.desvars[j, i])
        else:
            rmin3 = rmin
            U, V, W = indices((self.nelx, self.nely, self.nelz))
            for i in xrange(self.nelx):
                umin, umax = maximum(i - rmin - 1, 0),\
```

```
                            minimum(i + rmin + 2, self.nelx + 1)
                for j in xrange(self.nely):
                    vmin, vmax = maximum(j - rmin - 1, 0),\
                                 minimum(j + rmin + 2, self.nely + 1)
                    for k in xrange(self.nelz):
                        wmin, wmax = maximum(k - rmin3 - 1, 0),\
                                     minimum(k + rmin3 + 2, self.nelz + 1)
                        u = U[umin:umax, vmin:vmax, wmin:wmax]
                        v = V[umin:umax, vmin:vmax, wmin:wmax]
                        w = W[umin:umax, vmin:vmax, wmin:wmax]
                        dist = self.filtrad - sqrt((i - u) ** 2 + (j - v) **\
                               2 + (k - w) ** 2)
                        sumnumr = (maximum(0, dist) * self.desvars[w, v, u] *\
                                   self.df[w, v, u]).sum()
                        sumconv = maximum(0, dist).sum()
                        tmp[k, j, i] = sumnumr/(sumconv *\
                        self.desvars[k, j, i])

        self.df = tmp


    def update_desvars_oc(self):
        """
        Update the design variables by means of OC-like method, using the
        filtered sensitivities; return the updated design variables.

        EXAMPLES:
            >>> t.update_desvars_oc()

        See also: sens_analysis, filter_sens_sigmund

        """
        if not self.topydict:
            raise ToPyError('You must first load a TPD file!')
        # 'p' stays constant for a specified number of iterations from start.
        # 'p' is incremented, but not more than the maximum allowable value.
        # If continuation parameters are not specified in the input file, 'p'
        # will stay constant:
        if self.pcount >= self._phold:
            if (self.p + self._pincr) < self._pmax + self._pincr:
                if (self.pcount - self._phold) % self._pcon == 0:
                    self.p += self._pincr

        if self.qcount >= self._qhold:
            if (self.q + self._qincr) < self._qmax:
                if (self.qcount - self._qhold) % self._qcon == 0:
                    self.q += self._qincr

        self.pcount += 1
        self.qcount += 1

        # Exponential approximation of eta (damping factor):
        if self.itercount > 1:
            if self.topydict['ETA'] == 'exp': #  Check TPD specified value
                mask = equal(self.desvarsold / self.desvars, 1)
                self.a = 1 + log2(abs(self.dfold / self.df)) / \
                log2(self.desvarsold / self.desvars + mask) + \
                mask * (self.a - 1)
                self.a = clip(self.a, A_LOW, A_UPP)
```

```
            self.eta = 1 / (1 - self.a)

        self.dfold = self.df.copy()
        self.desvarsold = self.desvars.copy()

        # Change move limit for compliant mechanism synthesis:
        if self.probtype == 'mech':
            move = 0.1
        else:
            move = 0.2
        lam1, lam2 = 0, 100e3
        dims = self.desvars.shape
        while (lam2 - lam1) / (lam2 + lam1) > 1e-8 and lam2 > 1e-40:
            lammid = 0.5 * (lam1 + lam2)
            if self.probtype == 'mech':
                if self.approx == 'dquad':
                    curv = - 1 / (self.eta * self.desvars) * self.df
                    beta = maximum(self.desvars-(self.df + lammid)/curv, VOID)
                    move_upper = minimum(move, self.desvars / 3)
                    desvars = maximum(VOID, maximum((self.desvars - move),\
                    minimum(SOLID,  minimum((self.desvars + move), \
                    (self.desvars * maximum(1e-10, \
                    (-self.df / lammid))**self.eta)**self.q))))
                else:  # reciprocal or exponential
                    desvars = maximum(VOID, maximum((self.desvars - move),\
                    minimum(SOLID,  minimum((self.desvars + move), \
                    (self.desvars * maximum(1e-10, \
                    (-self.df / lammid))**self.eta)**self.q))))
            else:  # compliance or heat
                if self.approx == 'dquad':
                    curv = - 1 / (self.eta * self.desvars) * self.df
                    beta = maximum(self.desvars-(self.df+lammid)/curv, VOID)
                    move_upper = minimum(move, self.desvars / 3)
                    desvars = maximum(VOID, maximum((self.desvars - move),\
                    minimum(SOLID,  minimum((self.desvars + move_upper), \
                    beta**self.q))))
                else:  # reciprocal or exponential
                    desvars = maximum(VOID, maximum((self.desvars - move),\
                    minimum(SOLID,  minimum((self.desvars + move), \
                    (self.desvars * (-self.df / lammid)**self.eta)**self.q))))

            # Check for passive and active elements, modify updated x:
            if self.pasv.any() or self.actv.any():
                flatx = desvars.flatten()
                idx = []
                if self.nelz == 0:
                    y, x = dims
                    for j in range(x):
                        for k in range(y):
                            idx.append(k*x + j)
                else:
                    z, y, x = dims
                    for i in range(z):
                        for j in range(x):
                            for k in range(y):
                                idx.append(k*x + j + i*x*y)
                if self.pasv.any():
                    pasv = take(idx, self.pasv) #  new indices
                    put(flatx, pasv, VOID) #  = zero density
```

```
                if self.actv.any():
                    actv = take(idx, self.actv) #  new indices
                    put(flatx, actv, SOLID) #  = solid
                desvars = flatx.reshape(dims)

            if self.nelz == 0:
                if desvars.sum() - self.nelx * self.nely * self.volfrac > 0:
                    lam1 = lammid
                else:
                    lam2 = lammid
            else:
                if desvars.sum() - self.nelx * self.nely * self.nelz *\
                self.volfrac > 0:
                    lam1 = lammid
                else:
                    lam2 = lammid
        self.lam = lammid

        self.desvars = desvars

        # Change in design variables:
        self.change = (abs(self.desvars - self.desvarsold)).max()

        # Solid-void fraction:
        nr_s = self.desvars.flatten().tolist().count(SOLID)
        nr_v = self.desvars.flatten().tolist().count(VOID)
        self.svtfrac = (nr_s + nr_v) / self.desvars.size


    # ===================================
    # === Private methods and helpers ===
    # ===================================
    def _updateK(self, K):
        """
        Update the global stiffness matrix by looking at each element's
        contribution i.t.o. design domain density and the penalisation factor.
        Return unconstrained stiffness matrix.

        """
        if self.nelz == 0: #  2D problem
            for elx in xrange(self.nelx):
                for ely in xrange(self.nely):
                    e2sdofmap = self.e2sdofmapi + self.dofpn *\
                    (ely + elx * (self.nely + 1))
                    if self.probtype == 'comp' or self.probtype == 'mech':
                        updatedKe = self.desvars[ely, elx] ** self.p * self.Ke
                    elif self.probtype == 'heat':
                        updatedKe = (VOID + (1 - VOID) * \
                        self.desvars[ely, elx] ** self.p) * self.Ke
                    mask = ones(e2sdofmap.size, dtype=int)
                    K.update_add_mask_sym(updatedKe, e2sdofmap, mask)
        else: #  3D problem
            for elz in xrange(self.nelz):
                for elx in xrange(self.nelx):
                    for ely in xrange(self.nely):
                        e2sdofmap = self.e2sdofmapi + self.dofpn *\
                                    (ely + elx * (self.nely + 1) + elz *\
                                    (self.nelx + 1) * (self.nely + 1))
                        if self.probtype == 'comp' or self.probtype == 'mech':
```

```
                            updatedKe = self.desvars[elz, ely, elx] ** \
                            self.p * self.Ke
                        elif self.probtype == 'heat':
                            updatedKe = (VOID + (1 - VOID) * \
                            self.desvars[elz, ely, elx] ** self.p) * self.Ke
                        mask = ones(e2sdofmap.size, dtype=int)
                        K.update_add_mask_sym(updatedKe, e2sdofmap, mask)

        K.delete_rowcols(self._rcfixed) #  Del constrained rows and columns
        return K

# EOF topology.py
```

A.4.2 Parser module (parser.py) description and listing

This module takes reads the TPD file and parses the data to a Python dictionary, which is very easy and convenient to use.

```
"""
# ============================================================================
# Parse a ToPy problem definition (TPD) file to a Python dictionary.
#
# Author: William Hunter <willemjagter@gmail.com>
# Date: 01-06-2007
# Last change: 15-12-2008 (spell check)
# Copyright (C) 2008, William Hunter.
# ============================================================================
"""

from string import lower

from numpy import append, arange, array, ones, r_

from pysparse import spmatrix

from elements import *


# ===================================================
# === Messages used for errors, information, etc. ===
# ===================================================
MSG0 = 'An input error occurred, please try again.\nPerhaps your filename \
and/or path is incorrect?'

MSG1 = 'Input file or format not recognised.\nDid you specify the file \
version header? \nPerhaps it\'s just a typing error?'

MSG2 = 'One or more parameters incorrectly specified or not enough\n\
parameters specified. Please check your input file for errors.'

MSG3 = 'ToPy problem definition (TPD) file successfully parsed.'

MSG4 = "<TABs> are not allowed in the input file! Please \ncheck the entries \
listed at the top of this output and correct your TPD file."

MSG5 = 'Node numbers (and \'step\' values) can only be integers.\nPlease \
check your input file.'

MSG6 = 'Load vector and load value vector lengths not equal.'

MSG7 = 'No load(s) or no loaded node(s) specified.'


# ========================
# === ToPy Error class ===
# ========================
class ToPyError(Exception):
    """
    Base class for exceptions in this module.

    """
```

```
    pass #  Use default __init__ of Exception


# ========================
# === Public functions ===
# ========================
def tpd_file2dict(fname):
    """
    Read in *all* the parameters from a TPD file and return a dictionary.

    The file type must be a valid ToPy problem definition file, see the
    <examples/scripts> folder for an explanation of how the input must look
    (refer to 'template.tpd'). This function checks for the file version, and
    decides which parser to call.

    INPUTS:
        fname -- file name of tpd file.

    OUTPUTS:
        A dictionary.

    ADDITIONAL INPUTS (arguments and/or keyword arguments):
        None.

    EXAMPLES:
        >>> tpd_file2dict('2d_beam.tpd')

    """
    try:
        f = open(fname)
    except IOError:
        raise ToPyError(MSG0)
    s = f.read()
    # Check for file version header, and parse:
    if s.startswith('[ToPy Problem Definition File v2007]') != True:
        raise ToPyError(MSG1)
    elif s.startswith('[ToPy Problem Definition File v2007]') == True:
        d = _parsev2007file(s)
        print '\n' + '=' * 79
        print MSG3
        print "TPD file name:", fname, "(v2007)"
    # Very basic parameter checking, exit on error:
    _checkparams(d)
    # Future file versions, enter <if> and <elif> as per above and define new
    # _parsev20??file()
    # See method below...
    return d


# ====================================
# === Private functions and helpers ===
# ====================================
def _parsev2007file(s):
    """
    Parse a version 2007 ToPy problem definition file to a dictionary.

    """
    d = {} #  Empty dictionary that we're going to fill
    snew = []
```

```
    s = s.splitlines()
    for line in range(1, len(s)):
        if s[line] and s[line][0] != '#':
            if s[line].count('#'):
                snew.append(s[line].rsplit('#')[0:-1][0])
            else:
                snew.append(s[line])
    # Check for <TAB>s; if found print lines and exit:
    _checkfortabs(snew)
    # Create dictionary containing all lines of input file:
    for i in snew:
        pair = i.split(':')
        d[pair[0].strip()] = pair[1].strip()

    # Read/convert minimum required input and convert, else exit:
    try:
        d['PROB_TYPE'] = lower(d['PROB_TYPE'])
        d['VOL_FRAC'] = float(d['VOL_FRAC'])
        d['FILT_RAD'] = float(d['FILT_RAD'])
        d['P_FAC'] = float(d['P_FAC'])
        d['NUM_ELEM_X'] = int(d['NUM_ELEM_X'])
        d['NUM_ELEM_Y'] = int(d['NUM_ELEM_Y'])
        d['NUM_ELEM_Z'] = int(d['NUM_ELEM_Z'])
        d['DOF_PN'] = int(d['DOF_PN'])
        d['ELEM_TYPE'] = d['ELEM_K']
        d['ELEM_K'] = eval(d['ELEM_TYPE'])
        try:
            d['ETA'] = float(d['ETA'])
        except ValueError:
            d['ETA'] = lower(d['ETA'])
    except:
        raise ToPyError(MSG2)

    # Check for number of iterations or change stop value:
    try:
        d['NUM_ITER'] = int(d['NUM_ITER'])
    except KeyError:
        try:
            d['CHG_STOP'] = float(d['CHG_STOP'])
        except KeyError:
            raise ToPyError(MSG2)
    except KeyError:
        raise ToPyError(MSG2)

    # Check for GSF penalty factor:
    try:
        d['Q_FAC'] = float(d['Q_FAC'])
    except KeyError:
        pass

    # Check for continuation parameters:
    try:
        d['P_MAX'] = float(d['P_MAX'])
        d['P_HOLD'] = int(d['P_HOLD'])
        d['P_INCR'] = float(d['P_INCR'])
        d['P_CON'] = float(d['P_CON'])
    except KeyError:
        pass
```

```
    try:
        d['Q_MAX'] = float(d['Q_MAX'])
        d['Q_HOLD'] = int(d['Q_HOLD'])
        d['Q_INCR'] = float(d['Q_INCR'])
        d['Q_CON'] = float(d['Q_CON'])
    except KeyError:
        pass

    # Check for active elements:
    try:
        d['ACTV_ELEM'] = _tpd2vec(d['ACTV_ELEM']) - 1
    except KeyError:
        d['ACTV_ELEM'] = _tpd2vec('')

    # Check for passive elements:
    try:
        d['PASV_ELEM'] = _tpd2vec(d['PASV_ELEM']) - 1
    except KeyError:
        d['PASV_ELEM'] = _tpd2vec('')

    # Check if diagonal quadratic approximation is required:
    try:
        d['APPROX'] = lower(d['APPROX'])
    except KeyError:
        pass

    # How to do the following compactly (perhaps loop through keys)? Check for
    # keys and create fixed DOF vector, loaded DOF vector and load values
    # vector.
    dofpn = d['DOF_PN']

    x = y = z = ''
    if d.has_key('FXTR_NODE_X'):
        x = d['FXTR_NODE_X']
    if d.has_key('FXTR_NODE_Y'):
        y = d['FXTR_NODE_Y']
    if d.has_key('FXTR_NODE_Z'):
        z = d['FXTR_NODE_Z']
    d['FIX_DOF'] = _dofvec(x, y, z, dofpn)

    x = y = z = ''
    if d.has_key('LOAD_NODE_X'):
        x = d['LOAD_NODE_X']
    if d.has_key('LOAD_NODE_Y'):
        y = d['LOAD_NODE_Y']
    if d.has_key('LOAD_NODE_Z'):
        z = d['LOAD_NODE_Z']
    d['LOAD_DOF'] = _dofvec(x, y, z, dofpn)

    x = y = z = ''
    if d.has_key('LOAD_VALU_X'):
        x = d['LOAD_VALU_X']
    if d.has_key('LOAD_VALU_Y'):
        y = d['LOAD_VALU_Y']
    if d.has_key('LOAD_VALU_Z'):
        z = d['LOAD_VALU_Z']
    d['LOAD_VAL'] = _valvec(x, y, z)

    # Compliant mechanism synthesis values and vectors:
```

```
    x = y = z = ''
    if d.has_key('LOAD_NODE_X_OUT'):
        x = d['LOAD_NODE_X_OUT']
    if d.has_key('LOAD_NODE_Y_OUT'):
        y = d['LOAD_NODE_Y_OUT']
    if d.has_key('LOAD_NODE_Z_OUT'):
        z = d['LOAD_NODE_Z_OUT']
    d['LOAD_DOF_OUT'] = _dofvec(x, y, z, dofpn)

    x = y = z = ''
    if d.has_key('LOAD_VALU_X_OUT'):
        x = d['LOAD_VALU_X_OUT']
    if d.has_key('LOAD_VALU_Y_OUT'):
        y = d['LOAD_VALU_Y_OUT']
    if d.has_key('LOAD_VALU_Z_OUT'):
        z = d['LOAD_VALU_Z_OUT']
    d['LOAD_VAL_OUT'] = _valvec(x, y, z)

    # The following entries are created and added to the dictionary,
    # they are not specified in the ToPy problem definition file:
    Ksize = d['DOF_PN'] * (d['NUM_ELEM_X'] + 1) * (d['NUM_ELEM_Y']\
    + 1) * (d['NUM_ELEM_Z'] + 1) #  Memory allocation hint for PySparse
    d['K'] = spmatrix.ll_mat_sym(Ksize, Ksize) #  Global stiffness matrix
    d['E2SDOFMAPI'] =  _e2sdofmapinit(d['NUM_ELEM_X'], d['NUM_ELEM_Y'], \
    d['DOF_PN']) #  Initial element to structure DOF mapping

    return d

def _tpd2vec(seq):
    """
    Convert a tpd file string to a vector, return a NumPy array.

    EXAMPLES:
        >>> _tpd2vec('1|13|4; 20; 25|28')
        array([  1.,   5.,   9.,  13.,  20.,  25.,  26.,  27.,  28.])
        >>> _tpd2vec('5.5; 1.2@3; 3|7|2')
        array([ 5.5,  1.2,  1.2,  1.2,  3. ,  5. ,  7. ])
        >>> _tpd2vec(' ')
        array([], dtype=float64)

    """
    finalvec = array([], int)
    for s in seq.split(';'):
        if s.count('|') >= 1:
            vec = s.split('|')
            try:
                a = int(vec[0])
                b = int(vec[1])
            except ValueError:
                raise ToPyError(MSG5)
            try:
                c = int(vec[2])
            except IndexError:
                c = 1
            except ValueError:
                raise ToPyError(MSG5)
            vec = arange(a, b + 1, c)
        elif s.count('@'):
            vec = s.split('@')
```

```
            try:
                vec = ones(int(vec[1])) * float(vec[0])
            except ValueError:
                raise ToPyError(MSG2)
        else:
            try:
                if s.count('.') == 1:
                    vec = [float(s)]
                else:
                    vec = [int(s)]
            except ValueError:
                vec = array([])
        finalvec = append(finalvec, vec)
    return finalvec

def _dofvec(x, y, z, dofpn):
    """
    DOF vector.

    """
    dofx = (_tpd2vec(x) - 1) * dofpn
    dofy = (_tpd2vec(y) - 1) * dofpn + 1
    if dofpn == 2:
        dofz = []
    else:
        dofz = (_tpd2vec(z) - 1) * dofpn + 2
    return r_[dofx, dofy, dofz].astype(int)

def _valvec(x, y, z):
    """
    Values (e.g., of loads) vector.

    """
    valx = _tpd2vec(x)
    valy = _tpd2vec(y)
    if z:
        valz = _tpd2vec(z)
    else:
        valz = []
    return r_[valx, valy, valz]

def _e2sdofmapinit(nelx, nely, dofpn):
    """
    Create the initial element to structure (e2s) DOF mapping (connectivity).
    Return a vector as a NumPy array.

    """
    if dofpn == 1:
        e2s = r_[1, (nely + 2), (nely + 1), 0]
        e2s = r_[e2s, (e2s + (nelx + 1) * (nely + 1))]
    elif dofpn == 2:
        b = arange(2 * (nely + 1), 2 * (nely + 1) + 2)
        a = b + 2
        e2s = r_[2, 3, a, b, 0, 1]
    elif dofpn == 3:
        d = arange(3)
        a = d + 3
        c = arange(3 * (nely + 1), 3 * (nely + 1) + 3)
        b = arange(3 * (nely + 2), 3 * (nely + 2) + 3)
```

```
        h = arange(3 * (nelx + 1) * (nely + 1), 3 * (nelx + 1) * (nely + 1) + 3)
        e = arange(3 * ((nelx+1) * (nely+1)+1), 3 * ((nelx+1) * (nely+1)+1) + 3)
        g = arange(3 * ((nelx + 1) * (nely + 1) + (nely + 1)),\
        3 * ((nelx + 1) * (nely + 1) + (nely + 1)) + 3)
        f = arange(3 * ((nelx + 1) * (nely + 1) + (nely + 2)),\
        3 * ((nelx + 1) * (nely + 1) + (nely + 2)) + 3)
        e2s = r_[a, b, c, d, e, f, g, h]
    return e2s

def _checkfortabs(s):
    """
    Check for tabs inside input file, return message telling where offending
    tabs are.

    """
    l = []
    for i in s:
        if i.count('\t'):
            l.append(i)
    if len(l) > 0:
        print '\n' + '='*80
        for line in l:
            print line + '\n'
        print '='*80
        raise ToPyError(MSG4)

def _checkparams(d):
    """
    Does a few *very basic* checks with regards to the ToPy input parameters.
    A message will be printed to screen *guessing* a possible problem in
    the input data, if found.

    """
    if d['LOAD_DOF'].size != d['LOAD_VAL'].size:
        raise ToPyError(MSG6)
    if d['LOAD_VAL'].size + d['LOAD_DOF'].size == 0:
        raise ToPyError(MSG7)
    # Check for rigid body motion and warn user:
    if d['DOF_PN'] == 2:
        if not d.has_key('FXTR_NODE_X') or not d.has_key('FXTR_NODE_Y'):
            print '\n\tToPy warning: Rigid body motion in 2D is possible!\n'
    if d['DOF_PN'] == 3:
        if not d.has_key('FXTR_NODE_X') or not d.has_key('FXTR_NODE_Y')\
        or not d.has_key('FXTR_NODE_Z'):
            print '\n\tToPy warning: Rigid body motion in 3D is possible!\n'

# EOF parser.py
```

A.4.3 Visualisation module (visualisation.py) description and listing

This module creates 2D or 3D graphical representations of NumPy arrays. For 2D we create PNG files (or any other 2D format supported by Matplotlib) and for 3D we make use of legacy ASCII VTK files.

```
"""
# ============================================================================
# Functions in order to visualise 2D and 3D NumPy arrays.
#
# Author: William Hunter <willemjagter@gmail.com>
# Date: 01-06-2007
# Last change: 15-12-2008 (spell check)
# Copyright (C) 2008, William Hunter.
# ============================================================================
"""

import sys

from datetime import datetime

from pylab import axis, cm, figure, imshow, savefig, title

from numpy import arange, asarray

from pyvtk import CellData, LookupTable, Scalars, UnstructuredGrid, VtkData

__all__ = ['create_2d_imag', 'create_3d_geom']

#  Lower bound value used for pixel/voxel culling, any value below this won't
# be plotted. Should be same as VOID's value in 'topolgy.py'.
THRESHOLD = 0.001

# TO DO: Add this function in order to plot constraints and loads, 2D only:
# pyplot.quiver([fromx],[fromy],[tox],[toy], color='b', edgecolors='k',
# linewidths=(1,))
#
# 3D is another story...

def create_2d_imag(x, **kwargs):
    """
    Create an image from a 2D NumPy array (using Matplotlib commands).

    Takes a 2D array as argument and saves it as an image. Each value in the
    array is represented by a square and the 'transparency' of each square
    is determined by the value of the array entry, which must vary between 0.0
    and 1.0. A 'dd-mm-yyyy-HHhMM' timestamp is automatically added to the
    filename unless the function is called with the time='none' keyword
    argument. Default image type is PNG.

    INPUTS:
        x -- M-by-N array (rows x columns)

    OUTPUTS:
        <filename>.png

    ADDITIONAL INPUTS (keyword arguments):
```

```
        prefix -- A user given prefix for the file name; default is 'topy_2d'.
        filetype -- The visualisation file type, see above.
        iternum -- A number that will be appended after the filename; default
                   is 'nin'.
        time -- If 'none', then NO timestamp will be added.
        title -- Plot title, useful for iteration info.

    EXAMPLES:
        >>> create_2d_imag(x, iternum=12, prefix='mbb_beam')
        >>> create_2d_imag(x)
        >>> create_2d_imag(x, prefix='test', filetype='pdf', time='none')

    """
    # ====================================
    # === Start of Matplotlib commands ===
    # ====================================
    # x = flipud(x) #  Check your matplotlibrc file; might plot upside-down...
    figure()
    if kwargs.has_key('title'):
        title(kwargs['title'])
        imshow(-x, cmap=cm.gray, aspect='equal', interpolation='nearest')
    imshow(-x, cmap=cm.gray, aspect='equal', interpolation='nearest')
    axis('off')
    axis('equal')
    # ==================================
    # === End of Matplotlib commands ===
    # ==================================
    # Set the filename component defaults:
    keys = ['dflt_prefix', 'dflt_iternum', 'dflt_timestamp', 'dflt_filetype']
    values = ['topy_2d', 'nin', '_' + _timestamp(), 'png']
    fname_dict = dict(zip(keys, values))
    # Change the default filename based on keyword arguments, if necessary:
    fname = _change_fname(fname_dict, kwargs)
    # Save the domain as image:
    savefig(fname)

def create_3d_geom(x, **kwargs):
    """
    Create 3D geometry from a 3D NumPy array.

    Takes a 3D array as argument and saves it as geometry. Each value in the
    array is represented by a 1x1x1 cube and the colour of each cube is
    determined by the value of the array entry, which must vary between 0.0 and
    1.0. Array entries with values below THRESHOLD are culled from the geometry
    (see source for details). A 'dd-mm-yyyy-HHhMM' timestamp is automatically
    added to the filename unless the function is called with the time='none'
    keyword argument. Default, and only file type, is legacy VTK unstructured
    grid file ('vtk' extension); it does not have to be specified.

    INPUTS:
        x -- K-by-M-by-N array (depth x rows x columns)

    OUTPUTS:
        <filename>.<type>

    ADDITIONAL INPUTS (keyword arguments):
        prefix -- A user given prefix for the file name; default is 'topy_3d'.
        filetype -- The visualisation file type, see above.
        iternum -- A number that will be appended after the filename; default
```

```
                is 'nin'.
        time -- If 'none', then NO timestamp will be added.

    EXAMPLES:
        >>> create_3d_geom(x, iternum=12, prefix='mbb_beam')
        >>> create_3d_geom(x)
        >>> create_3d_geom(x, time='none')

    """
    # Set the filename component defaults:
    keys = ['dflt_prefix', 'dflt_iternum', 'dflt_timestamp', 'dflt_filetype']
    values = ['topy_3d', 'nin', '_' + _timestamp(), 'vtk']
    fname_dict = dict(zip(keys, values))
    # Change the default filename based on keyword arguments, if necessary:
    fname = _change_fname(fname_dict, kwargs)
    # Save the domain as geometry:
    _write_geom(x, fname)


# ====================================
# === Private functions and helpers ===
# ====================================
def _change_fname(fd, kwargs):
    # Default file name:
    filename = fd['dflt_prefix'] + '_' + fd['dflt_iternum'] + \
    fd['dflt_timestamp'] + '.' + fd['dflt_filetype']
    if kwargs == {}:
        pass
    else:
        # This is not pretty but it works...
        if kwargs.has_key('prefix'):
            filename = filename.replace(fd['dflt_prefix'], kwargs['prefix'])
        if kwargs.has_key('iternum'):
            fixed_iternum = _fixiternum(str(kwargs['iternum']))
            filename = filename.replace(fd['dflt_iternum'], fixed_iternum)
        if kwargs.has_key('filetype'):
            filename = filename.replace(fd['dflt_filetype'],
            kwargs['filetype'])
        if kwargs.has_key('time'):
            filename = filename.replace(fd['dflt_timestamp'], '')
    return filename

def _write_geom(x, fname):
    if fname.endswith('vtk', -3):
        _write_legacy_vtu(x, fname)
    else:
        print 'Other file formats not implemented, only legacy VTK.'
        #_write_vrml2(x, fname)

def _write_legacy_vtu(x, fname):
    """
    Write a legacy VTK unstructured grid file.

    """
    # Voxel local points relative to its centre of geometry:
    voxel_local_points = asarray([[-1,-1,-1],[ 1,-1,-1],[-1, 1,-1],[ 1, 1,-1],
                                  [-1,-1, 1],[ 1,-1, 1],[-1, 1, 1],[ 1, 1, 1]])\
                                    * 0.5 # scaling
    # Voxel world points:
```

```
    points = []
    # Culled input array -- as list:
    xculled = []

    try:
        depth, rows, columns = x.shape
    except ValueError:
        sys.exit('Array dimensions not equal to 3, possibly 2-dimensional.\n')

    for i in xrange(depth):
        for j in xrange(rows):
            for k in xrange(columns):
                if x[i,j,k] > THRESHOLD:
                    xculled.append(x[i,j,k])
                    points += (voxel_local_points + [i,j,k]).tolist()

    voxels = arange(len(points)).reshape(len(xculled), 8).tolist()
    topology = UnstructuredGrid(points, voxel = voxels)
    file_header = \
    'ToPy data, created '\
    + str(datetime.now()).rsplit('.')[0]
    scalars = CellData(Scalars(xculled, name='Densities', lookup_table =\
    'default'))
    vtk = VtkData(topology, file_header, scalars)
    vtk.tofile(fname, 'binary')

def _timestamp():
    """
    Create and return a timestamp string.

    """
    now = datetime.now()
    day = _fixstring(str(now.day))
    month = _fixstring(str(now.month))
    year = str(now.year)
    hour = _fixstring(str(now.hour))
    minute = _fixstring(str(now.minute))
    ts = day + '-' + month + '-' + year + '-' + hour + 'h' + minute
    return ts

def _fixstring(s):
    """
    Fix the string by adding a zero in front if single digit number.

    """
    if len(s) == 1:
        s = '0' + s
    return s

def _fixiternum(s):
    """
    Fix the string by adding a zero in front if double digit number, and two
    zeros if single digit number.

    """
    if len(s) == 2:
        s = '0' + s
    elif len(s) == 1:
        s = '00' + s
```

```
    return s

# EOF visualisation.py
```

A.4.4 Element module (element.py) description and listing

The elements used in this thesis is coded as part of ToPy, however, the user can very easily add and use his own elements, see the 'data' directory of the software.

```
"""
# ============================================================================
# Finite element stiffness matrices.
#
# To define your own finite elements, see Python scripts in 'data' directory.
#
# Author: William Hunter <willemjagter@gmail.com>
# Date: 01-06-2007
# Last change: 15-12-2008 (spell check)
# Copyright (C) 2008, William Hunter.
# ============================================================================
"""

from __future__ import division

from os import path

from numpy import array, linspace, unique, sqrt, round, load
from numpy.linalg import eigvalsh

from topy.core.data.matlcons import _a, _nu, _E

__all__ = ['Q4', 'Q5B',  'Q4a5B',  'Q4T',\
           'H8', 'H18B', 'H8T']

# ====================================================
# === Messages used for errors, information, etc. ===
# ====================================================
MSG0 = 'finite element stiffness matrix.'

MSG1 = 'Element stiffness matrix does not exist... Created! \nPlease re-run \
your last attempt.'

# Set path to data folder:
pth = path.join(path.split(__file__)[0], 'data')

# 2D elements
# ############################################################################
# ============================================
# === KBar of Q4, see De Klerk and Groenwold ===
# ============================================
fname = path.join(pth, 'Q4bar.K')
try:
    Q4bar = load(fname)
except IOError:
    from topy.core.data import Q4bar_K
    raise IOError(MSG1)

# ============================================================================
# === Stiffness matrix of a square 4 node plane stress bi-linear element ===
# ============================================================================
fname = path.join(pth, 'Q4.K')
try:
```

```
    Q4 = load(fname)
except IOError:
    from topy.core.data import Q4_K
    raise IOError(MSG1)

# ===========================================================================
# === Stiffness matrix of a square 4 node plane stress '5-beta' element ===
# ===========================================================================
fname = path.join(pth, 'Q5B.K')
try:
    Q5B = load(fname)
except IOError:
    from topy.core.data import Q5B_K
    raise IOError(MSG1)

# ==========================================================
# === Matrix for an element used in 2D thermal problems ===
# ==========================================================
fname = path.join(pth, 'Q4T.K')
try:
    Q4T = load(fname)
except IOError:
    from topy.core.data import Q4T_K
    raise IOError(MSG1)

# ==========================================================
# === Stiffness matrix of a square 4 node 'Q4a5B' element ===
# ==========================================================
# This element is based on the '5-beta' assumed stress element for plane
# stress, but elemental parameters are introduced and selected such that
# spurious zero energy modes are not introduced, for which an investigation
# of characteristic equations of the elemental stiffness matrix is needed.
# Element thickness set = 1. See De Klerk and Groenwold for details.
# Symbolic value of alpha_opt for bending:
alpha2D = (2 * _a**2 * (1 - _nu) * (2 * _nu**2 - _nu + 1)) \
/ (3 * (_nu + 1) * _E**2)
Q4a5B = Q4 - alpha2D * _E * Q4bar  # stiffness matrix

# 3D elements
# ###########################################################################
# ===================================================================
# === Stiffness matrix for a hexahedron 8 node tri-linear 3D element ===
# ===================================================================
fname = path.join(pth, 'H8.K')
try:
    H8 = load(fname)
except IOError:
    from topy.core.data import H8_K
    raise IOError(MSG1)

# ============================================================
# === Stiffness matrix of a cubic 8 node '18-beta' element ===
# ============================================================
fname = path.join(pth, 'H18B.K')
try:
    H18B = load(fname)
except IOError:
    from topy.core.data import H18B_K
    raise IOError(MSG1)
```

```
# ============================================================================
# === Stiffness matrix for a hexahedron 8 node tri-linear 3D element for ===
# === thermal problems.                                                  ===
# ============================================================================
fname = path.join(pth, 'H8T.K')
try:
    H8T = load(fname)
except IOError:
    from topy.core.data import H8T_K
    raise IOError(MSG1)

# EOF elements.py
```

A.4.5 Materials constants module (matlcons.py) description and listing

The material constants are defined in this file, located in the 'data' directory.

```
"""
# ============================================================================
# Material constants and finite element dimensions.
#
# Author: William Hunter <willemjagter@gmail.com>
# Date: 01-06-2007
# Last change: 18-11-2008
# Copyright (C) 2008, William Hunter.
# ============================================================================
"""

from __future__ import division

__all__ = ['_a', '_b', '_c', '_E', '_nu', '_G', '_g', '_k']

# =====================================
# === Element and material constants ===
# =====================================
# NOTE! Any changes you make here is only reflected in the elements once
# you've re-created them! There's a 'recreate_all.py' script that you can run.

_a, _b, _c = 0.5, 0.5, 0.5  # element dimensions (half-lengths) don't change!
_E  = 1  # modulus of elasticity
_nu = 1 / 3  # poisson's ratio
_G = _E / (2 * (1 + _nu))  # modulus of rigidity
_g = _E /  ((1 + _nu) * (1 - 2 * _nu))
_k = 1  # thermal conductivity of steel = 50 (ref. Mills)
```

Appendix

B

ToPy input files

Here we give some (for brevity) of the TPD files used to produce the results in Chapter 6.

```
[ToPy Problem Definition File v2007]

# Author: William Hunter
# The 'classic' 60x20 2d mbb beam, as per Ole Sigmund's 99 line code.

PROB_TYPE  : comp
PROB_NAME  : beam_2d_reci
ETA        : 0.5  # reciprocal approx.
DOF_PN     : 2
VOL_FRAC   : 0.5
FILT_RAD   : 1.5
ELEM_K     : Q4
NUM_ELEM_X : 60
NUM_ELEM_Y : 20
NUM_ELEM_Z : 0
FXTR_NODE_X: 1|21
FXTR_NODE_Y: 1281
LOAD_NODE_Y: 1
LOAD_VALU_Y: -1

NUM_ITER   : 300

P_FAC      : 3
```

Listing B.1: Reference 2D beam problem: TPD file contents

```
[ToPy Problem Definition File v2007]

# Author: William Hunter
# 'Classic' 40x40 2d plate, as per Ole Sigmund's 91 line code,
# Matlab command: toph(40, 40, 0.4, 3.0, 1.2)

PROB_TYPE  : heat
PROB_NAME  : heat_2d_reci
ETA        : 0.5
DOF_PN     : 1
VOL_FRAC   : 0.4
FILT_RAD   : 1.2
P_FAC      : 3
ELEM_K     : Q4T
NUM_ITER   : 100
NUM_ELEM_X : 40
NUM_ELEM_Y : 40
NUM_ELEM_Z : 0
FXTR_NODE_X: 19|23
LOAD_NODE_X: 1|1681
LOAD_VALU_X: 0.01@1681
```

Listing B.2: Reference 2D heat problem: TPD file contents

```
[ToPy Problem Definition File v2007]

# Author: William Hunter
# 'Classic' 40x20 2d domain, as per Ole Sigmund's 104 line code.
# Matlab command: topm(40, 20, 0.3, 3.0, 1.2)

PROB_TYPE  : mech
PROB_NAME  : inverter_2d_eta03
ETA        : 0.3
DOF_PN     : 2
VOL_FRAC   : 0.3
FILT_RAD   : 1.2
P_FAC      : 3
ELEM_K     : Q4
NUM_ITER   : 100
NUM_ELEM_X : 40
NUM_ELEM_Y : 20
NUM_ELEM_Z : 0
FXTR_NODE_X: 20; 21
FXTR_NODE_Y: 1|841|21; 20; 21
LOAD_NODE_X: 1
LOAD_VALU_X: 1

LOAD_NODE_X_OUT: 841 #  (NUM_ELEM_X + 1) * (NUM_ELEM_Y + 1) - NUM_ELEM_Y
LOAD_VALU_X_OUT: -1
```

Listing B.3: Reference 2D mechanism synthesis problem: TPD file contents

```
[ToPy Problem Definition File v2007]

# Author: William Hunter
# The 'classic' 60x20 2d mbb beam, as per Ole Sigmund's 99 line code, but with
# grey-scale filtering.

PROB_TYPE  : comp
PROB_NAME  : beam_2d_reci_gsf
ETA        : 0.5  # reciprocal approx.
DOF_PN     : 2
VOL_FRAC   : 0.5
FILT_RAD   : 1.5
ELEM_K     : Q4
NUM_ELEM_X : 60
NUM_ELEM_Y : 20
NUM_ELEM_Z : 0
FXTR_NODE_X: 1|21
FXTR_NODE_Y: 1281
LOAD_NODE_Y: 1
LOAD_VALU_Y: -1

NUM_ITER   : 100

# Grey-scale filter (GSF)
P_FAC      : 1
P_HOLD     : 25  # num of iters to hold p constant from start
P_INCR     : 0.2  # increment by this amount
P_CON      : 1  # increment every 'P_CON' iters
P_MAX      : 3  # max value of 'P_CON'

Q_FAC      : 1
Q_HOLD     : 25  # num of iters to hold q constant from start
Q_INCR     : 0.05  # increment by this amount
Q_CON      : 1  # increment every 'Q_CON' iters
Q_MAX      : 5  # max value of 'Q_CON'
```

Listing B.4: 2D MBB beam with GSF: TPD file contents

```
[ToPy Problem Definition File v2007]

# Author: William Hunter

PROB_TYPE:   comp
PROB_NAME:   cantilvr_3d_etaopt_gsf
ETA:         0.4
DOF_PN:      3
VOL_FRAC:    0.15
FILT_RAD:    1.5
ELEM_K:      H8
NUM_ELEM_X:  28
NUM_ELEM_Y:  37
NUM_ELEM_Z:  111
FXTR_NODE_X: 1
FXTR_NODE_Y: 1|1102
FXTR_NODE_Z: 1|1102
LOAD_NODE_Y: 122892
LOAD_VALU_Y: -1

NUM_ITER   : 50

# Grey-scale filter (GSF)
P_FAC      : 1
P_HOLD     : 15  # num of iters to hold p constant from start
P_INCR     : 0.2  # increment by this amount
P_CON      : 1  # increment every 'P_CON' iters
P_MAX      : 3  # max value of 'P_CON'

Q_FAC      : 1
Q_HOLD     : 15  # num of iters to hold q constant from start
Q_INCR     : 0.05  # increment by this amount
Q_CON      : 1  # increment every 'Q_CON' iters
Q_MAX      : 5  # max value of 'Q_CON'
```

Listing B.5: 3D cantilever with GSF: TPD file contents

```
[ToPy Problem Definition File v2007]

# Author: William Hunter
# Based on example in Sigmund and Bendsoe, p26, but now with GSF.

PROB_TYPE  : comp
PROB_NAME  : dogleg_3d_etaopt_gsf
ETA        : 0.4
DOF_PN     : 3
VOL_FRAC   : 0.3
FILT_RAD   : 1.8
ELEM_K     : H8
NUM_ELEM_X : 60
NUM_ELEM_Y : 30
NUM_ELEM_Z : 60
FXTR_NODE_X: 931|1861|31; 961|1891|31; 931|961; 1861|1891
FXTR_NODE_Y: 931|1861|31; 961|1891|31; 931|961; 1861|1891
FXTR_NODE_Z: 931|1861|31; 961|1891|31; 931|961; 1861|1891
LOAD_NODE_Y: 113475|113477; 113506|113508; 113537|113539
LOAD_VALU_Y: -1@9

PASV_ELEM  : 1|900; 1801|2700; 3601|4500; 5401|6300; 7201|8100; 9001|9900; 10801|11700;

NUM_ITER   : 50

# Grey-scale filter (GSF)
P_FAC      : 1
P_HOLD     : 15  # num of iters to hold p constant from start
P_INCR     : 0.2  # increment by this amount
P_CON      : 1  # increment every 'P_CON' iters
P_MAX      : 3  # max value of 'P_CON'

Q_FAC      : 1
Q_HOLD     : 15  # num of iters to hold q constant from start
Q_INCR     : 0.05  # increment by this amount
Q_CON      : 1  # increment every 'Q_CON' iters
Q_MAX      : 5  # max value of 'Q_CON'
```

Listing B.6: 3D dogleg with GSF: TPD file contents

```
[ToPy Problem Definition File v2007]

# Author: William Hunter
# 3d cube domain, heat sink on middle of one face,
# nodes have thermal load.

PROB_TYPE  : heat
PROB_NAME  : heat_3d_etaopt_gsf_dquad
ETA        : 0.4  # 'optimal' choice for eta
DOF_PN     : 1
VOL_FRAC   : 0.3
FILT_RAD   : 1.4
ELEM_K     : H8T  # use 'thermal' element
NUM_ELEM_X : 50
NUM_ELEM_Y : 5
NUM_ELEM_Z : 50

# Use only X for heat problems!
FXTR_NODE_X: 139|168
LOAD_NODE_X: 1|15606
LOAD_VALU_X: 0.01@15606

NUM_ITER   : 100

# Grey-scale filter (GSF)
P_FAC      : 1
P_HOLD     : 10  # num of iters to hold p constant from start
P_INCR     : 1  # increment by this amount
P_CON      : 10  # increment every 'P_CON' iters
P_MAX      : 3  # max value of 'P_CON'

Q_FAC      : 1
Q_HOLD     : 20  # num of iters to hold q constant from start
Q_INCR     : 0.0075  # increment by this amount
Q_CON      : 1  # increment every 'Q_CON' iters
Q_MAX      : 3  # max value of 'Q_CON'

# Diagonal quadratic approximation
APPROX     : dquad
```

Listing B.7: 3D heat conduction with GSF and diagonal quadratic approximation: TPD file contents

Appendix

C

The Falk dual and separable problems

C.1 Method

The steps involved in solving a strictly convex and separable multivariable constrained optimisation problem using the Falk dual is:

1. Construct the Lagrangian.
2. Get lambda λ in terms of the design variables x_i by differentiating the Lagrangian $L(x, \lambda)$ with respect to x_i and setting the result equal to zero.
3. Create the Falk dual function by substituting $x_i = x_i(\lambda)$, which we obtained above in 2.
4. Construct the Falk dual problem, thus, maximise the Falk dual function, handling the side constraints directly.

C.2 Example 1 (1D)

The first example illustrates the method in one of its simplest forms, namely, we minimise a single variable, non-linear objective function subject to a single linear constraint and side constraints.

C.2.1 Primal problem

$$\begin{aligned} \min_x \quad & f(x) = 5x^2 + 10, \\ \text{s.t.} \quad & g(x) = 2x - 1 \geq 0, \\ & 1.0 \leq x \leq 1.5. \end{aligned}$$

The unconstrained optimum is clearly at $x = 0$, if $g(x)$ is active (ignoring the side constraints) the optimum is at $x = 0.5$, and if the side constraints are included the solution is at $x^* = 1$. The graphical solution is easily obtained and shown in Figure C.2.1.

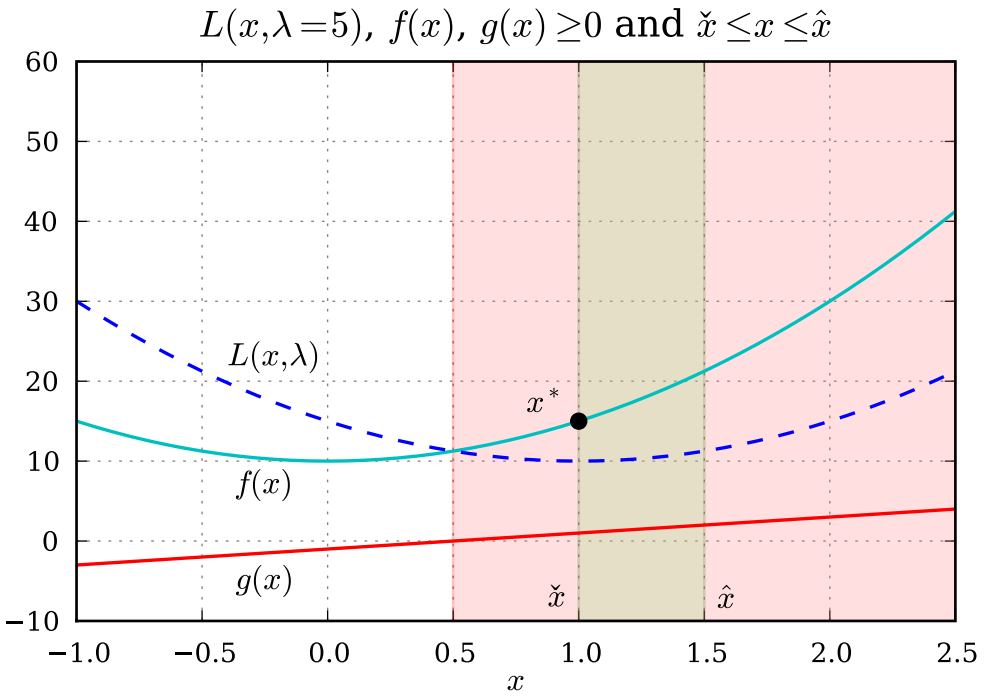


Figure C.1: Falk dual example for 1D problem.

C.2.2 Dual problem

C.2.2.1 Lagrangian

We first construct the Lagrangian:

$$\begin{aligned} L(x,\lambda) &= f(x) - \lambda g(x) \\ &= 5x^2 + 10 - \lambda(2x-1). \end{aligned}$$

C.2.2.2 Falk's dual function

The Falk dual function is constructed as:

$$\gamma(\lambda) = \min_x \{L(x,\lambda) : 1.0 \leq x \leq 1.5\}.$$

Minimising the Lagrangian with respect to x, we find

$$\begin{aligned} \frac{\partial L(x,\lambda)}{\partial x} = 0 = \frac{\partial f}{\partial x} - \lambda \frac{\partial g}{\partial x}, \\ \lambda = \frac{\partial f}{\partial x} \Big/ \frac{\partial g}{\partial x}, \\ \lambda = 2 \times 5 \times x - \lambda \times 2, \\ x = \lambda/5, \end{aligned}$$

hence we have x in terms of λ. By substituting $x = \lambda/5$ into the the Lagrangian, we obtain the Falk dual function (and eventually the Falk dual formulation as will be shown below). The fact that we can get x as a function of λ is crucial, otherwise we will not be able to construct the Falk dual problem.

C.2.2.3 Falk's dual problem

The Falk dual formulation is:

$$\begin{aligned} &\text{Maximise} \quad \gamma(\lambda) = (5)(\lambda/5)^2 + 10 - \ (\lambda)(2\lambda/5 - 1), \\ &\text{subject to} \quad \lambda \geq 0. \end{aligned}$$

We now maximise the Falk dual function using the relationship $x = \lambda/5$ obtained earlier and choose λ subject to the side constraints $1.0 \leq x \leq 1.5$ and of course $\lambda \geq 0$. This can be done using a number of approaches and any 1D method will work. We use the simple bisection method (as opposed to the superior golden section method)[1] for illustrative purposes. Here is a simple Python implementation for this example:

```
# Some constants:
x_upp = 1.5
x_low = 1.0
l1, l2 = 0.0, 100.0
l, l_old = 1, 0
eps = 0.01
cnt = 1

# x in terms of lambda, i.e., x = x(lambda):
def x_new(lam):
    x_new = lam / 5.0
    if x_new <= x_low:
        x_new = x_low
    elif x_new >= x_upp:
        x_new = x_upp
    return x_new

# Constraint function:
def g(x):
    return 2.0 * x - 1.0

# Calculate new lambda (bisection):
def l_new(l1, l2):
    return (l1 + l2) / 2.0

# Bisection algorithm:
while abs(l - l_old) >= eps:
    l_old = l
    l = l_new(l1, l2)
    x = x_new(l)
    if g(x) < 0:
        l1 = l
    else:
        l2 = l
    print 'Iter.', cnt, ': x =', x,\
', diff. lambda =', l - l_old
    cnt += 1

# EOF
```

The output from the script is:

[1] Bisection is also used in the 99-line code.

```
Iter. 1 : x = 1.5 , diff. lambda = 49.0
Iter. 2 : x = 1.5 , diff. lambda = -25.0
Iter. 3 : x = 1.5 , diff. lambda = -12.5
Iter. 4 : x = 1.25 , diff. lambda = -6.25
Iter. 5 : x = 1.0 , diff. lambda = -3.125
Iter. 6 : x = 1.0 , diff. lambda = -1.5625
Iter. 7 : x = 1.0 , diff. lambda = -0.78125
Iter. 8 : x = 1.0 , diff. lambda = -0.390625
Iter. 9 : x = 1.0 , diff. lambda = -0.1953125
Iter. 10 : x = 1.0 , diff. lambda = -0.09765625
Iter. 11 : x = 1.0 , diff. lambda = -0.048828125
Iter. 12 : x = 1.0 , diff. lambda = -0.0244140625
Iter. 13 : x = 1.0 , diff. lambda = -0.01220703125
Iter. 14 : x = 1.0 , diff. lambda = -0.006103515625
```

The numerical solution is $\lambda = 0.0$ and $x = 1.0$ (for $\epsilon = 0.01$ in the bisection algorithm), which corresponds to the graphical solution. Also note that the side constraint on x is active.

Of course we can also differentiate $\gamma(\lambda)$ with respect to λ and set the result equal to zero, but this will give us the constrained optimum since the side constraints are ignored. Thus, differentiating $\gamma(\lambda)$ and setting equal to zero gives $\lambda = 2.5$ (and therefore $x = 0.5$), as expected, since $g(x)$ is active (see Figure C.2.1).

C.3 Example 2 (2D)

In the second example we minimise a two-variable, non-linear objective function subject to a single linear constraint and side constraints on each variable. The graphical solution is again easily obtained and is shown below in Figure C.3.

C.3.1 Primal problem

$$\begin{aligned} \min_{x} \quad & f(\bar{x}) = (x_1 + 2)^2 + (2x_2 + 3)^2, \\ \text{s.t.} \quad & g(\bar{x}) = x_1 + x_2 - 10 \geq 0, \\ & 3 \leq x_1 \leq 8, \\ & 1 \leq x_2 \leq 10. \end{aligned}$$

C.3.2 Dual problem

C.3.2.1 Lagrangian

The Lagrangian:

$$\begin{aligned} L(\bar{x}, \lambda) &= f(\bar{x}) - \lambda g(\bar{x}) \\ &= (x_1 + 2)^2 + (2x_2 + 3)^2 - \lambda(x_1 + x_2 - 10). \end{aligned}$$

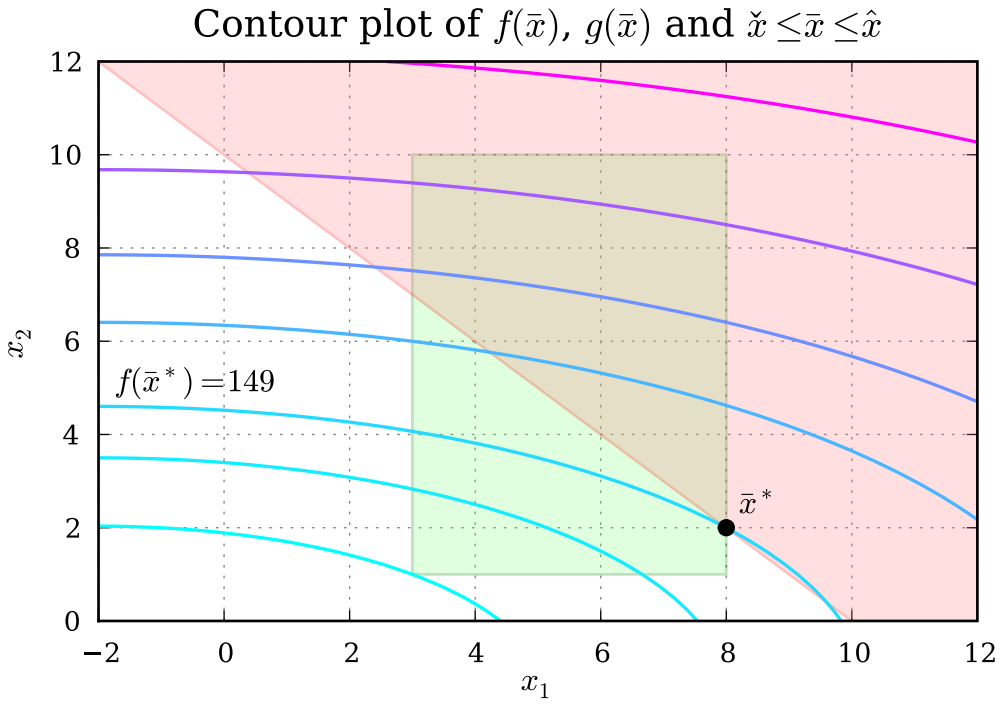


Figure C.2: Falk dual example for 2D problem.

C.3.2.2 Falk's dual function

The Falk dual function:

$$\gamma(\lambda) = \min_x \{L(x, \lambda) : \check{x} \leq \bar{x} \leq \hat{x}\}.$$

Minimising the Lagrangian with respect to $\bar{x}$, we find

$$x_1 = \frac{\lambda - 4}{2}$$

and

$$x_2 = \frac{\lambda - 12}{12}.$$

C.3.2.3 Falk's dual problem

The Falk dual formulation is:

$$\begin{aligned} &\text{Maximise} \quad \gamma(\lambda) = (\frac{\lambda - 4}{2} + 2)^2 + (2\frac{\lambda - 12}{12} + 3)^2 - \lambda(\frac{\lambda - 4}{2} + \frac{\lambda - 12}{12} - 10), \\ &\text{subject to} \quad \lambda \geq 0. \end{aligned}$$

As for the first example, we use bisection to solve for λ and we get $x_1 = 8.0$ and $x_2 = 2.0$ as our solution, with the side constraint for x_1 being active.

Appendix

D

wxMaxima session scripts

These files, which are really just records of work sessions, can be opened in wxMaxima (v. 0.7.1).

D.1 Computation of $Q4(\hat{\alpha})$ element stiffness matrix eigenvalues

```
/* [wxMaxima batch file version 1] [ DO NOT EDIT BY HAND! ]*/

/* [wxMaxima: comment start ]
Computation of Q4(alpha) element stiffness matrix eigenvalues.
Author: William Hunter
Date: 07-Nov-2008
   [wxMaxima: comment end   ] */

/* [wxMaxima: input   start ] */
N1: (a - x) * (b - y) / (4 * a * b);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
N2: (a + x) * (b - y) / (4 * a * b);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
N3: (a + x) * (b + y) / (4 * a * b);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
N4: (a - x) * (b + y) / (4 * a * b);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
B1: diff([N1, 0, N2, 0, N3, 0, N4, 0], x);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
B2: diff([0, N1, 0, N2, 0, N3, 0, N4], y);
```

```
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
B3y: diff([N1, 0, N2, 0, N3, 0, N4, 0], y);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
B3x: diff([0, N1, 0, N2, 0, N3, 0, N4], x);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
B3: B3x + B3y;
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
B: matrix(B1, B2, B3);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: comment start ]
Constitutive matrix (material property matrix):
   [wxMaxima: comment end   ] */

/* [wxMaxima: input   start ] */
C: (E / (1 - nu**2)) * matrix([1, nu, 0], [nu, 1, 0], [0,  0, (1 - nu) / 2]);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
CB: C . B;
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
delCB1x: map(lambda([u], diff(u, x)), row(CB,1));
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
delCB1y: map(lambda([u], diff(u, y)), row(CB,3));
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
delCB1: delCB1x + delCB1y;
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
delCB2x: map(lambda([u], diff(u, x)), row(CB,3));
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
delCB2y: map(lambda([u], diff(u, y)), row(CB,2));
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
delCB2: delCB2x + delCB2y;
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
Bbar: addrow(delCB1, delCB2);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
```

```
dK: transpose(B) . C . B;
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
dKbar: transpose(Bbar) . Bbar;
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
K: integrate(dK, x, -a, a);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
K: integrate(K, y, -b, b);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
Kbar: integrate(dKbar, x, -a, a);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
Kbar: integrate(Kbar, y, -b, b);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
K: subst(a, b, K);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
K: radcan(K);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
Kbar: subst(a, b, Kbar);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
Kbar: radcan(Kbar);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
Kalfa: K - alfa * E * Kbar;
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
Kalfa: radcan(Kalfa);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
Kalfa: factor(Kalfa);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
eigKalfa: eigenvalues(Kalfa);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: comment start ]
Calculate critical value of alpha:
   [wxMaxima: comment end   ] */
```

```
/* [wxMaxima: input   start ] */
alfa_crit_eig: eigKalfa[1][2];
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
alfa_crit: solve([alfa_crit_eig], [alfa]);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
alfa_crit: radcan(alfa_crit);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
alfa_crit: factor(alfa_crit);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: comment start ]
Calculate optimal value of alpha by setting equal to equivalent eigenvalue of
the 5-beta element (E / 3 in this case) and solving for alpha:
   [wxMaxima: comment end   ] */

/* [wxMaxima: input   start ] */
alfa_opt: solve([alfa_crit_eig - E / 3], [alfa]);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
alfa_opt: factor(alfa_opt);
/* [wxMaxima: input   end   ] */

/* Maxima can't load/batch files which end with a comment! */
"Created with wxMaxima"$
```

D.2 Computation of 5β-NC element stiffness matrix eigenvalues

```
/* [wxMaxima batch file version 1] [ DO NOT EDIT BY HAND! ]*/

/* [wxMaxima: comment start ]
Computation of 5beta-NC element stiffness matrix eigenvalues.
Author: William Hunter
Date: 10-Nov-2008
   [wxMaxima: comment end   ] */

/* [wxMaxima: input   start ] */
N1: (a - x) * (b - y) / (4 * a * b);
N2: (a + x) * (b - y) / (4 * a * b);
N3: (a + x) * (b + y) / (4 * a * b);
N4: (a - x) * (b + y) / (4 * a * b);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
B1: diff([N1, 0, N2, 0, N3, 0, N4, 0], x);
B2: diff([0, N1, 0, N2, 0, N3, 0, N4], y);
B3y: diff([N1, 0, N2, 0, N3, 0, N4, 0], y);
B3x: diff([0, N1, 0, N2, 0, N3, 0, N4], x);
B3: B3x + B3y;
B: matrix(B1, B2, B3);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: comment start ]
Constitutive matrix (material property matrix):
   [wxMaxima: comment end   ] */

/* [wxMaxima: input   start ] */
C: (E / (1 - nu**2)) * matrix([1, nu, 0], [nu, 1, 0], [0,  0, (1 - nu) / 2]);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
PI: ident(3);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
PH2: matrix([y / b, 0], [0, x / a], [0, 0]);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
P: addcol(PI, PH2);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
transP: transpose(P);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
G: transP . B;
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
G: integrate(G, x, -a, a);
```

```
G: integrate(G, y, -b, b);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
G: subst(a, b, G);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
invC: ratsimp(invert(C));
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
H: ratsimp(transP . invC);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
H: ratsimp(H . P);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
H: integrate(H, x, -a, a);
H: integrate(H, y, -b, b);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
H: ratsimp(subst(a, b, H));
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
transG: transpose(G);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
invH: ratsimp(invert(H));
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
K: transG . invH . G;
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
K: subst(a, b, K);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
K: ratsimp(K);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
K: factor(K);
/* [wxMaxima: input   end   ] */

/* [wxMaxima: input   start ] */
eigK: eigenvalues(K);
/* [wxMaxima: input   end   ] */

/* Maxima can't load/batch files which end with a comment! */
"Created with wxMaxima"$
```