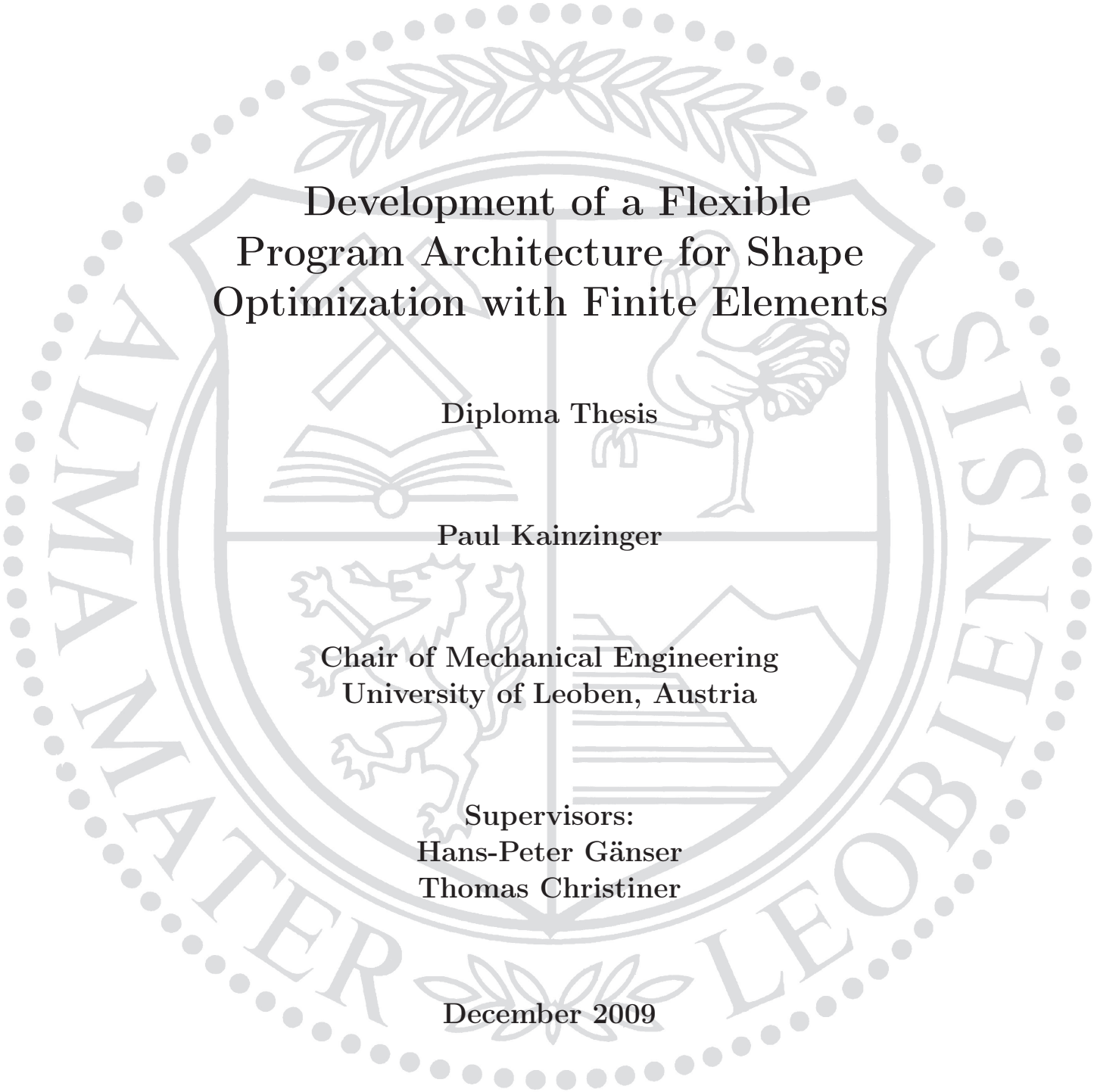

Development of a Flexible Program Architecture for Shape Optimization with Finite Elements

Diploma Thesis

Paul Kainzinger

Chair of Mechanical Engineering
University of Leoben, Austria

Supervisors:
Hans-Peter Gänser
Thomas Christiner

December 2009

Paul Kainzinger
University of Leoben
Franz–Josef–Straße 18

A–8700 Leoben, Austria

Internet:	http://amb.mu-leoben.at/
E–Mail:	amb@mu-leoben.at
	fatigue@mu-leoben.at
	paul.kainzinger@stud.unileoben.ac.at
Tel.:	++43 (0)3842 402 1401
Fax.:	++43 (0)3842 402 1402

Affidavit

I declare in lieu of oath, that I wrote this thesis and performed the associated research myself, using only literature cited in this volume.

Paul Kainzinger
Leoben, Dec. 2009

Acknowledgments

The present thesis was written during my studies in mechanical engineering, focusing on computational design, at the Chair of Mechanical Engineering in the Department of Product Engineering at the University of Leoben.

Financial Support of part of this work by the Austrian Government (Federal Ministry of Transport, Innovation and Technology and Federal Ministry of Economy, Family and Youth) and the Province of Styria via the Austrian Research Promotion Agency (Österreichische Forschungsförderungsgesellschaft mbH) and the Styrian Business Promotion Agency (Steirische Wirtschaftsförderungsgesellschat mbH) within the framework of the K2 Research Center for Materials, Processing and Product Engineering, a member of the Austrian COMET Center of Competence Program, based at Materials Center Leoben, is gratefully acknowledged.

First of all, I would like to express my sincere gratitude to my supervisors Priv.–Doz. Dipl.–Ing. Dr. mont. Hans–Peter Gänser and Dipl.–Ing. Thomas Christiner for their excellent support and assistance throughout my work.

I want to use this opportunity to thank Univ. Prof. Dipl.–Ing. Dr. techn. Wilfried Eichlseder, the head of the Chair of Mechanical Engineering, for laying the foundations that made this thesis possible.

Moreover I would like to thank Bernd Maier for his helpful discussions and useful remarks. Furthermore I express my deep gratitude to all the employ-

ees of the Chair of Mechanical Engineering who helped me during this thesis.

On this way I want to thank my girlfriend, Katharina Bruckmoser, for her loving support and my parents, Elisabeth and Hannes Kainzinger, for offering me the oportunity to this excellent education.

Last but not least, I would like to thank all my colleagues during my studies in Leoben for their wonderful cooperation and beautiful years in Leoben.

Abstract

Within the scope of this thesis a flexible interface (Interface for Parametric Optimization, IPO) between the finite element solver Abaqus and the open source optimization library DAKOTA (Design Analysis Kit for Optimization and Terascale Applications) was developed. Finite element models created with Abaqus can be parametrized and optimized with respect to an arbitrary objective function and optional restrictions. Any mathematical combination of output variables available in Abaqus may serve as an objective function or restriction.

DAKOTA provides a wide variety of different algorithms for optimization, parametric studies, uncertainty quantification and many other applications. Gradient based algorithms as well as gradient free methods, e.g., evolutionary strategies, can be chosen for solving the optimization problem. The IPO combines the advantages of both software packages. One can use the finite element solver Abaqus, which is capable of solving highly nonlinear (material as well as geometric nonlinearities) engineering problems and join it with the extensive optimization and parametric study capabilities of DAKOTA.

The Abaqus Python application programming interface (API) serves as an easy–to–use basis for the coding, since all Abaqus pre– and postprocessing commands are available in this API. An object oriented approach was chosen for the Interface for Parametric Optimization since is fits best into the Abaqus Python API and provides a convenient way for further extensions of the interface.

The program was applied to the optimization of a simple truss construction and a more sophisticated bridge construction with their total weight as an objective function. The differences between several different optimization algorithms are then discussed in detail, highlighting their advantages and disadvantages.

Kurzfassung

Im Rahmen dieser Arbeit wurde eine flexible Schnittstelle (Interface for Parametric Optimization, IPO) zwischen dem Finite Elemente Programm Abaqus und der Open Source Optimierungsbibliothek DAKOTA (Design Analysis Kit for Optimization and Terascale Applications) entwickelt. Finite Elemente Modelle können mit Abaqus parametrisiert und auf eine beliebige Zielfunktion unter Berücksichtigung von optionalen Restriktionen optimiert werden. Alle in Abaqus verfügbaren Ausgabevariablen können beliebig zu Zielfunktionen oder Restriktionen kombiniert werden.

DAKOTA bietet eine Vielfalt an unterschiedlichen Algorithmen für Optimierungen, Parameterstudien, die Vorhersage der Ergebnisunschärfe sowie viele weitere Anwendungen. Gradienten–basierte Verfahren sowie gradientenfreie Methoden wie z.B. evolutionäre Algorithmen können zur Optimierung verwendet werden. Das IPO kombiniert die Vorteile beider Programme, die Fähigkeit von Abaqus hoch nichtlineare (material– sowie geometrische Nichtlinearitäten) Probleme zu lösen sowie die weitreichenden Optimierungsmetoden bzw. Möglichkeiten für Parameterstudien von DAKOTA.

Die von Abaqus zur Verfügung gestellte Python Programmierschnittstelle dient als Basis für die entwickelte Software, da auf alle Pre– bzw. Postprocessing Befehle einfach zugegriffen werden kann. IPO wurde objektorientiert in der Programmiersprache Python geschrieben, da dies sehr gut zu der vorhandenen Programmierschnittstelle passt, bzw. eine spätere Erweiterung erleichtert.

Die entwickelte Software wurde anschließend auf zwei Beispiele wurden anschließend angewandt, die Gewichtsoptimierung einfaches Fachwerk und einer aufwändigeren Brückenkonstruktion. Diese beiden Fachwerke wurden auf ihr Gewicht hin optimiert. Verschiedene unterschiedliche Optimierungsalgorithmen wurden untersucht und deren Vor– bzw. Nachteile diskutiert.

Contents

Chapter 1

Introduction

Due to the constantly rising needs of the modern economy for optimizing mechanical components and thereby decreasing production costs and increasing lifetime, stable and easy–to–use methods need to be developed to achieve these goals. With computational power rising steadily and simultaneously getting cheaper, numerical simulation methods like, e.g., the finite element method become more and more affordable.

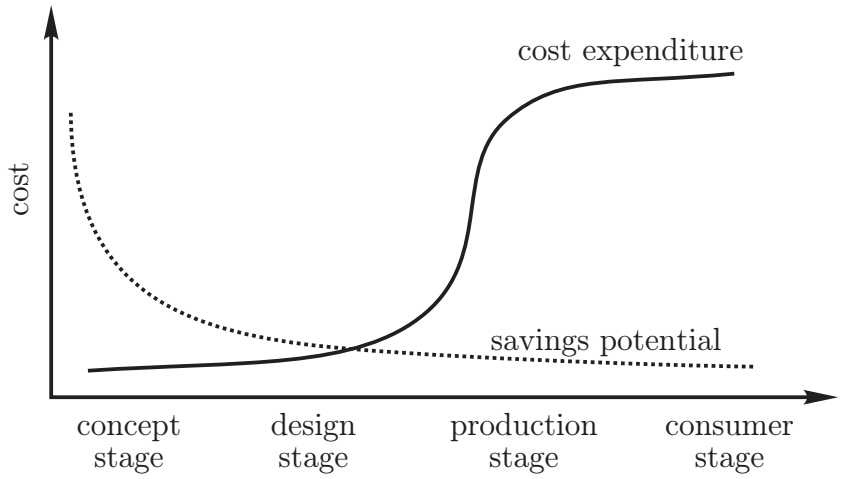

Figure 1.1: Product development cycle

In Fig. 1.1 one can see a typical product development cycle. It is clearly derivable that the costs caused by a single production step increase dramatically the further the cycle proceeds while the savings potential decreases.

Hence it is obvious that it is most efficient to save costs in the early design stage of a project. Modern simulation and optimization tools provide the capability of doing so. They allow the designers and engineers to find the optimal solution for a problem without having to go through complex and expensive trial and error procedures. The variety of these programs and tools is very rich, they go from small and simple optimization tasks (e.g., Microsoft Excel solver), to complex program solutions like ABAQUS or TOSCA. The aim of this thesis is to develop an easy–to–use tool for parametric optimization with ABAQUS. An interface called IPO[1] is implemented which allows a general application of various optimization, least squares and parametric study algorithms. DAKOTA (Design Analysis Kit for Optimization and Terascale Applications) [3], an open source library of optimization algorithms, is used to provide the mathematical capabilities for the optimization loop.

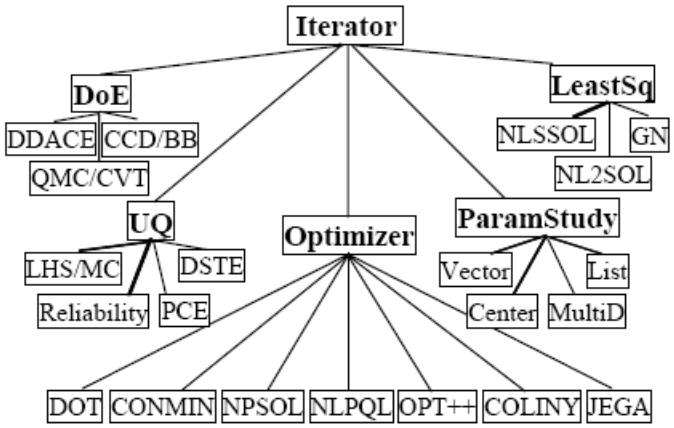

Figure 1.2: Capabilities of DAKOTA

Fig. 1.2 shows an overview of all the algorithms implemented in DAKOTA. The ABAQUS Python API[2] [21] is used as a basis for all coding needed to complete the optimization loop. It provides an object–oriented scripting framework to control all actions from ABAQUS/Standard and ABAQUS/-Explicit. This API is used to do all the pre– and postprocessing to handle the

[1]Interface for Parametric Optimization
[2]Application Programming Interface

input parameters given from DAKOTA and to pass the restrictions and objective functions back. One of the main advantages of this interface is that it grants ABAQUS access to the enormous capabilities of DAKOTA. One can run simple parametric studies as well as much more complex nonlinear restricted optimization problems with multiple objective functions. This provides a way for optimizing components from an early design stage of the product development cycle and therefore finding and fixing problems within a short amount of time.

Chapter 2

Terminology

According to [18] the following general definitions apply.

Optimization algorithm: Mathematical method for optimizing an objective function with or without following certain restrictions.

Optimization method: Combination of optimization approaches and optimization algorithm for solving an optimization problem.

Optimization strategy: Method of reducing a complex optimization problem to a more basic system which represents the original one but is much easier to solve.

Objective function: Mathematical formulation of one or more design goals.

Restriction: Mathematical formulation of certain constraints that have to be complied to.

Simulation model: Mathematical formulation of the model characteristics.

State variable: Response from the simulation model.

Variable: A changeable parameter in the simulation model.

Initial point: Set of start values for the variables of the simulation model.

Chapter 3

Mathematical Background

This Chapter describes the mathematical background needed to define the existence and the position of a local or global optimum.

3.1 Formulation

An arbitrary optimization problem can be expressed by minimizing an objective function

$$\min \mathbf{f}(\mathbf{x}) \tag{3.1}$$

complying to the following restrictions:

$g_j(\mathbf{x}) \leq 0$	$j = 1 \ldots m_g$	inequality constraints
$h_k(\mathbf{x}) = 0$	$k = 1 \ldots m_h$	equality constraints
$x_i^l \leq x_i \leq x_i^u$	$i = 1 \ldots n$	upper and lower bounds

With m_g being the number of inequality constraints, m_h the number of equality constraints and n the number of degrees of freedom. The confinement to $\min \mathbf{f}(\mathbf{x})$ does not violate the general formulation since a maximization problem can always be transformed into a minimization problem via $\max \mathbf{f}(\mathbf{x}) = -\min \mathbf{f}(\mathbf{x})$ or $\max \mathbf{f}(\mathbf{x}) = \min(-\mathbf{f}(\mathbf{x}))$. The same applies to inequality and equality constraints.

In closed formulation the optimization problem can be written as:

$$\mathbf{f}^{*}(\mathbf{x}) = \min\{\mathbf{f}(\mathbf{x}) \mid \mathbf{x} \in \mathbf{X}\} \text{ with } \mathbf{X} = \{\mathbf{x} \in \Re^{n} \mid \mathbf{g}(\mathbf{x}) \leq 0,\ \mathbf{h}(\mathbf{x}) = 0\} \tag{3.2}$$

Where $\Re^{n}$ is the set of n–dimensional real numbers, $\mathbf{X}$ the design space, $\mathbf{h}(\mathbf{x})$ the vector of equality constraints and $\mathbf{g}(\mathbf{x})$ the vector of inequality constraints.

3.2 Global and Local Minimum

One of the main problems in optimizing a design problem is the fact that there might be more than one optimum. Depending on the shape of the objective function it may be easy or more difficult for a mathematical algorithm to find the global minimum and not get stuck in a local one. Many algorithms may only find a local minimum. If one wants to be sure to really find the global optimum one has to be certain that there is only one local minimum which is, in this case, also the global one. Therefore the function $f(x)$ needs to be convex within the interval $x \in \left[x^{l},\, x^{u}\right]$. A function is called convex if

$$f(\theta x_A + (1-\theta) x_B) \leq \theta f(x_A) + (1-\theta) f(x_B) \tag{3.3}$$

for all $x_A, x_B \in \left[x^{l}, x^{u}\right]$ and $\theta \in [0,1]$. As one can see in Fig. 3.1, Eq. 3.3 illustrates that a straight line between two points may never touch or intersect the function line.

Fig. 3.1 also shows a function which is not convex, because a straight line from point A to point B intersects the function line twice within the interval $\left[x^{l}, x^{u}\right]$.

As shown in Fig. 3.2, the convexity is actually a too strong condition for a function to have only one minimum. The minimum shown there would be found by an optimization algorithm.

The restrictions have to be convex too. A set $\mathbf{M}$ is called convex if

$$\mathbf{y} = \theta \mathbf{x_A} + (1-\theta)\,\mathbf{x_B} \in \mathbf{M} \tag{3.4}$$

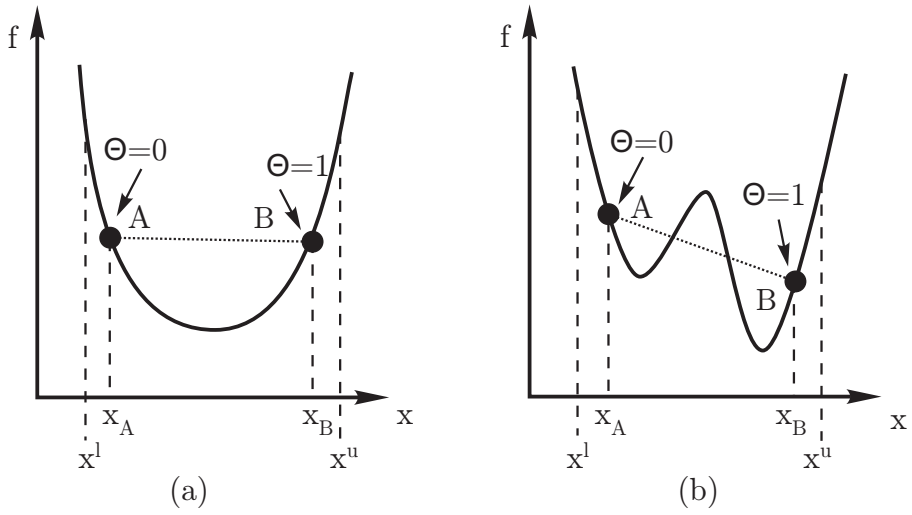

Figure 3.1: Convex and non–convex function

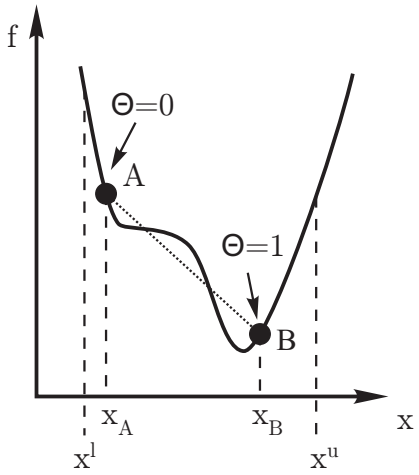

Figure 3.2: Non–convex function with only one minimum

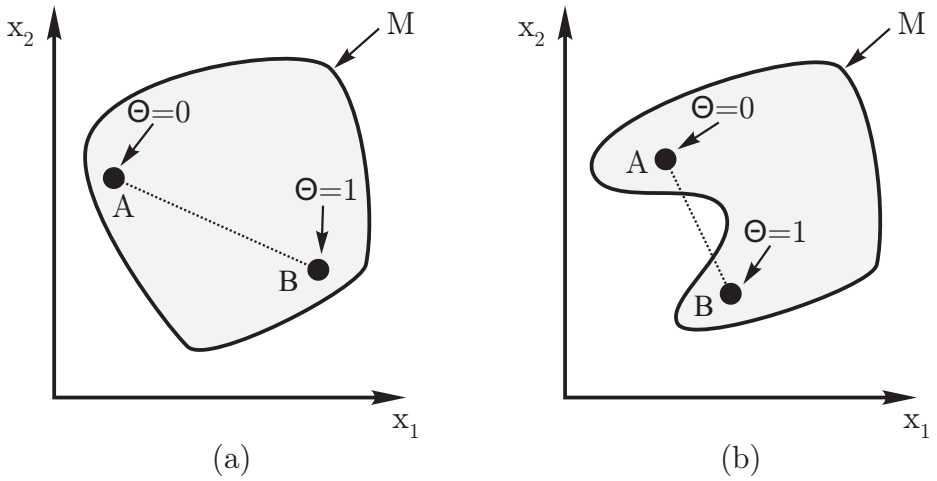

Figure 3.3: Convexity of restrictions

with $x_A, x_B \in \mathbf{M}$ and $\theta \in [0, 1]$.

Fig. 3.3 shows a convex and a non–convex design space. Generally speaking, one can say that an optimization problem is convex if the optimization function $\mathbf{f}(\mathbf{x})$ is convex corresponding to Eq. 3.3 and the restrictions $\mathbf{g}(\mathbf{x})$ and $\mathbf{h}(\mathbf{x})$ are convex corresponding to Eq. 3.4.

The fact that the shape of an objective function for an engineering problem is usually not analytically describable and therefore only available at discrete points makes it very hard to decide whether a problem is convex or not. Hence one can never be sure to have found the global and not only a local optimum. So it can be useful to combine the features of several optimization algorithms by running one after the other, or running the simulation with different initial points. [9]

3.3 Existence of a Local Minimum

For a local minimum to exist at a certain point $\mathbf{x}^*$ the following equation needs to be fulfilled:

$$\begin{pmatrix} \dfrac{\partial \mathbf{f}}{\partial x_1} \\ \dfrac{\partial \mathbf{f}}{\partial x_2} \\ \vdots \\ \dfrac{\partial \mathbf{f}}{\partial x_n} \end{pmatrix}_{\mathbf{x}^*} = 0 \tag{3.5}$$

Eq. 3.5 shows the necessary condition for a local minimum, where f is the objective function, n is the number of variables and $x_1 \dots x_n$ are the variables. All partial derivatives of the objective function evaluated at $\mathbf{x}^*$ need to be zero.

$$\begin{pmatrix} \dfrac{\partial^2 \mathbf{f}}{\partial x_1^2} & \dfrac{\partial^2 \mathbf{f}}{\partial x_1 \partial x_2} & \cdots & \dfrac{\partial^2 \mathbf{f}}{\partial x_1 \partial x_n} \\ \dfrac{\partial^2 \mathbf{f}}{\partial x_2 \partial x_1} & \dfrac{\partial^2 \mathbf{f}}{\partial x_2^2} & \cdots & \dfrac{\partial^2 \mathbf{f}}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 \mathbf{f}}{\partial x_n \partial x_1} & \dfrac{\partial^2 \mathbf{f}}{\partial x_n \partial x_2} & \cdots & \dfrac{\partial^2 \mathbf{f}}{\partial x_n^2} \end{pmatrix} \tag{3.6}$$

A sufficient condition for a local minimum can be formulated using the Hessian matrix (Eq. 3.6) which is assembled using the second partial derivatives. It needs to be positive definite, i.e., all eigenvalues λ according to Eq. 3.7 need to be greater than zero:

$$\det\left[\begin{pmatrix} \dfrac{\partial^2\mathbf{f}}{\partial x_1^2} & \dfrac{\partial^2\mathbf{f}}{\partial x_1\partial x_2} & \cdots & \dfrac{\partial^2\mathbf{f}}{\partial x_1\partial x_n} \\ \dfrac{\partial^2\mathbf{f}}{\partial x_2\partial x_1} & \dfrac{\partial^2\mathbf{f}}{\partial x_2^2} & \cdots & \dfrac{\partial^2\mathbf{f}}{\partial x_2\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2\mathbf{f}}{\partial x_n\partial x_1} & \dfrac{\partial^2\mathbf{f}}{\partial x_n\partial x_2} & \cdots & \dfrac{\partial^2\mathbf{f}}{\partial x_n^2} \end{pmatrix} - \lambda \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}\right] = 0 \tag{3.7}$$

Chapter 4

Categorization and Description of Commonly Used Optimization Algorithms

This chapter will focus on several commonly used optimization algorithms. Since the number of algorithms developed is almost infinite, only a small representative selection is highlighted.

4.1 Gradient Based Algorithms

As the name suggests, this type of algorithms takes the local gradients into account. As stated in Eq. 3.5, the first derivative, otherwise known as the gradient, of a function has a significant influence on finding an optimum. Gradient based algorithms calculate the local derivative either numerically or, if available, analytically. They then use the information gained for continuing their iteration.

One of the main advantages of these methods is their speed of convergence. Since the gradient gets smaller and finally reaches zero at a minimum, it is obvious that such methods are very quick in finding the optimal solution. Their main advantage is also their greatest shortcoming: these methods only work properly if the objective function is smooth. Discontinuities or rapid

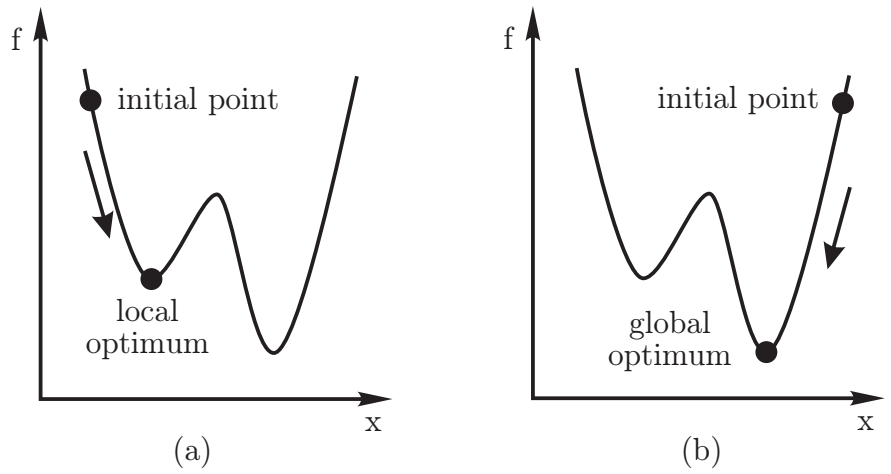

Figure 4.1: Gradient based algorithms

changes in the objective function can lead to incorrect results or divergence. Fig. 4.1 shows another disadvantage, the result of a gradient–based minimum search depends on the starting point. Choosing a bad starting position may lead to only finding a local minimum instead of the global optimum. Another disadvantage is the fact that gradient based algorithms may get stuck in a local minimum, there is no guarantee that they will find the global one.

4.1.1 Newton–Raphson Method

4.1.1.1 Newtons' Method for Root Finding

The Newton–Raphson Method is a gradient based algorithm used for finding the roots of a function. This method converges quadratically if the initial point is sufficiently near the root. On the other hand, Newton's algorithm may converge never, or only with difficulty, if the initial point is chosen poorly.

The basic idea of this method is to linearize the function by means of its tangent. Fig. 4.2 illustrates the iteration process. After calculating the local tangent, the root of the tangent is determined.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{4.1}$$

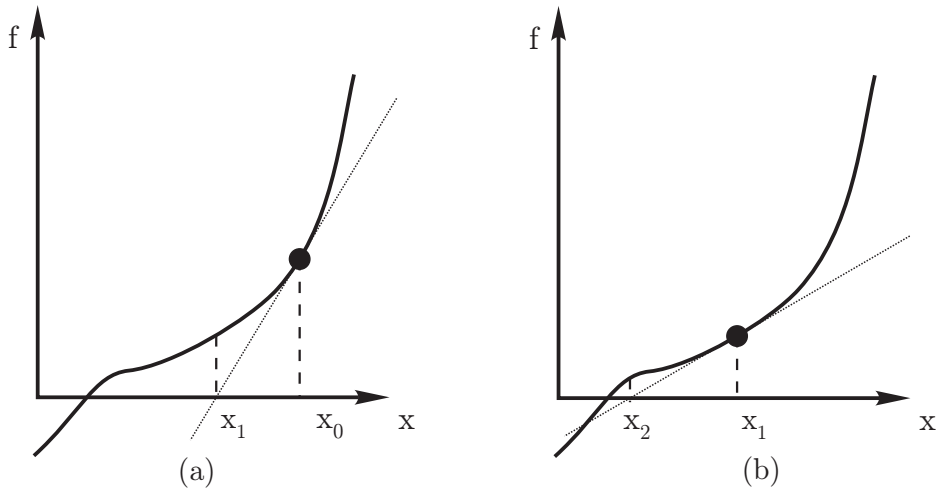

Figure 4.2: Newton–Raphson method

This value serves as the new input for the next iteration. If a certain convergence tolerance is reached, the iteration stops and the current value x_n is returned as the desired solution.

4.1.1.2 Newton's Method in Optimization

This method can easily be transformed into an approach to finding an optimum. Since the first derivative of a function, otherwise known as the gradient, has to be 0 according to Eq. 3.5, one can apply Newton's method to the local gradient of an objective function and find its optimum according to

$$\mathbf{f}^* \left(\mathbf{x}^*\right) = \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \left(\mathbf{x}^*\right) = 0 \tag{4.2}$$

[11] describes the unconstrained algorithm as follows for the iteration k, initial value $\mathbf{x}^0$ for $k = 0$:

For $\mathbf{x}^k$ given and while $\left\| \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \left(\mathbf{x}^k\right) \right\| > \varepsilon$, do

1. compute the Jacobian $\mathbf{A}^k = \frac{\partial^2 \mathbf{f}}{\partial \mathbf{x}^2} \left(\mathbf{x}^k\right)$
2. solve the linear system $\mathbf{A}^k \cdot d\mathbf{x} = -\frac{\partial \mathbf{f}}{\partial \mathbf{x}} \left(\mathbf{x}^k\right)$
3. set $\mathbf{x}^{k+1} = \mathbf{x}^k + d\mathbf{x}$

One disadvantage of this method is that there is no guaranteed convergence for dimensions higher than one. Quadratic convergence can also only be found within a certain neighborhood of $\mathbf{x}^*$; the size of this neighborhood depends on the individual shape of the objective function.

4.1.2 Direction Set Methods in Multidimensions

If one wants to minimize a multidimensional objective function, the complexity of the problem increases dramatically. But a multidimensional optimization can be reduced to an optimization with only one variable. This is done by starting at an initial point $\mathbf{P}$ of an N–dimensional function $\mathbf{f}(\mathbf{P})$. One then proceeds along any vector direction $\mathbf{n}$ and the function $\mathbf{f}(\mathbf{P})$ can now be minimized along this direction with a one–dimensional method. Different methods only differ in the way they choose the vector direction n and in the way they find the optimum along this line (otherwise known as line minimization). A schematic line minimization algorithm is given below [16]

1. given the input vectors $\mathbf{P}$ and $\mathbf{n}$, and the function $\mathbf{f}(\mathbf{P})$
2. find the scalar λ that minimizes $\mathbf{f}(\mathbf{P} + \lambda\mathbf{n})$
3. replace $\mathbf{P}$ by $\mathbf{P} + \lambda\mathbf{n}$

An exemplary algorithm for this method is the method of steepest descent, where the direction $\mathbf{n}$ is chosen such that the gradient of $\mathbf{f}$ at $\mathbf{P}$ has its maximum. For further information on these methods the reader is referred to [16, 10, 9].

Fig. 4.3 shows a simple example for line minimization, the ellipses represents isocontours of the objective function, the arrows the path of descent.

4.2 Gradient Free Algorithms

This type of algorithm does not take the information gained by the gradient into account. Most of them use stochastically generated variable values to calculate the optimal solution for a problem.

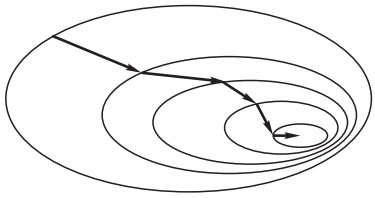

Figure 4.3: Example for line minimization

Gradient free algorithms are not that sensitive to discontinuous or rough objective functions as they do not use the local gradient in their calculations. Since most of them use a stochastic approach in finding the optimum, they are by far not as fast as gradient based methods. Depending on the randomly chosen variable values, these methods may never find the optimum. On the other hand, the chance of getting near a minimum increases with increasing number of iterations, hence one needs to run the algorithm for a certain amount of time to get a satisfactory result. Gradient free methods have the great advantage of independency from their initial point, since they use mostly randomly chosen variable values.

4.2.1 Monte Carlo Simulation

According to [6] the following definition of the Monte Carlo method can be made:

> "The Monte Carlo method is defined as representing the solution of a problem as a parameter of a hypothetical population, and using a random sequence of numbers to construct a sample of the population, from which statistical estimates of the parameter can be obtained."

A randomly generated set of parameters is used to generate a number of objective function values. This set may then be used for further statistical analysis or simply for finding the smallest function value. A simple example is shown in Fig. 4.4, a random set of points is created inside a rectangle. The number of all points p within the rectangle and all points p_i within the circle with radius r can be used to calculate π according to

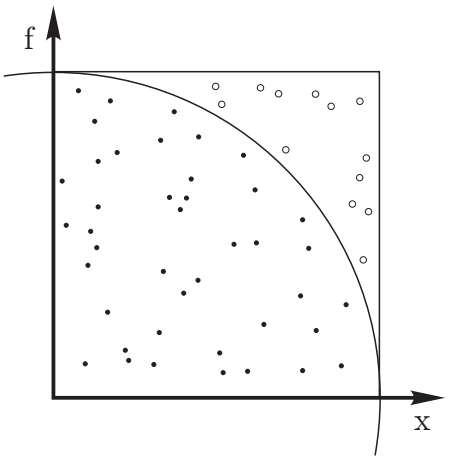

Figure 4.4: Monte Carlo Simulation

$$\lim_{p\to\infty} 4 \cdot \frac{p_\mathrm{i}}{p} = \pi \tag{4.3}$$

Using the law of large numbers, one can prove that with increasing number of points the solution of Eq. 4.3 converges to π.

4.2.2 Evolutionary Strategies

An evolutionary algorithm uses nature's concept of evolution to achieve the optimal solution for a problem. According to [10] the following iteration process describes an evolutionary algorithm:

1. Select an initial population randomly and perform function evaluations on these individuals

2. Perform selection for parents according to their relative fitness

3. Apply crossover and mutation to generated new individuals from the selected parents

    - Apply crossover with a fixed probability from two selected parents
    - If crossover is applied, apply mutation to the newly generated individual with a fixed probability

- If crossover is not applied, apply mutation with a fixed probability to a single selected parent

4. Perform function evaluations on the new individuals
5. Perform replacement according to their relative fitness to determine the new population
6. Return to step 2 and continue the algorithm until the convergence criteria are satisfied or the iteration limit is exceeded

At first, a set of random function evaluations, called a population, is generated (Fig. 4.5 (a)). This population is then evaluated according to its fitness, meaning the lowest values are the best ones (marked with an 'x'). The fittest individuals are then used to generate new populations near them (Fig. 4.5 (b)). This process is then continued until certain termination criteria are met. The number of function evaluations, a defined fitness or a certain amount of time can be used as a termination condition. The survivors at the end of the iteration represent the best function values according to the evolutionary strategy (Fig. 4.5 (c)).

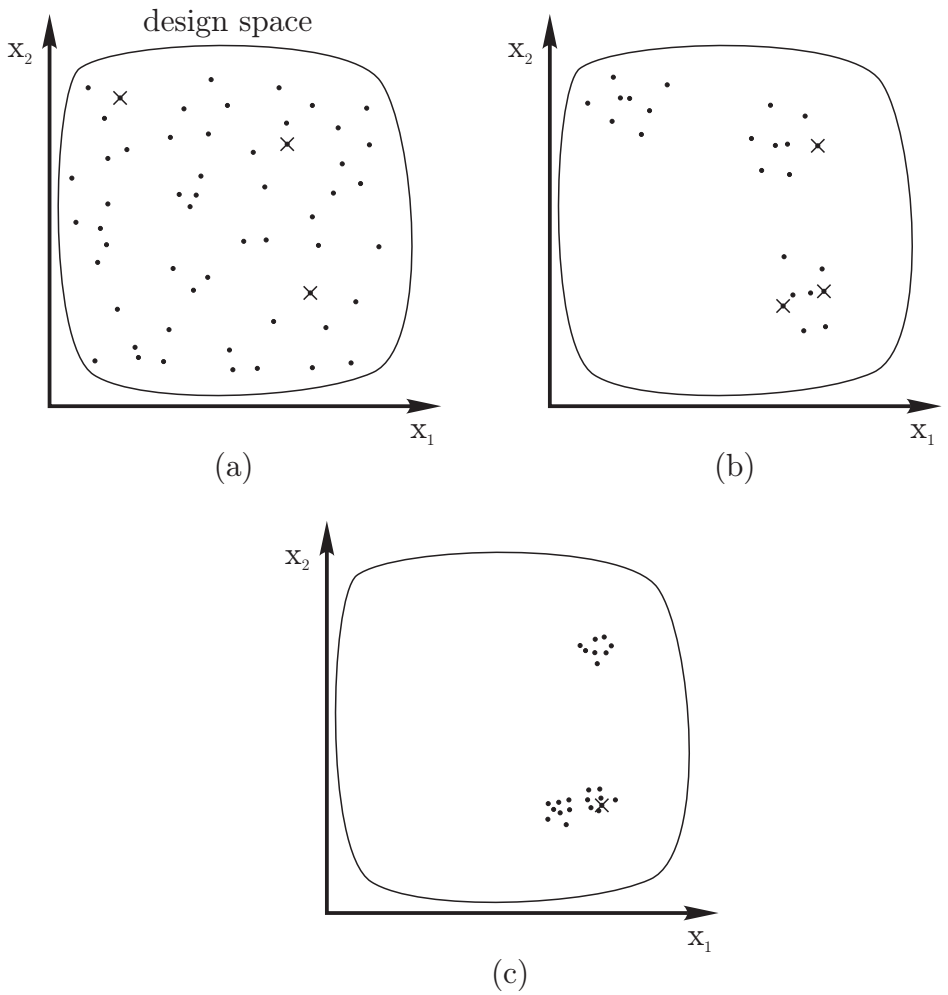

Figure 4.5: Evolutionary strategy

Chapter 5

Structural Optimization

Structural optimization has to be seen as a design tool. A short example should illustrate this: if one can manage to reduce the weight of a car in a way that the fuel consumption is reduced by only one percent, this would lead to enormous fuel savings. In Germany for example, calculated with an average kilometrage of about 15.000 km per year and an average consumption of 10 liters per 100 km, this would sum up to a reduction of 450.000.000 liters per year [18]. The potential for optimization is huge, but it is often not clear which way to pursue for finding it. So, the challenge for the engineer is to find out the changeable parameters and to define criteria to quantify the outcome. Two typical tasks for structural optimization might be as follows:

1. Minimize the weight of a structure without increasing the stresses or displacements above the critical threshold.

2. Maximize the lowest eigenfrequency without influencing the weight.

For every optimization, one needs a corresponding model to represent the optimization task and to abstract it into a mathematical relationship. If the task is very simple, an analytical model might be the best choice. When approaching more complex problems, the potential of analytical methods will certainly be exceeded and numerical methods will need to be applied. Several different numerical methods, e.g., the finite element method or the finite difference method, have been developed to assist the engineer in fulfilling his tasks.

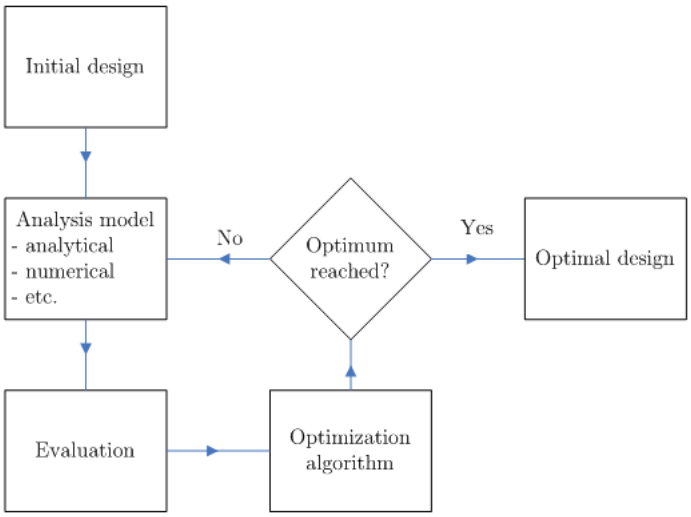

Figure 5.1: Flow chart of a typical optimization loop

When one has successfully defined methods to quantify the optimization result several more questions need to be answered:

- When is the optimal solution reached? Which objective function value needs to be reached to satisfy the needs?
- Which are the restrictions, e.g., which critical deformations are not to be exceeded?
- Which are the changeable parameters, and are these influencing the objective functions and restrictions?

After defining all the aims and parameters of an optimization, the schematic procedure of an optimization loop is described in Fig. 5.1. The initial values are used as a first input for the analysis model. This model is then evaluated and passed to the optimization algorithm, where the actual optimization takes place. After that, the criteria defined earlier are checked if the optimum has been reached. If so, the optimization task is finished; if not, the loop starts over from the beginning.

5.1 Classification of Structural Optimization Problems

According to [12], structural optimization problems can be classified by their type of design parameters and therefore by the strategy that needs to be applied to solve the problem. Fig. 5.2 illustrates the different optimization tasks.

- The choice of construction method, e.g. a solid girder, a carcass or a composite structure
- The choice of material, e.g., steel, aluminum, wood or composite materials
- Topology optimization: the design parameters define the arrangement of structural elements
- Shape optimization: the geometry of the structure is changed without influencing its topology
- Dimensioning: wall thicknesses and profiles are chosen

5.2 Construction Method

The main task in optimizing a structure is usually to optimize a specific objective function under certain restrictions. An example for this may be to minimize the weight of a structure without exceeding a critical stress threshold. In any case one needs an initial design, the first input for the optimization. These initial designs can influence the outcome of the optimization significantly. By following several basic construction principles, these initial structures can be designed in a way to improve the simulation result or at least shorten the simulation time.

Several methods for improving the initial design are as follows:

- Choice of material: utilize anisotropy and respect material–related manufacturing issues.

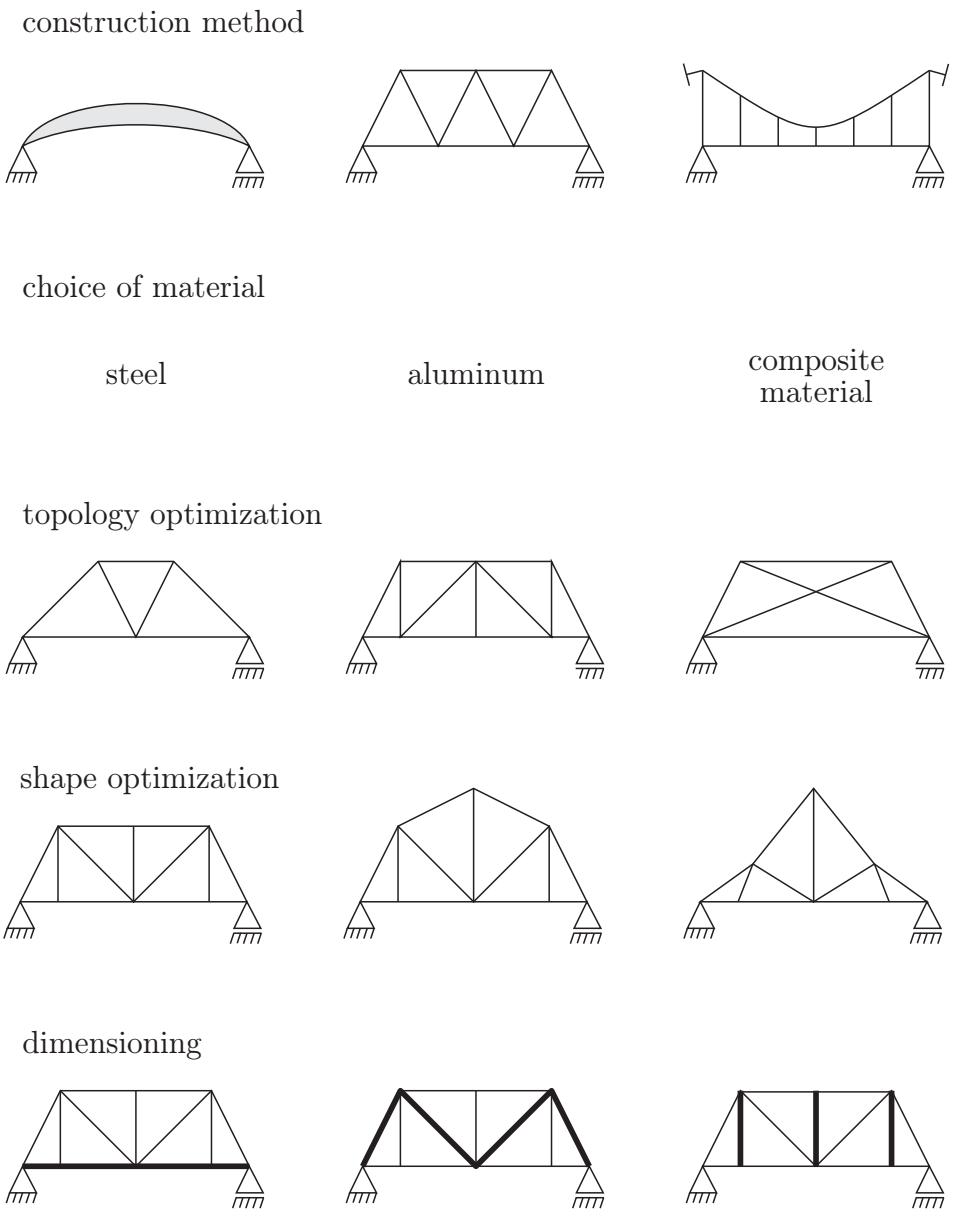

Figure 5.2: Classification of structural optimization

- Be sure to determine the loads and boundary conditions as accurately as possible; distinguish between static, cyclic and dynamic loads. If necessary, use Multi Body Dynamics to verify the measured results.
- Use multifunctional components, which perform several different tasks simultaneously, to reduce weight.
- Use the shortest possible levers to reduce bending stresses.
- Beware of buckling and warpage, especially with very thin structures.
- Used beads wherever possible in sheet or plate constructions.

Another problem in designing a component is its manufacturing. Optimization processes like, e.g., topology optimization can lead to abstract results that may not be fabricable. Cast components, for example, have to be designed in a way to allow their removal from the die. Components manufactured on a turning lathe need to be axisymmetric and milled parts need to be designed in a way that the cutting tool is able to reach everywhere necessary.

One should also take economical aspects into account. Depending on the batch size, the size of the component and several other demands, the manufacturing method may differ.

Fig. 5.3 shows a few examples for how to improve an initial design and thereby reduce bending stresses and weight and strenghten the structure.

5.3 Topology Optimization

The topology of a body describes how many voids it contains. The exact shape of the voids and their borders are not exactly defined. Topological properties are the most general properties of a body. Fig. 5.4 (a) shows two topologically equivalent bodies, they belong to the same topological class. Topological classes are distinguished by their degree of region connectivity, fig 5.4 (b) shows a simply, doubly and three times connected shape. (n–1) "cuts" are necessary to transform a n–connected region into a simply connected one,

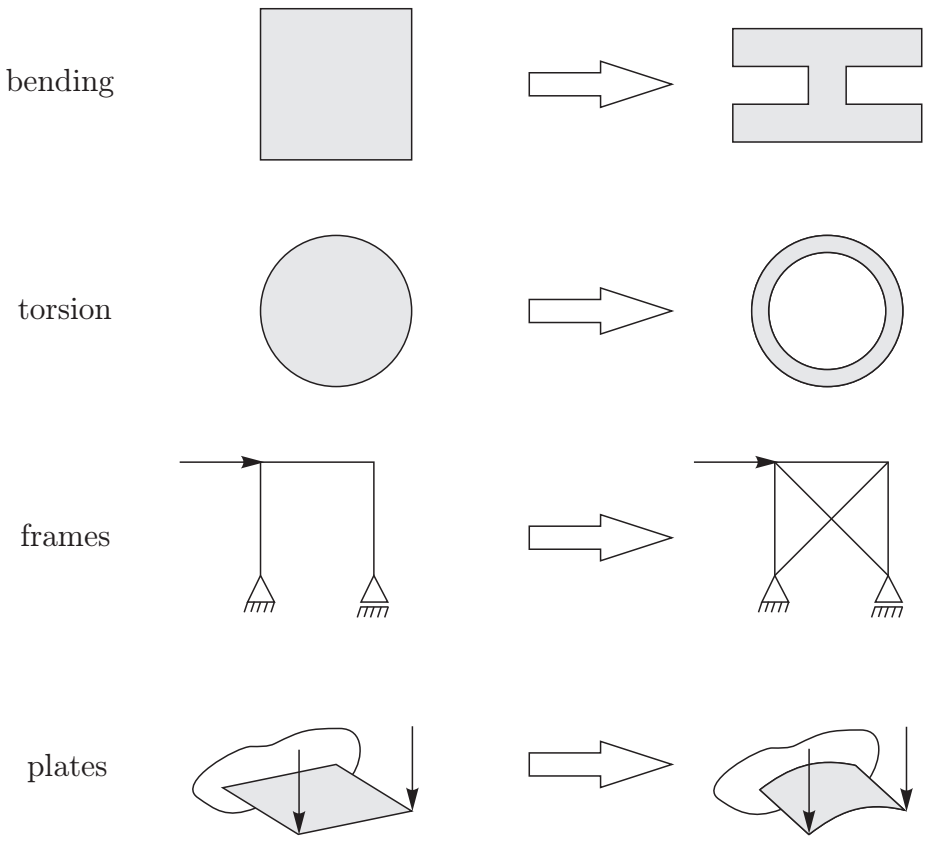

Figure 5.3: Examples for improving an initial design

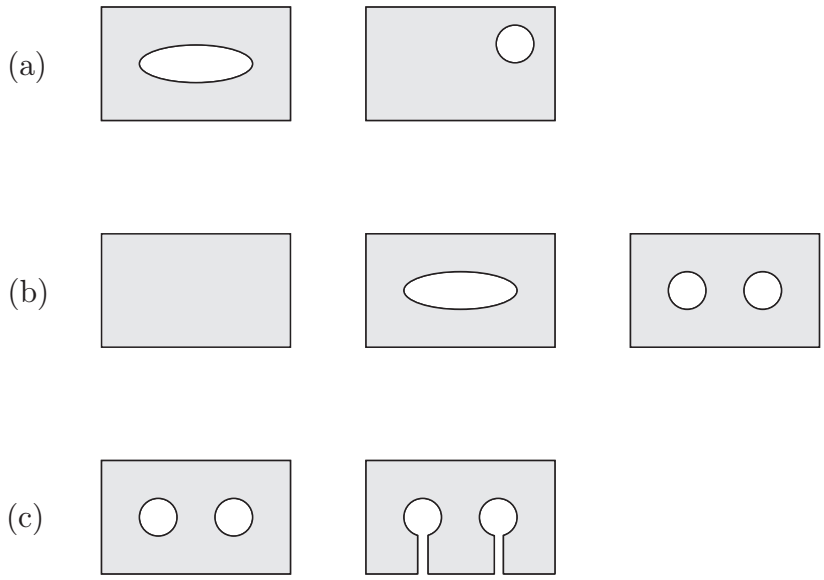

Figure 5.4: Topologically identical (a) and different (b, c) bodies

Fig. 5.4 (c) shows the transformation of a three times connected region into a simply connected one.

Topology optimization represents a very time efficient way for optimizing a structure in the early design stage. It is only necessary to define the design space, the fixed regions called "frozen elements" which are not to be removed, the position, direction and value of the forces and the boundary conditions, and the optimizer will find the best solution for the problem regarding a specific objective function. This provides a way to design completely new structures, without knowing a–priori what they might look like. Hence, topology optimization is a tool often used very early in a design process to create an initial design which is then used as an input for further optimization. A typical example for a topology optimization problem would be to minimize the weight while maximizing the first eigenfrequency.

Modern topology optimization methods can be categorized in different ways, described in the following sections:

- definition of the design space
- type of objective function

- type of design variable
- algorithm used

5.3.1 Definition of the Design Space

Topology optimization methods can be classified according to their definition of the design space. Two different approaches can be used:

- Methods for optimizing discrete problems use a space filled with points, which are then connected by as many rods in as many variations as possible. From this structure, the optimal rods are then chosen as the best ones.
- Continuous topology optimization does not need the design space described above, it only requires the definition of the design space with its — possibly very complex — boundary conditions. This space is then filled with finite elements which are then iteratively removed by the algorithm until the best solution is found.

5.3.2 Type of Objective Function

Most of the topology optimization methods use weight as their objective function,

$$f_{\mathrm{w}} = \int_V \rho \, \mathrm{d}V \tag{5.1}$$

where ρ represents the density.

Other possibilities are the strain energy

$$f_{\mathrm{E}} = \frac{1}{2} \int_V \sigma_{ij} \cdot \varepsilon_{ij} \, \mathrm{d}V \tag{5.2}$$

where σ_{ij}represents the stress tensor and ε_{ij} the strain tensor, or the mass moment of inertia

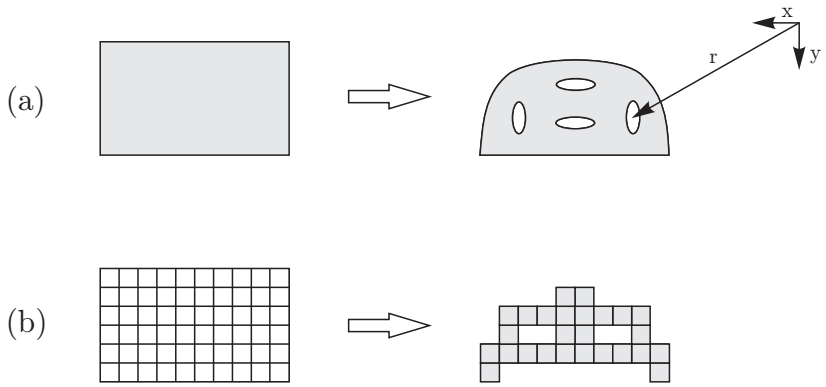

Figure 5.5: Different types of design variables

$$f_{\mathrm{M}} = \int_V \rho r^2 \, \mathrm{d}V \tag{5.3}$$

where r represents the distance to the axis of rotation.
Possible restrictions are, e.g.,

- stiffness
- lowest eigenfrequency
- durability
- etc.

5.3.3 Types of Design Variables

When using the method of parametrized boundaries, the number of design variables is very small (Fig. 5.5 (a)). The other method is to divide the design space into many different very small (finite) elements, each of them representing a changeable parameter. One can clearly see that this approach can soon lead to a high number of design variables (Fig. 5.5 (b)).

5.3.4 Algorithms

Different types of algorithms are used to solve the problem. All algorithms implemented in COLIN[1] [10], gradient based algorithms and evolutionary algorithms may be used.

5.3.5 Method of Homogenization

Since it is the most common method in topology optimization, the method of homogenization is now described in more detail as one example out of the variety of different approaches available.

The main idea behind this method is to divide the design space into finite spaces or elements. Each of these elements now represents a design variable. The goal of the algorithm is to vary the density of each of these elements in a way to satisfy all the restrictions and objective functions. [5]

An integer function $\chi(x_i)$ is used to describe the material distribution throughout the design space Ω_s. Its values can either be 1 or 0, representing the presence or absence of material. x_i represents the vector of design variables, each entry corresponds to one finite element. The mass density and stiffness vectors are thus represented by:

$$\begin{aligned} \varrho(x_i) &= \varrho_0 \cdot \chi(x_i) \\ C(x_i) &= C_0 \cdot \chi(x_i) \end{aligned} \tag{5.4}$$

with the integer function

$$\chi(x_i) = \begin{cases} 1 & \forall x_i \in \Omega_\mathrm{m} \\ 0 & \forall x_i \in \Omega_\mathrm{s} \setminus \Omega_\mathrm{m} \end{cases} \tag{5.5}$$

where Ω_m represents the set with high density ϱ_0 and stiffness C_0. With the strain energy as an exemplary objective function, the following functional has to be minimized

[1] Common Optimization Library INterface

$$f_{\mathrm{E}} = \frac{1}{2} \int_{\Omega_{\mathrm{s}}} \sigma_{ij} \cdot \varepsilon_{ij} \, \mathrm{d}\Omega \tag{5.6}$$

or, by inserting Hooke's law,

$$f_{\mathrm{E}} = \frac{1}{2} \int_{\Omega_{\mathrm{s}}} C_{ijkl}\left(\chi\right) \cdot \varepsilon_{ij} \cdot \varepsilon_{kl} \, \mathrm{d}\Omega \tag{5.7}$$

Solving this unrestricted problem would lead to the trivial solution of filling the whole design space with material. To prevent this, a constraint ensuring a certain target mass is applied.

To achieve the design which satisfies the equation stated above, the algorithm starts to iteratively reduce material in areas with low stresses to homogenize the stress in the remaining areas. This iteration continuous until certain abortion criteria are reached.

5.4 Shape Optimization

The shape of things is often a compromise between esthetic looks and mechanical requirements. These two demands can both be achieved at the same time, as shown in many structures of nature like, e.g., trees. Mattheck performed extensive studies on the shape of trees [13, 14]. He discovered that trees grow in a way to homogenize their stresses and therefore minimize them. He also discovered that trees strengthen themselves by developing denser and differently shaped structures when experiencing periodical loads such as from wind. The same applies if the tree gets somehow damaged: damage–induced notches are also repaired in a way to reduce the surface stress.

In contrast to topology optimization, shape optimization deals with changing the shape of objects while leaving their topology untouched. This is mainly done by altering the surface of a body; thereby it is possible to increase the lifetime by reducing the surface stresses. There is an almost infinite variety of methods for shape optimization, modern methods are almost always computer aided. One possibility is to change the coordinates of the

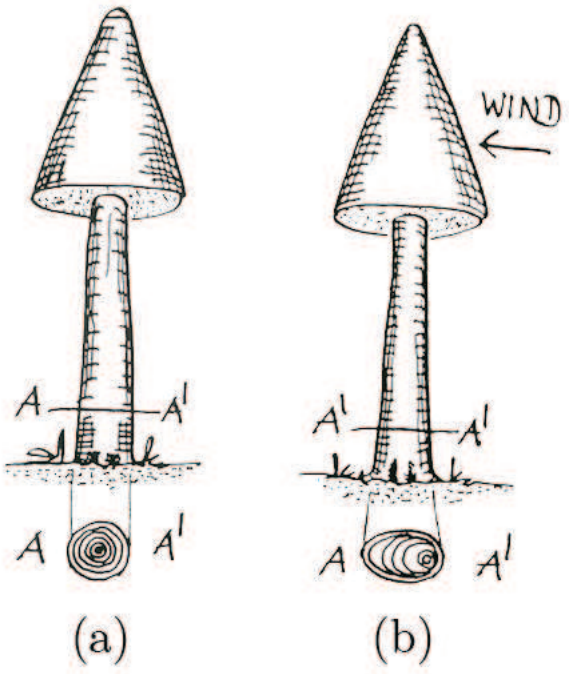

Figure 5.6: Trees changing their shape when experiencing periodical loads [13]

finite element nodes on the bodies' surface, these nodes are parametrized and their position is varied until the optimum is reached. Another possibility it to parametrize the curve by defining it via control points. The following curves are mainly used:

- Splines
- NURBS[2]
- Bézier curves

5.4.1 Law of Stress Decay

H. Neuber also performed studies on stress distributions. In 1958 he formulated his law of stress decay [15]:

[2]Non–Uniform Rational B–Spline

"Die bei allen Kerbproblemen auftretende starke Spannungsüberhöhung hat in der Umgebung der hochbeanspruchten Zone stets eine beträchtliche Abminderung der Spannung zur Folge. Je höher die Spannungsspitze ausgebildet ist, um so stärker erfolgt das Abklingen der Spannungen mit zunehmender Entfernung von der hochbeanspruchten Zone. Es handelt sich gewissermaßen um ein Reaktionsgesetz der Kerbwirkung."

This law states that every increase in stress due to notches leads to a decrease in stress in the notches' surroundings. The higher the stress peak, the faster the stress decay takes place in the notches' neighborhood.

5.4.2 Stress Homogeneity in the Variational Space

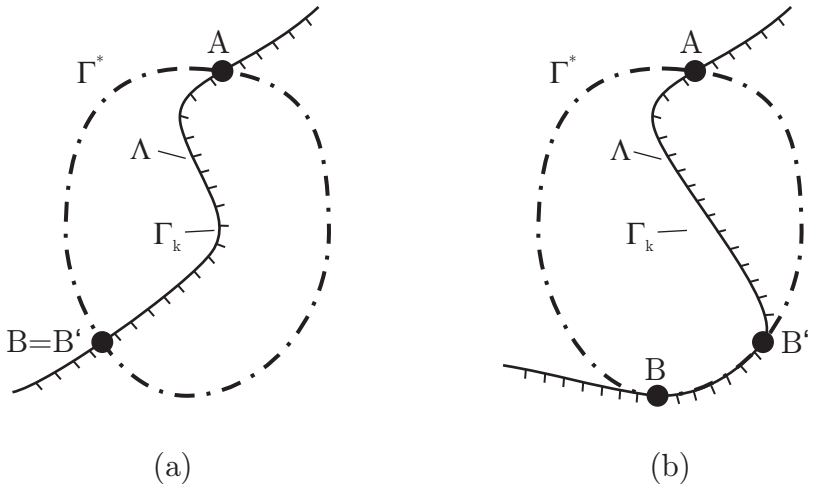

Figure 5.7: Stress homogeneity in the variational space

E. Schnack developed two hypotheses in 1978 regarding the stress homogeneity in the variational space [17]. If Γ_k represents the notch surface between point A and point B, Λ represents the part of the notch surface between point A and point B' not on the boundary of Γ^*, then the following two hypotheses can be made:

a) If there is a notch surface Γ_k within a defined region Γ^* between two fixed points A and B with constant tangential stress σ_t, then the resulting

notch stress is minimal (Fig. 5.7 (a)).

b) If there is a notch surface Γ_k within a defined region Γ^* between two fixed points A and B with constant tangential stress σ_t then the resulting notch stress is minimal if the interior segment Λ of Γ_k with constant $|\sigma_t|$ is maximum and the tangential stress$|\sigma_t|$ on the boundary segment $(\Gamma_k - \Lambda)$ is smaller than the constant stress on Λ (Fig. 5.7 (b)).

Chapter 6

Introduction to DAKOTA

> "The DAKOTA (Design Analysis Kit for Optimization and Terascale Applications) toolkit provides a flexible, extensible interface between analysis codes and iterative systems analysis methods." [3]

DAKOTA was developed by Sandia National Laboratories using an object–oriented approach; it was coded using C++ as a programming language. Originally, it was designed for Linux operating systems; a version for the UNIX API Cygwin is available to provide a possibility for running DAKOTA in a Microsoft Windows environment. Since the interface described in this thesis was developed in a Windows environment, the Cygwin version was used throughout. A generic interface is provided to ensure a flexible framework for designing an interface between DAKOTA and an arbitrary program. The following capabilities are included:

- Design of experiments
- Least squares methods
- Uncertainty quantification
- Parametric studies
- Optimization methods

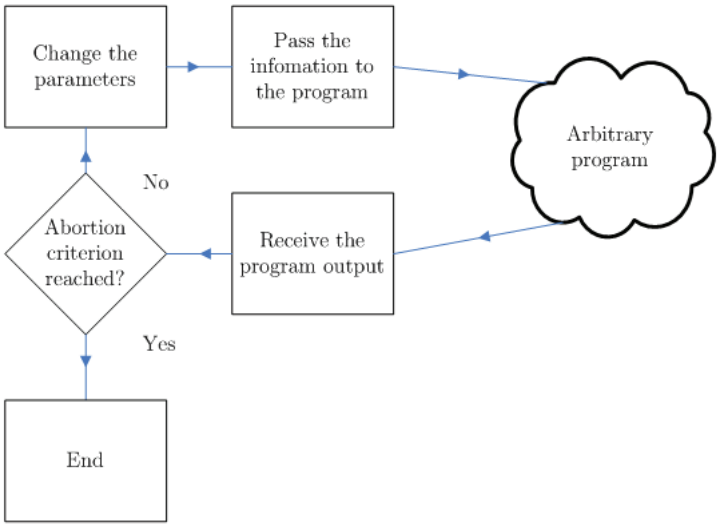

Figure 6.1: Overview of DAKOTA

- gradient based
- gradient free

Basically, DAKOTA takes the input variables from the user input, and provides them for the interface to the arbitrary program (in this case, Abaqus). Afterwards, when the program has finished its calculations, DAKOTA reads the program's output and runs its internal iterator (e.g., optimizer or a simple parametric study) to provide the new variable values for the next program call. Fig. 6.1 gives a basic overview of this process. As the cloud emphasizes, DAKOTA works as a black–box optimizer. It only knows the values of the parameters and objective functions but does not know their meaning or relation.

6.1 DAKOTA Input File

DAKOTA is controlled using a text input file. Fig. 6.2 shows an example of this input file. The file is divided into several groups:

- Strategy

- Method
- Model
- Variables
- Interface
- Responses

The *strategy* section controls DAKOTA's advanced meta–procedures, e.g., hybrid optimization, Pareto optimization or multi–start optimization. Furthermore, it specifies the graphical output and the tabular data output.

The *method* section specifies the iterative technique that DAKOTA will use. In the example in Fig. 6.2 the keyword *multidim_parameter_study* is used which specifies a multidimensional parameter study without any optimization. The range of values for both variables will be evenly divided into 5 partitions (6 data points) starting with the lower bound and ending with the upper bound. Other choices for the method section could be optimization methods or data sampling techniques.

In the *model* section, the model used by DAKOTA is specified. The term "model" is defined as follows:

> "A model provides the logical unit for determining how a set of variables is mapped into a set of responses in support of an iterative method." [9]

One can choose between a single interface, as done in the example in Fig. 6.2, or a more sophisticated multi–interface model.

The *variable* section specifies all the information needed for the parameters of the optimization. Variables can be either continuous (as in the example shown) or discrete, they can be classified as design variables, uncertain variables, or state variables. In the example in Fig. 6.2 there are two continuous variables labeled 'angle' and 'width'. Their lower bounds are 10 and 15, their upper bounds are 20 and 25, respectively.

In the *interface* section, the method of exchanging data with the analysis code is specified. This example shows a system call interface; more detailed information on the interface section is provided in section 6.2.

The *responses* section of the input file defines the data that will be returned to DAKOTA from the analysis code. Information about the objective function, constraints, gradients and Hessian matrix is provided. A single objective function, no gradients and no Hessian matrix are used in the present example.

6.2 DAKOTA Interfaces

Several options are provided for implementing an interface between the analysis code and the iterator. These choices are discussed in what follows. We provide only a small overview of the different approaches; for more detailed instructions the reader is referred to [9].

6.2.1 Direct Function

The direct function interface can be used for interfaces between simulations that are directly linked into the DAKOTA executable. This method creates the least overhead because there is no need for files since the information is passed directly within DAKOTA. Therefore this is the method of choice if one wants to run massively parallel simulations with multiple function calls. On the other hand, this is also the interface which takes the most effort to create, since it is necessary to implement the analysis code into a library with a subroutine interface. The following exemplary code shows the definition of the direct interface (Fig 6.3).

6.2.2 System Call Interface

The system call approach includes an analysis code by calling it via the system function from the standard C library [7]. This call then creates a new

```
# DAKOTA INPUT FILE
strategy,
  single_method
    graphics,tabular_graphics_data

method,
  multidim_parameter_study
    partitions 5 5

model,
  single

variables,
  continuous_design = 2
    lower_bounds    10      15
    upper_bounds    20      25
    descriptors 'angle' 'width'

interface,
  system
    asynchronous evaluation_concurrency = 1
    analysis_driver = '/cygdrive/.../Python/ipo.bat'
    parameters_file = 'params.in'
    results_file    = 'results.out'
    file_tag
    file_save

responses,
  num_objective_functions = 1
  no_gradients
  no_hessians
```

Figure 6.2: Example of a DAKOTA input file

```
interface
  direct
    analysis_driver = 'rosenbrock'
```

Figure 6.3: Example code for the direct interface

```
interface,
  system
    analysis_driver = 'text_book'
    parameters_file = 'text_book.in'
    results_file = 'text_book.out'
```

Figure 6.4: Example code for the system call interface

```
                        2 variables
   1.000000000000000e+01 x
   2.000000000000000e+00 y
                        1 functions
                        1 ASV_1
                        2 derivative_variables
                        1 DVV_1
                        2 DVV_2
                        0 analysis_components
```

Figure 6.5: Example for a parameter file

process which runs the simulation code. Communication between DAKOTA and the analysis code is handled via basic file I/O[1]. An input and an output file are specified and all required information is transferred through these files. This approach creates much more overhead and more processes then the direct interface. On the other hand it is much easier to implement because there is no need to become acquainted with the DAKOTA source code. One only needs to implement a simple file I/O operation to complete the task. This method is most commonly used because of its simplicity. An example for a system call interface is shown in Fig. 6.4.

This interface was also chosen for the application developed in this thesis. The increase in overhead does not slow down the iteration significantly because the finite element simulation is much more time consuming. This makes the system call the most suitable interface for the present application.

Fig. 6.5 shows an example for a parameter file generated by DAKOTA which hands the parameter values to the simulation code. There are two variables defined in this example, one named 'x' with a value of 10 and one

[1]Input/Output

```
599.436279699 f1
```

Figure 6.6: Example for a result file

Integer Code	Binary Expression	Meaning
7	111	Get Hessian matrix, gradient and value
6	110	Get Hessian matrix and gradient
5	101	Get Hessian matrix and value
4	100	Get Hessian matrix
3	011	Get gradient and value
2	010	Get gradient
1	001	Get value
0	000	No data required

Table 6.1: Active set vector

named 'y' with a value of 2. One objective function has been defined.

ASV stands for 'Active Set Vector' which contains an integer describing all the possible combinations of value, gradient and Hessian matrix. The most significant bit corresponds to the Hessian matrix, the intermediate one to the gradient and the least significant one to the value of the objective function. Table 6.1 shows a list of valid values for the active set vector with their meaning. The ASV informs the simulation code which values need to be returned, of course the gradients and Hessian matrix can only be returned if they are analytically available.

The next line gives the number of derivative variables, in this case two, followed by the 'DVV_1' and 'DVV_2' representing the derivative variable identifiers. The final line provides the analysis components which are used to pass additional information to the simulation code if necessary.

```
interface,
  fork
    input_filter = 'test_3pc_if'
    output_filter = 'test_3pc_of'
    analysis_driver = 'test_3pc_ac'
    parameters_file = 'tb.in'
    results_file = 'tb.out'
    file_tag
```

Figure 6.7: Example code for the fork interface

6.2.3 Fork Interface

The fork simulation interface uses the Linux functions *fork*, *exec* and *wait* of the Linux fork function family to manage simulation codes and simulation drivers [1]. *Fork* and *vfork* are used to create a copy of the DAKOTA process, *execvp* replaces this copy with the simulation code and DAKOTA finally waits using *wait* or *waitpid* until the simulation code has finished. An example for a DAKOTA input file using the fork interface is given in Fig. 6.7.

Chapter 7

Interface for Parametric Optimization (IPO)

As has been mentioned in the introductory remarks, the aim of this thesis is to develop a flexible program interface between the finite element solver Abaqus and the open source optimization library DAKOTA. With this interface, one should be able to parametrically optimize a component and also to use all capabilities of DAKOTA available. The Abaqus Python API[1] serves as an easy–to–use basis for the coding, since all Abaqus pre– and postprocessing commands are available in this API. An object–oriented approach fits best into the existing API and allows an easy further extension of the interface. The interface combines the advantages of both software packages. One can use the finite element solver Abaqus, which is capable of solving highly nonlinear (material as well as geometric nonlinearities) engineering problems, and join it with the extensive optimization and parametric study capabilities of DAKOTA. The Abaqus Python API ensures a flexible program architecture and an easy interface for ongoing extensions.

[1] For more information on the Abaqus Python API please refer to [21]

7.1 Object Structure

As mentioned above, an object–oriented programming approach is used for implementing the interface. The following section will focus on describing the principles of object–oriented programming and the object data structure used for linking the two programs.

Object–oriented programming uses "objects" to store data in a more efficient and organized way. Abstract objects include data fields and methods for manipulating them. The main idea behind this is to encapsulate the data from different objects to avoid accidental manipulation of these data sets. Classes are used to define abstract things (objects), their properties (variables) and their capabilities (methods). Instances of these classes, called objects, are then created to store all the data needed. A simple example for a class is a bank account. It has certain properties, e.g. owner, bank corporation, amount of money on it etc., and different capabilities like, e.g., adding or transferring money. When creating a new bank account, the new object inherits all the variables and methods from the original class.

Fig. 7.1 shows the structure used for the interface. The following classes are used:

- LOG
- PARAMETER
- OBJECTIVEFUNCTION
- CONSTRAINT
- VARIABLE
- VARIABLEKEY
- IPO

The *LOG* class is used for creating and editing the log file. All actions taken by the interface are stored in this file. This proves very useful for debugging when encountering miscellaneous errors.

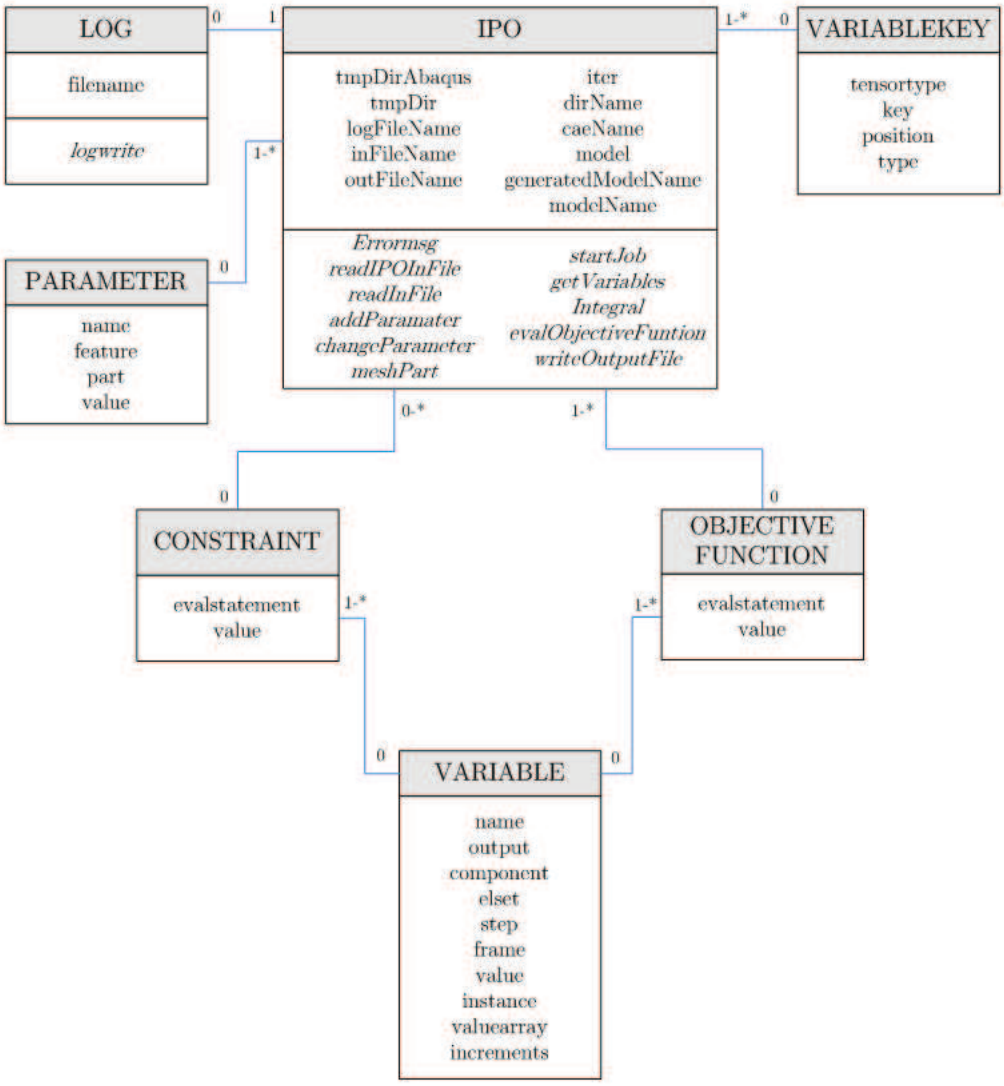

Figure 7.1: Object structure of the IPO

Data fields are written beneath the class name, methods are written in italics. The lines represent the connectivity of the classes. The numbers stand for the amount of instances of the class on the other end of the line which are included in the class right next to the number.

The *PARAMETER* class stores all information needed to identify a parameter. Parameters are used for storing information about the changeable values of the model, e.g. diameter, height or width. These parameters are then changed during the simulation to provide different objective function results.

Output information gained after the simulation is stored in the *OBJECTIVEFUNCTION* and *CONSTRAINT* classes. The user specified objective function or constraint is gathered when the Abaqus simulation is finished and stored in objects of these classes.

VARIABLES are combined into objective functions and constraints. This is necessary since one objective function or constraint can consist of several different variables. One variable represents one output from Abaqus, e.g. strain, stress or displacement.

The *VARIABLEKEY* class is used internally for temporally storing information about the key indexes used by Abaqus. It sorts the data in a way to be easy accessible by the interface.

The *IPO* (Interface for Parametric Optimization) class is used as a basis for all the other classes. It contains all the other classes and several methods for controlling them. All the actions and changes made by the interface are controlled from here. A detailed description of the operations will be given in section 7.3.

7.2 IPO External Workflow

Fig. 7.2 is used to illustrate the general workflow of the interface for parametric optimization. As stated before, the UNIX API Cygwin is used to run DAKOTA in a Microsoft Windows environment. Hence, DAKOTA is started from the Cygwin shell at first. After reading the DAKOTA input file, a Windows batch file is launched through the DAKOTA system call interface described in section 6.2.2. The Windows batch file then creates a new folder in order to keep all the files created by Abaqus organized, changes the working directory to the newly created one and starts Abaqus with the option -nogui and the appropriate input and output files used by DAKOTA.

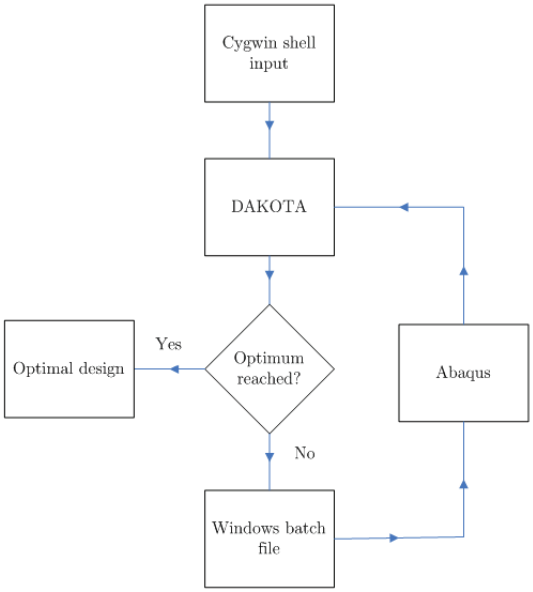

Figure 7.2: IPO workflow

```
@echo off
if exist abaqus_%1 goto next
mkdir abaqus_%1
:next
cd abaqus_%1
abq671 cae nogui="D:\...\ipo.py" -- %1 %2
```

Figure 7.3: Windows batch file

This option starts Abaqus without the GUI[2] and executes the IPO Python script. The Windows batch file is shown in Fig. 7.3. The Python script then executes the main program of the interface described in detail in section 7.3. After the script is finished, Abaqus is closed and DAKOTA continues with the next iteration.

[2]Graphical User Interface

7.3 IPO Internal Workflow

After executing Abaqus CAE with the -nogui option, the Python routine is started. Fig. 7.4 expands Fig. 7.2 with the internal Python workflow.

7.3.1 Reading the Input Files

At first, the Python script reads the parameter input file provided by DAKOTA (see section 6.2.2). Then it reads an additional input file required for the interface, the IPO input file. This file provides information about the Abaqus calculation and the specified objective functions. Fig. 7.5 shows an exemplary IPO input file.

```
cae_name = v1.cae
model_name = v1
mesh_size = 0.075
mesh_factor = 0.1
num_cpus = 1
pre_memory = 1024
standard_memory = 1024

variable
  name = pressure
  step = apply_pressure
  frame = -1
  output = CPRESS
  # component = S11
  instance = p32-2
  # element_set = tip

objective_function
  value = average(absolute(pressure))
```

Figure 7.5: IPO input file

The first block provides information needed for the Abaqus calculation. *cae_name* represents the name of the simulation file, *model_name* the name of the Abaqus model, *mesh_size* the seed size for remeshing the

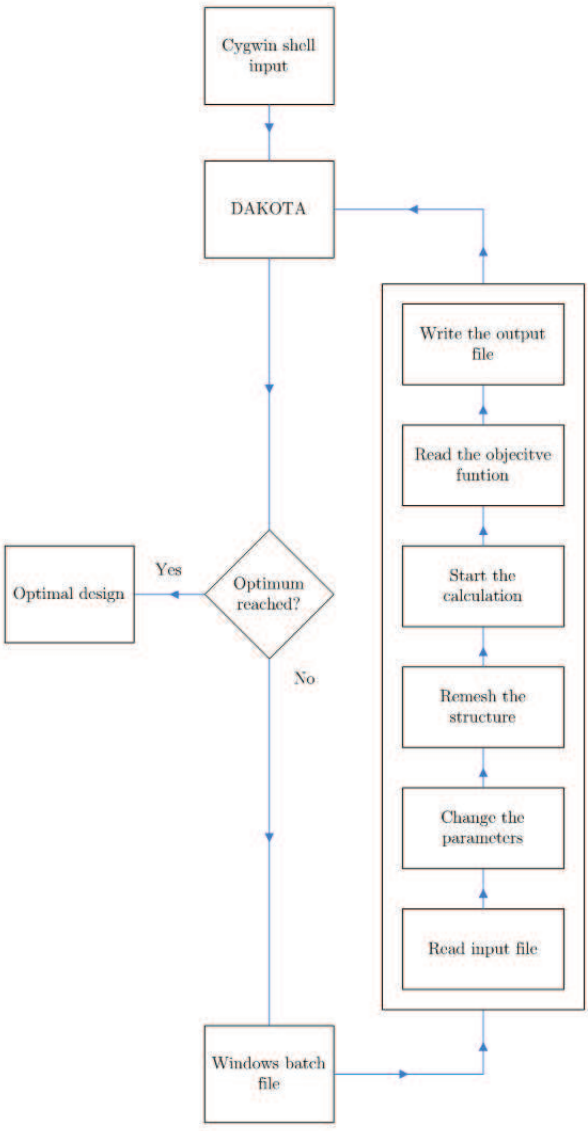

Figure 7.4: IPO flow diagram with internal workflow

part, *mesh_factor* the mesh deviation factor, *num_cpus* the number of central processing units used for the simulation and *pre_memory* and *standard_memory* the amount of memory that need to be allocated for the simulation.

Beneath that, the variable section follows. As stated in section 7.1, the interface uses variables to combine them into objective functions. Each variable represents a field output provided by Abaqus. All the information needed for the definition of these field outputs is given in this section. The *name* represents an unique identifier for internal variable handling. *Step* and *frame* define the analysis time at which to take the output, the integer value for *frame* can also be negative if one wants to define the time starting from the last frame. The field output is defined by the keyword *output*; for more information on the available keywords please refer to [20]. If the field output consists of multiple values, e.g., the stress tensor, further definition of the required output can be made with the *component* keyword. In this example the *component* keyword is not needed, since the field output request CPRESS is a scalar value; therefore this line has been commented using the hash key. With the *instance* keyword one can specify the part instance from which to take the field output; it is also possible to define a specific element set by using the *element_set* keyword. When defining the element set within a part instance, both keywords (*instance* and *element_set*) are required; when defining the element set within the assembly, only the *element_set* keyword is required.

Objective functions are then defined using the *objective_function* keyword. Those can consist of several variables; variable names are used as placeholders for their respective values. This is done using vector operations provided by the Python library 'numeric' [4]. The *value* string is evaluated in Python using the exec routine, this allows a very flexible calculation of the objective functions and constraints. One can define an arbitrary objective function by mathematical combination of any output variable provided by Abaqus.

In this example, the absolute value of the pressure is averaged throughout the instance p32–2.

```
objective_function
value = max(integral(absolute(disp)*absolute(mises)))
```

Figure 7.6: A more sophisticated example for an objective function

$$average \left| \begin{array}{c} f_1 \\ f_2 \\ \vdots \\ f_n \end{array} \right| = \frac{1}{n} \sum_{i=1}^{n} |f_i| \tag{7.1}$$

Eq. 7.1 illustrates the example given in Fig. 7.5, the vector values $f_1 \dots f_n$ represent the nodal values for the output variable CPRESS with n being the number of nodes.

A more sophisticated example is given in Fig. 7.6; Eq. 7.2 describes the mathematical evaluation. The integral over time is evaluated for each node, the maximum value serves as an objective function. This integral function is not included in the Python numeric library, it has been coded separately within the IPO.

$$\max_n \left(\int |u_n| \cdot |\sigma_{Mises_n}| \, \mathrm{d}t \right) \tag{7.2}$$

7.3.2 Changing the Parameters

After reading the input files, the script continues to find and change the specified parameters. Abaqus provides a possibility to label certain dimensions with a name; this option is used to identify the dimension that is supposed to change.

Fig. 7.7 shows the Abaqus parameter manager and a sketch with the dimension labeled 'y' marked in red.

All the information gained from the input files is used to find the correct parameter, if the parameter is not found an error message will be written

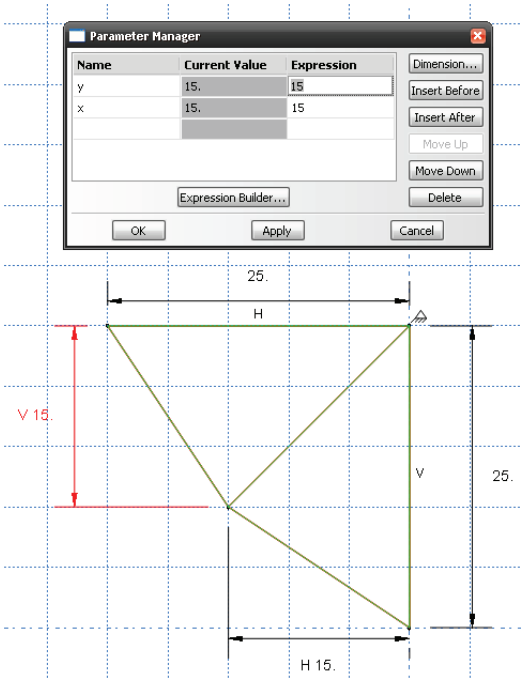

Figure 7.7: Abaqus parameter manager

to the log file. The parameter name has to be unique throughout the whole Abaqus model in order to be identified.

One has to make sure that the sketch is dimensioned correctly. Over– or underdetermining the sketch will lead to incorrect model regeneration and result in an error. When underdeterminig the sketch Abaqus may also change unintended other dimensions when updating the parameter.

7.3.3 Remeshing the Structure

Right after updating the parameters to their respective values the structure is remeshed. The values specified in the IPO input file (*mesh_size* and *mesh_factor*) are used to remesh the part. The meshing strategy remains the same as defined earlier by the user in the model. Fig. 7.8 shows an example for the mesh controls; here, quad–dominated elements with the advancing front algorithm are used.

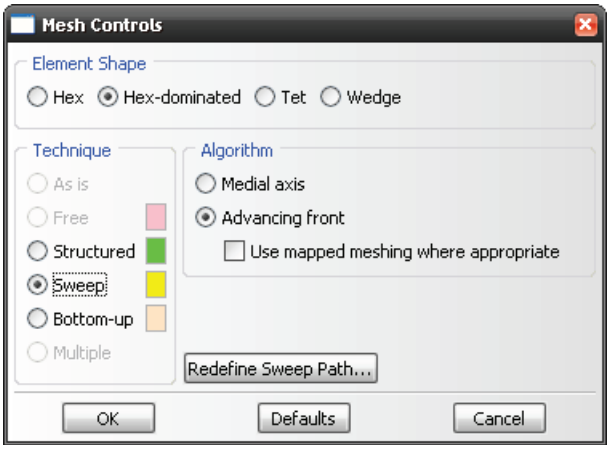

Figure 7.8: Mesh control in Abaqus

7.3.4 Starting the Simulation

The next step is to submit the model for simulation. The parameters defined in die IPO input file (*num_cpus*, *pre_memory* and *standard_memory*) are used to define the properties of the job. After submitting the job, the script waits until the simulation is completed.

7.3.5 Reading the Objective Function

After the simulation has successfully finished, the Abaqus ODB file[3] is opened to evaluate the objective function and the optional restrictions. The objective functions and restrictions are then calculated as described in section 7.3.1. All variables used in one objective function or constraint have to be from the same element set and the same step in order to ensure proper execution of the vector operations. If not done correctly this may otherwise lead to faulty matrix dimensions and result in an error.

7.3.6 Writing the Output File

At last all the data gathered for the objective function and the constraints are written to the result output file. Abaqus now closes and the data is

[3]Output Database file

passed back to DAKOTA which continues its iteration from the beginning.

7.4 Restrictions

The interface for parametric optimization only runs under certain conditions. The restrictions are as follows:

- It was developed and tested with Abaqus 6.7–1. There is no guarantee that the program will run under any other version of Abaqus, especially not with an older one. Further testing and coding would be needed in order to get IPO working with newer versions of Abaqus.
- Only the Cygwin version of DAKOTA was tested with the interface. For use of other DAKOTA versions the input files, more precisely the application paths need to be modified.
- The operating system used was Windows XP Professional 32 bit. For the use with other operating systems the paths and possibly more sophisticated settings need to be changed.

Chapter 8

Example Simulations

The following chapter illustrates the capabilities of the Interface for Parametric Optimization (IPO) by explaining certain representative examples.

8.1 Simple Truss Construction

This example was taken from [8] which treats optimization with evolutionary algorithms and uses this example to illustrate the capabilities of these methods. A simple truss construction consisting of 4 nodes and 5 trusses is shown in Fig. 8.1. It features two bearings, one being a fixed bearing and the other one being a floating one. The force F is applied at the top left corner where trusses number 1 and 2 meet. The bottom left node is parametrized by the variables x and y and therefore represents the moveable node.

This simple truss construction example was chosen to illustrate the general capabilities of the IPO. It may look trivial at first sight, but will lead to surprising insights in the following sections. Several approaches with different optimization algorithms are used to demonstrate the varying outcomes of these methods.

The aim of the optimization is to find the optimal values for x and y such that the weight of the structure is minimized. Eq. 8.1 shows the formula used for computing the objective function.

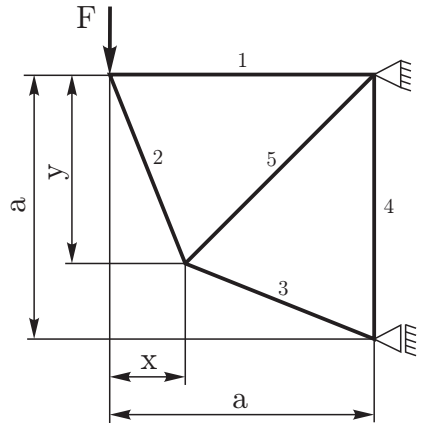

Figure 8.1: Simple truss construction

$$f = \rho \sum_{i=1}^{n} A_i \cdot l_i \tag{8.1}$$

Where ρ represents the density, n the number of beams, 5 in this case, A_i the cross sectional area, l_i the length of the individual beams. For simplification purposes the constant material density ρ was chosen to be $1\,kg/mm^3$.

To determine the individual cross sectional area of the beams, the forces acting in them are calculated and afterwards divided by the nominal design stress $\sigma_{\max}$. This results in the minimal allowable area for each beam, but does not consider the possibility of failure due to buckling.

$$A_i = \frac{|F_i|}{\sigma_{\max}} \tag{8.2}$$

$\sigma_{\max}$ was also set to $1\,MPa$ because of simplification reasons.

The width and height of the truss a where chosen to be $25\,mm$.

8.1.1 Finite Element Model

The truss construction given in Fig. 8.1 was implemented in Abaqus using a two–dimensional wire part. Abaqus provides two possibilities for characterizing the deformation behavior of wire structures:

- beam

- truss

Beams can absorb axial and bending stresses, while trusses can only transfer axial stresses. Trusses were chosen in this example, according to the analytical model (see section 8.1.2).

A linear elastic material formulation was used to describe the deformation behavior of the truss construction, the parameters for this model are summarized in Tab. 8.1.

parameter	value	unit
Young's modulus	210000	MPa
Poisson's ratio	0.3	–

Table 8.1: Linear elastic material constants

A force F of 1 Newton was applied as shown in Fig. 8.1. To be able to compare the results of the finite element simulation with the analytical calculation, the amount of the force was chosen very small compared to the material properties. This is necessary to keep the deformations to a minimum, since the analytical calculation was done using a first order approach which does not take the deformation of the nodes and beams into account. This approach is suitable for most such engineering constructions, because massive deformations of the structure will definitely lead to malfunction.

Truss sections in Abaqus need to have a cross sectional area; for simplification purposes this area A_0 was chosen to be $1\,mm^2$. With this area and the resulting stress σ_i in each beam it is possible to calculate the acting force F_i according to

$$F_i = A_0 \cdot \sigma_i \tag{8.3}$$

In order to calculate the objective function described in Eq. 8.1 the following equations need to be solved

$$f_i = \rho \cdot A_i \cdot l_i \tag{8.4}$$

using Eq. 8.2

$$f_i = \rho \cdot \frac{|F_i|}{\sigma_{max}} \cdot l_i \tag{8.5}$$

and Eq. 8.3

$$f_i = \rho \cdot \frac{A_0 \cdot |\sigma_i|}{\sigma_{max}} \cdot l_i \tag{8.6}$$

with the length l_i being

$$l_i = \frac{V_0}{A_0} \tag{8.7}$$

Eq. 8.6 becomes

$$f_i = \rho \cdot \frac{|\sigma_i|}{\sigma_{max}} \cdot V_0 \tag{8.8}$$

Since ρ and σ_{max} are 1 the above equation simplifies to

$$f_i = |\sigma_i| \cdot V_0 \tag{8.9}$$

This objective function is now easily applicable for the IPO (see section 7.3.1) because the acting stress σ_i and the original volume V_0 are available as field output in Abaqus. σ_i (stress) is represented by the field output *S11* and the volume V_0 (volume) is represented by the element volume *EVOL*. The resulting IPO input file is listed in Fig. 8.2.

The simplifications made within this process do not need to be made in order to run this simulation with the IPO, all are only made to simplify the example for this manner.

8.1.2 Model Verification

To verify the finite element model, the forces acting in the truss construction pictured in Fig. 8.1 were analytically calculated using Ritter's method, the

```
variable
  name = volume
  step = Step-1
  frame = -1
  output = EVOL
  instance = fachwerk-1

variable
  name = stress
  step = Step-1
  frame = -1
  output = S
  component = S11
  instance = fachwerk-1

objective_function
  value = sum(abs(volume*stress))
```

Figure 8.2: IPO input file for the simple truss construction

parameters where chosen to be $x = 10\,mm$ and $y = 15\,mm$. The stress results are illustrated in Tab. 8.2 and the volume results in Tab. 8.3.

As one can clearly see, the two calculation methods result in completely the same stresses and length of the beams (Table 8.2, 8.3). Therefore they both provide the same objective function value of 86.6668 kg.

8.1.3 Parametric Study

Because of the small amount of only two variables, a parametric study can be performed to get a general overview of the shape of the objective function. The study was run within the intervals

$$-25 \leq x \leq 50$$

and

$$-25 \leq y \leq 50$$

beam	analytical $[MPa]$	finite element $[MPa]$
1	0.666667	0,666667
2	-1.20185	-1.20185
3	-1.20185	-1.20185
4	0,666667	0.666667
5	0,471405	0.471405

Table 8.2: Comparison of the analytical and the finite element stress calculations

beam	analytical $[mm]$	finite element $[mm]$
1	25	25
2	18.0278	18.0278
3	18.0278	18.0278
4	25	25
5	21.2132	21.2132

Table 8.3: Comparison of the analytical and the finite element volume calculations

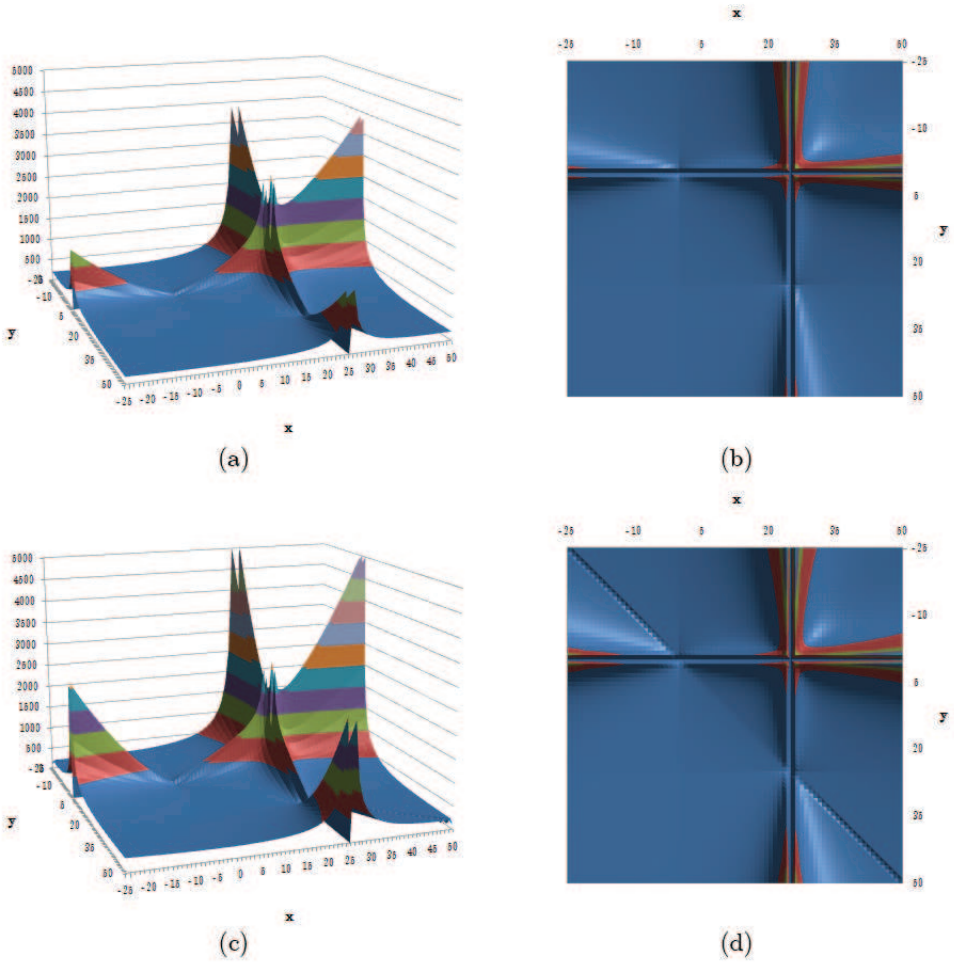

Figure 8.3: Parametric study of the objective function
(a) represents a three dimensional surface plot of the analytically calculated objective function according to Eq. 8.9, (b) shows a 2d surface plot of the analytically calculated objective function viewed from above. (c) and (d) display the corresponding finite element calculations.

minima	x $[mm^3]$	y $[mm^3]$	value $[kg]$
1.	8	18	82.876
2.	43	-18	482.889
3.	-1	-1	104.080
4.	-25	-25	99.999
5.	26	25	104.080
6.	50	50	99.999

Table 8.4: Local minima

Despite the seemingly simple problem, the behavior of the objective function is all but simple. Fig. 8.3 shows the shape of the objective function calculated using a parametric study. It is multimodal and has, besides one global minimum, several local minima. All local minima are listed in Tab. 8.4, the first one represents the global optimum. These values are not the exact values of the minima since the resolution of the parametric study was only 75×75 and therefore the objective function was only evaluated at integer values.

The difference between the analytical calculation (Fig. 8.3 (a) and (b)) and the finite element calculation (Fig. 8.3 (c) and (d)) is very small. The general shape is very similar, only the high peaks differ slightly from each other. This is because the structure is very distorted at these points and therefore the deformations are relatively large resulting in a difference between the analytical and the finite element calculation due to reasons described in section 8.1.1.

One can clearly see the scale of a seemingly simple looking structural optimization problem at this structure. This truss construction represents an excellent example for this manner because of its unique shape, several different outcomes of different algorithms will be discussed later.

```
method,
  coliny_ea
    max_iterations = 1000
    max_function_evaluations = 500
    population_size = 50
    fitness_type merit_function
    mutation_type offset_normal
    mutation_rate 1.0
    crossover_type two_point
    crossover_rate 0.0
    replacement_type chc = 10
```

Figure 8.4: DAKOTA input file for the evolutionary algorithm, first run

8.1.4 Optimization

In the following sections, different optimization algorithms will be applied to the truss construction described above in order to illustrate their individual strengths and weaknesses.

8.1.4.1 Evolutionary Algorithm

As a first approach an evolutionary algorithm was used to optimize the simple truss construction. The *coliny_ ea*[1] algorithm included in DAKOTA was the method of choice, for a more detailed documentation of the COLIN package the reader is referred to [10, 9].

Fig. 8.4 shows the method definition within the DAKOTA input file. A maximum of 1000 iterations with a maximum of 500 function evaluations was chosen. Each population contains of 50 individuals, the best 10 are chosen to survive during mutation.

The first run with the evolutionary optimizer leads to the solution illustrated in Fig. 8.5. This point represents the global minimum shown in Tab. 8.4, all differences are within the computational tolerance. Since the evolutionary strategy is a nondeterministic algorithm, its outcome depends on the number of function evaluations. Fig. 8.6 shows the evolution of the objective function during the first run. Since there are several local minima

[1]Common Optimization Library INterface = COLIN

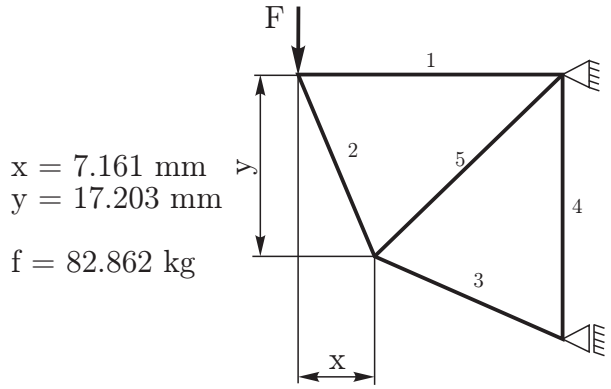

Figure 8.5: Solution found using the evolutionary algorithm, first run

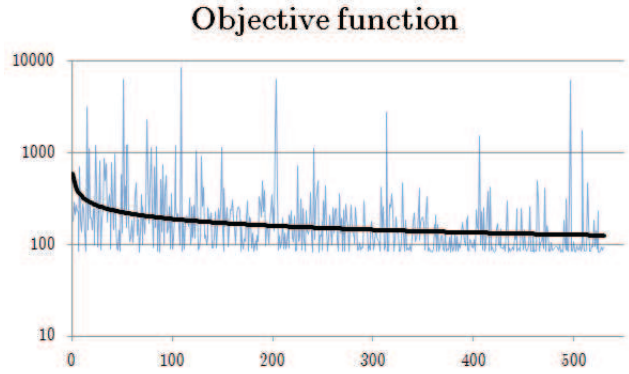

Figure 8.6: Evolution of the objective function, first run

the clear trend downwards is a little bit distorted. To improve the outcome of this simulation further, a second simulation was run using more function evaluations.

The second simulation was done using the parameters given in Fig. 8.7. This time, a maximum number of 5000 iterations with a maximum number of 2500 function evaluations was selected. Out of a population size of 80, 20 individuals survived each mutation.

The second run results in a similar outcome, the objective function differs only by 0.019 *kg* from the first run and is also in the range of tolerance. For the structure to be symmetric along beam number 5 the following equation needs to be fulfilled:

```
method,
  coliny_ea
    max_iterations = 5000
    max_function_evaluations = 2500
    population_size = 80
    fitness_type merit_function
    mutation_type offset_normal
    mutation_rate 1.0
    crossover_type two_point
    crossover_rate 0.0
    replacement_type chc = 20
```

Figure 8.7: DAKOTA input file for the evolutionary algorithm, second run

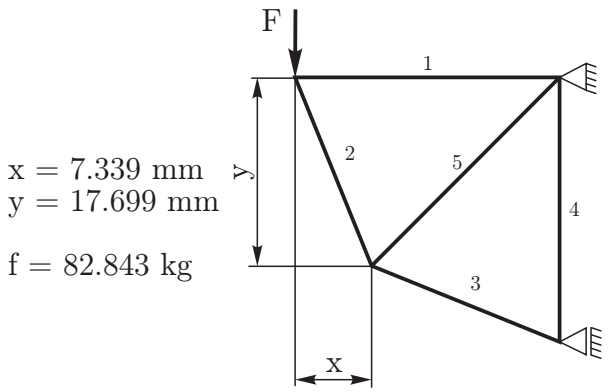

Figure 8.8: Solution found using the evolutionary algorithm, second run

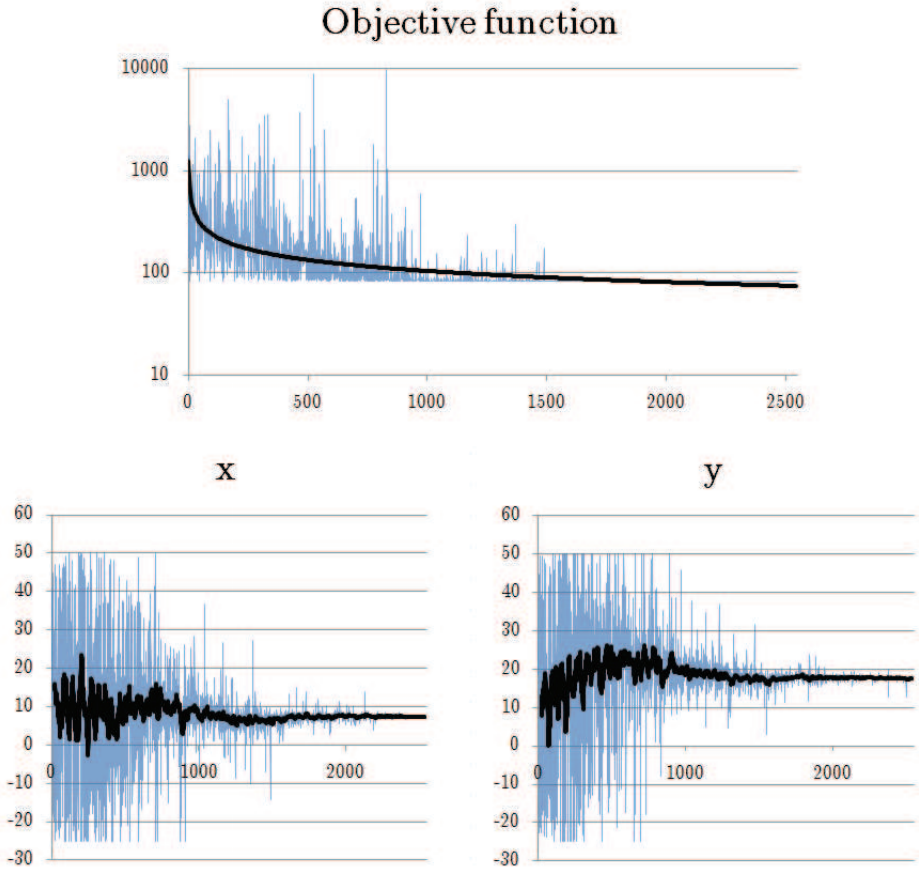

Figure 8.9: Evolution of the objective function, second run

$$x + y = 25 \tag{8.10}$$

The second simulation results in a smaller error regarding Eq. 8.10 and is therefore more symmetric. Hence the second attempt leads to a better objective function value, although the difference is very small.

Fig. 8.9 shows the evolution of the objective function and the respective parameter values during the second run. The descending trend has now become much more clearer due to the increased number of function evaluations. A graphical illustration version of the solution is depicted in Fig. 8.8.

One can easily derive the advantages of evolutionary algorithms from this example: they find the global optimum even if the objective function

```
method,
  conmin_frcg
    max_iterations = 1000
    convergence_tolerance = 5e-4
    \vdots
responses,
  num_objective_functions = 1
  numerical_gradients
    method_source dakota
    interval_type forward
    fd_gradient_step_size = 1.e-2
  no_hessians
```

Figure 8.10: DAKOTA input file for the gradient based algorithm

is very complex and contains several local minima. On the other hand, an evolutionary algorithm needs a certain amount of function evaluations in order to converge to this minimum. In this case, the limitation to 500 function evaluations was not enough to converge in the first place; an increase to 2500 was found to be adequate.

8.1.4.2 Gradient Based Algorithm

The second algorithm applied to this example was the Fletcher–Reeves conjugate gradient algorithm implemented as *conmin_frgc* in the DAKOTA CONMIN (constrained minimization) package [22, 9]. Fig. 8.10 shows parts of the input file used for this simulation. A maximum number of 1000 iterations with a convergence tolerance of $5 \cdot 10^{-4}$ was used. Since the gradient is not analytically available, DAKOTA calculates it internally using the forward difference method and a step size of $1 \cdot 10^{-2}$.

To demonstrate the influence of the initial values, the parameter interval was divided into nine equally spaced parts. Nine different simulations with their initial points at the center of each of these intervals where run. All initial values and their respective outcomes are listed in Tab. 8.5. The best values where gained by simulation number 7 with the final values $x = 7.313\,mm$ and $y = 17.686\,mm$. All solutions are graphically illustrated in Fig. 8.11.

	initial		final		
	x $[mm]$	y $[mm]$	x $[mm]$	y $[mm]$	value $[kg]$
1.	-13	-12	-1.955	-1.953	107.845
2.	12	-12	-3.921	-3.906	111.853
3.	37	-12	42.537	-17.537	482.852
4.	-13	13	7.015	17.588	82.851
5.	12	13	7.310	17.690	82.843
6.	37	13	28.908	28.925	115.899
7.	-13	38	7.313	17.686	82.842
8.	12	38	7.419	17.993	82.851
9.	37	38	26.780	26.780	107.123

Table 8.5: Initial points and outcomes for the gradient based algorithm

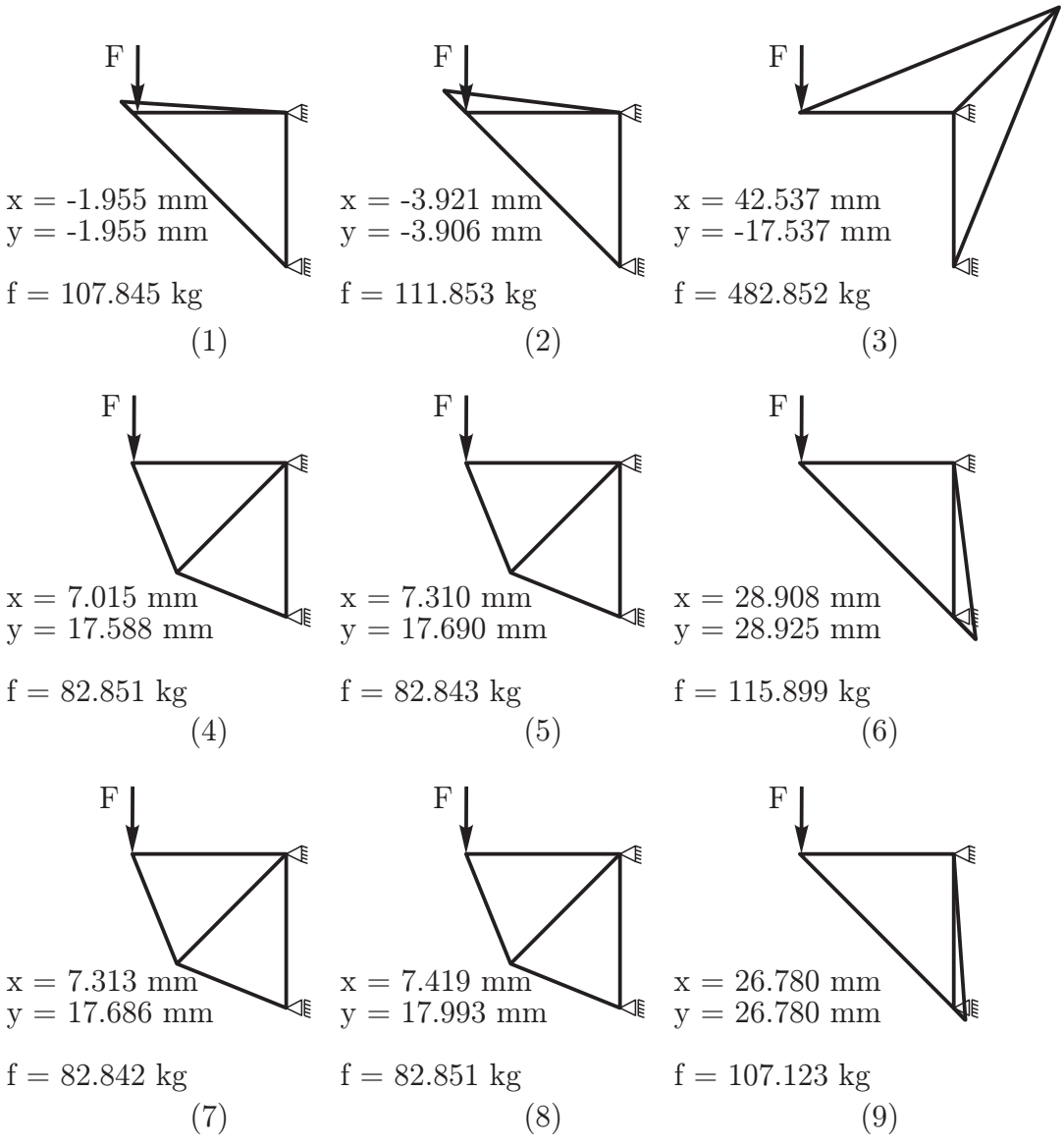

Figure 8.11: Solution of the gradient based optimizer

Only four out of nine initial points actually lead to the global minimum (simulations number 4, 5, 7 and 8), the other ones got stuck in a local minimum. This illustrates the main disadvantage of gradient based algorithms: they tend not to overcome local minima and therefore lead to incorrect results. On the other hand, also the main advantage of gradient based algorithms is highlighted: their enormous speed of convergence. On average it took the CONMIN algorithm only 22 iterations to converge to a minimum, compared to 500 and 2500 for the evolutionary algorithms, respectively. This saves a huge amount of simulation time.

Fig. 8.12 shows the path of the gradient based algorithm while descending downwards, triangles represent the initial point, squares the final ones.

8.1.4.3 Hybrid Algorithm

The DAKOTA hybrid algorithm provides a possibility to combine the advantages of both gradient based and gradient free algorithms. One can run several different algorithms one after the other and use the solution of the first one as an input for the following one. This is useful if the shape of the objective function is rough or if there are many different local minima. The evolutionary algorithm first scans the whole design space for the best evaluation, afterwards this point is used as a starting point for the gradient based algorithm which refines the solution using the first derivative of the objective function.

In this example, the algorithms investigated previously are combined. Fig. 8.13 shows parts of the input file. First the evolutionary algorithm runs 800 function evaluations, afterwards the gradient based algorithm refines the search. The solution found with the hybrid optimization is illustrated in Fig. 8.14.

Only a slight improvement is obtained, compared to the purely evolutionary strategy (Fig. 8.8),. However the simulation time could be reduced by almost 60 percent.

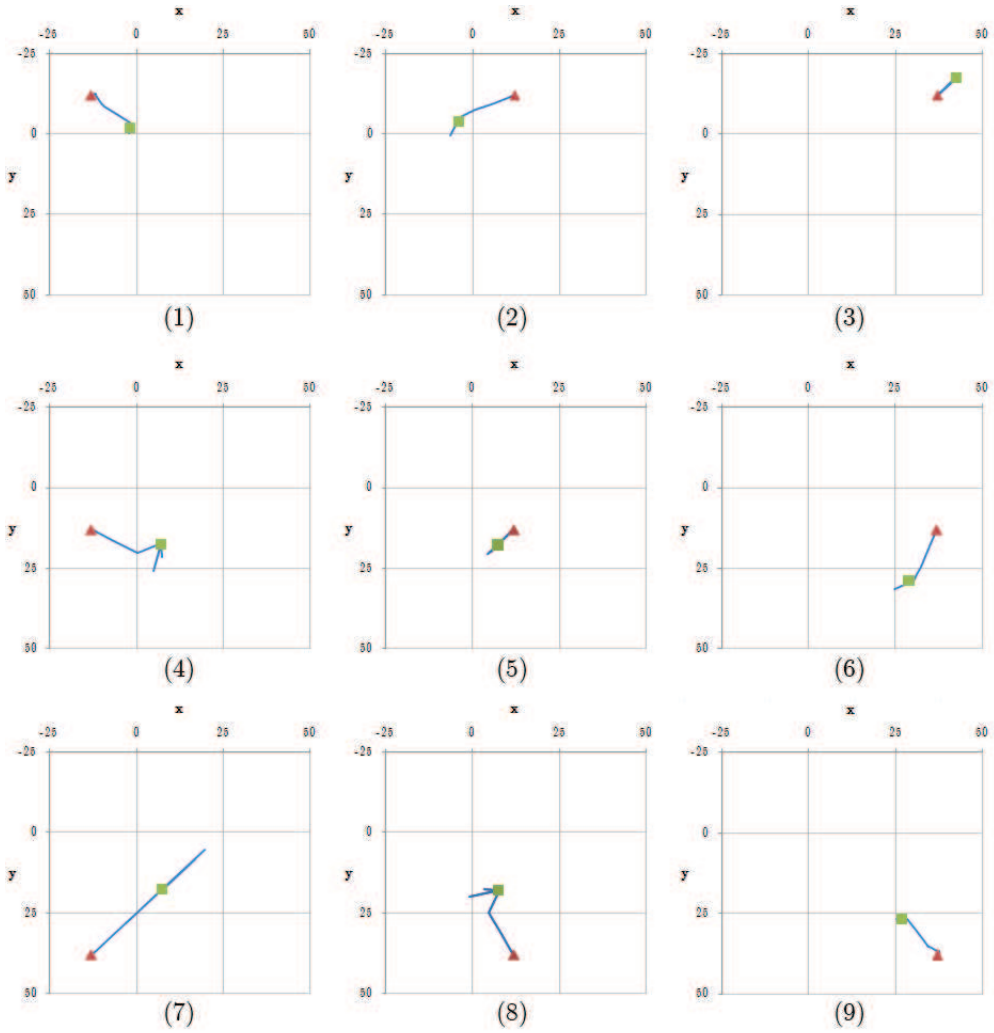

Figure 8.12: Path of the gradient based algorithm.
Triangles represent the initial points of the algorithm, squares the final solution of the gradient based approach.

```
method,
  id_method = 'EA'
  model_pointer = 'M1'
  coliny_ea
    max_iterations = 1000
    max_function_evaluations = 800
    population_size = 80
    fitness_type merit_function
    mutation_type offset_normal
    mutation_rate 1.0
    crossover_type two_point
    crossover_rate 0.0
    replacement_type chc = 20

method,
  id_method = 'GRADIENT'
  model_pointer = 'M2'
  conmin_frcg
    max_iterations = 1000
    convergence_tolerance = 5e-4
```

Figure 8.13: DAKOTA input file for the hybrid optimization

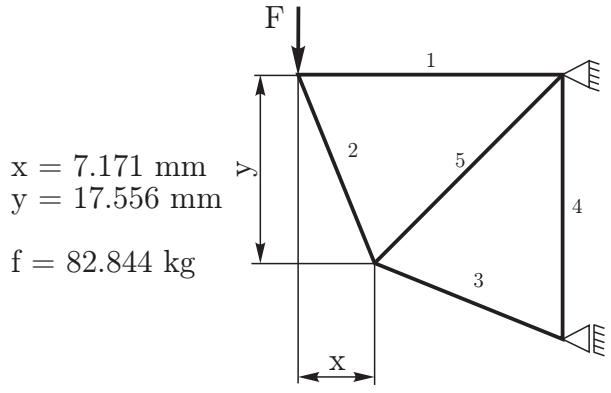

Figure 8.14: Solution of the hybrid optimizer

8.1.5 Discussion

The first run made with the evolutionary algorithm was clearly made with too few function evaluations, Fig. 8.6 shows no clear trend downwards. Therefore, a second run was necessary, this time using more function evaluations. This proves useful, since Fig. 8.9 shows a much more clear descending trend, thus indicating better convergence. The evolutionary algorithm always finds the global minimum; the quality of its outcome depends only on the number of function evaluations, not on the initial point. One can clearly see the great potential of this type of algorithm because of its ability of finding the global optimum even on a quite rough objective function (Fig. 8.3).

On the other hand, the gradient based algorithm *conmin_frcg* was not always able to find the global minimum, depending on the initial point the outcome was more or less suitable. Fig. 8.11 illustrates the different solutions, some of them are quite distorted. This demonstrates the main disadvantage of gradient based algorithms, one needs to choose a good initial guess for the algorithm to succeed. However, if the gradient based algorithm finds the global minimum, it does so in a very short amount of time.

Combining the advantages of both previously mentioned algorithms, the hybrid method serves of this application best. The capabilities of the evolutionary algorithms in finding the global minimum are joined with the high convergence speed of the gradient based ones. It was possible to reduce the simulation time by almost 60%, still leading to a very good result. This would be the algorithm of choice for optimization problems with highly rough objective functions and a reasonable number of parameters.

8.2 Bridge

The second example is a truss bridge construction, as illustrated in Fig. 8.15. It consists of 25 nodes connected by 50 trusses; 11 point loads are applied to the bottom girder of the structure. The coordinates of the top nodes represent the parameters, two degrees of freedom per node sum up to a total of 24 parameters. These parameters can be changed within the given range:

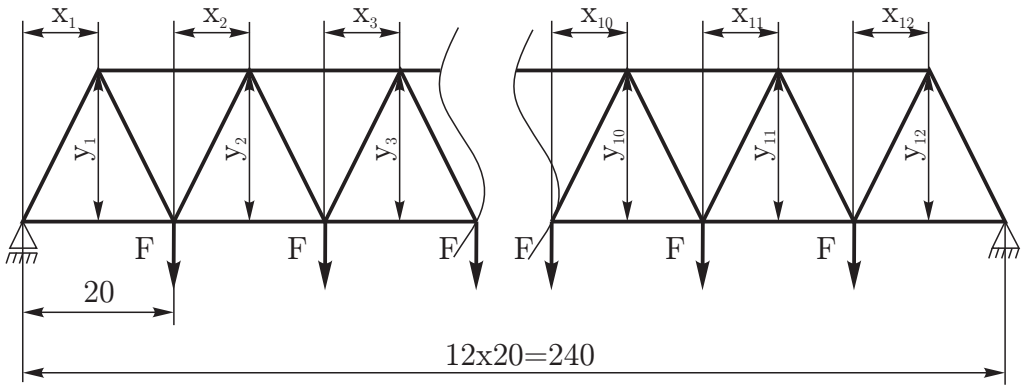

Figure 8.15: Bridge construction

$$1 \leq x_i \leq 19$$

and

$$1 \leq y_i \leq 29$$

Values 0, 20 and 30 where avoided in order to prevent errors due to regeneration failure. Abaqus would otherwise have resulted in an exception and have aborted the simulation.

This example is used to illustrate the capabilities of DAKOTA for handling a large amount of parameters. No parametric study was performed for this example because of the large amount of changeable parameters. Even if one divided each parameter range into only 10 equally spaced sections, 10^{12} different simulations would need to be made. If every simulation took only one second, the whole study would take more than 32000 years. Therefore no a–priori statement can be made where the optimal solution may be.

The same objective function as described above for the simple truss construction (Eq. 8.9) was used, the constants Since ρ and σ_{max} where also set to 1 kg/mm^3 and 1 MPa, respectively.

8.2.1 Finite Element Model

The bridge truss construction was again modeled in Abaqus using a two–dimensional wire structure and a truss formulation. The material parameters are also the same as displayed in Tab. 8.1. Cross sectional areas A_0 were, for simplification purposes, chosen to be $1\,mm^2$, applied forces are also $1\,N$. As the objective function remained unchanged, the IPO input file is the same as shown in Fig. 8.2.

8.2.2 Optimization

8.2.2.1 Evolutionary Algorithm

The first algorithm applied to the truss bridge construction was again the *coliny_ea* evolutionary algorithm. A maximum number of 5000 function evaluations and a maximum number of 1000 iterations was used, each population has a size of 50, whereas 10 survive each mutation.

Fig. 8.16 provides an overview of the evolution of the objective function. At first there is a rapid decrease in the objective function within the first 1000 function evaluations, afterwards the function somehow stabilizes if one neglects the outliers. The objective function seems to drift off very easily, since the number of outliers is quite high, indicating a highly distorted shape.

The optimal solution was captured at function evaluation 4943 and is listed in Tab. 8.6. A graphical illustration of the result is available in Fig. 8.17. It is quite obvious that the solution found is not the global minimum since it is not symmetric, but the overall shape looks very much like a typical bridge construction. The arc–like shape provides the best support for the applied load.

To improve the simulation further, another run was made, this time using 9000 function evaluations instead of 5000. The results are displayed in Fig. 8.18. Although almost twice as much function evaluations were made, the result seems not to have improved significantly. There is still a lack of symmetry and the objective function does decrease only by 1.2%.

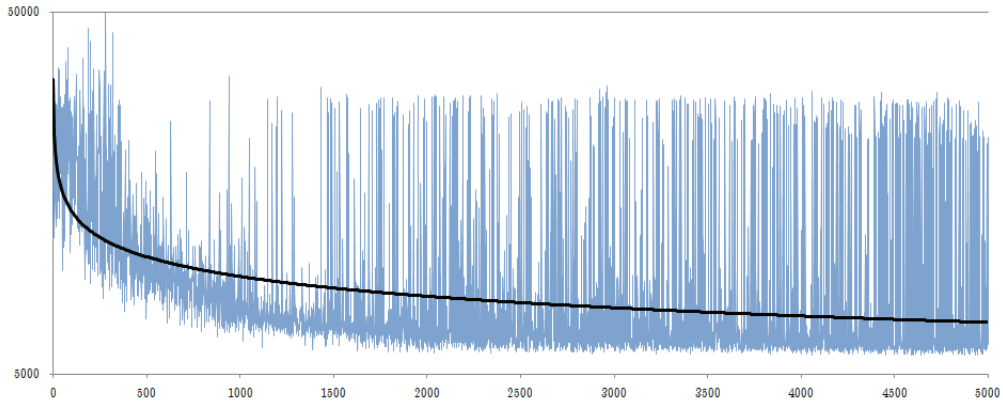

Figure 8.16: Evolution of the objective function using the evolutionary algorithm, first run

x1	y1	x2	y2	x3	y3	x4	y4	x5	y5
10.5	13.1	11.9	21.9	1.8	25.1	3.4	29	10.2	29

x5	y5	x6	y6	x7	y7	x8	y8	x9	y9
10.2	29	7.1	28.3	5.3	29	9.1	29	15.4	27.7

x10	y10	x11	y11	x12	y12	function value
14.9	24.5	7.6	20.5	12.3	11.6	5653

Table 8.6: Optimal parameters for the bridge, length in mm and function value in kg

f = 5653 kg

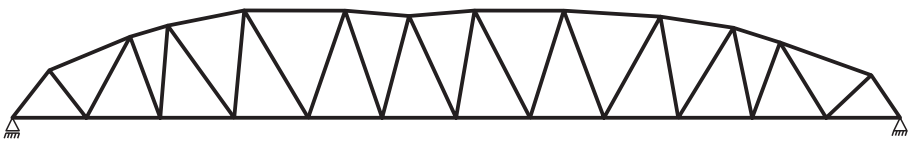

Figure 8.17: Graphically illustrated solution of the bridge, first run

f = 5585 kg

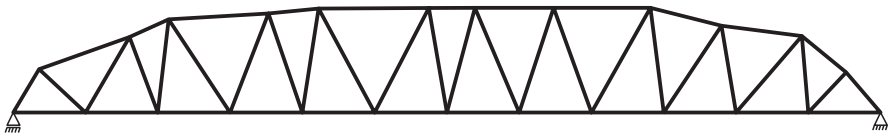

Figure 8.18: Graphically illustrated solution of the bridge, second run

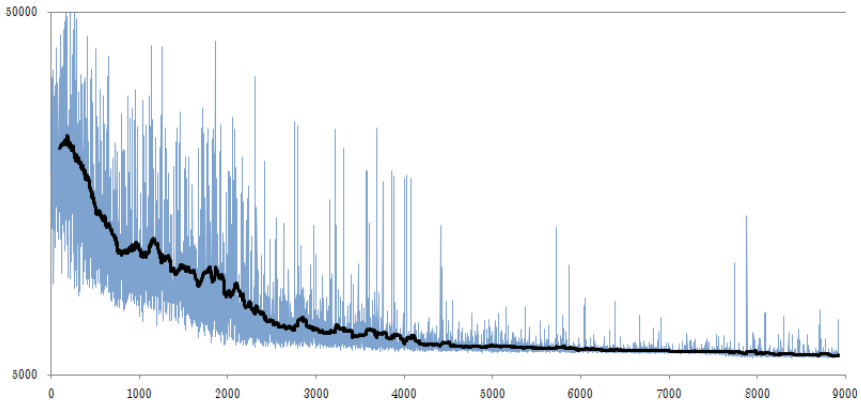

Figure 8.19: Evolution of the objective function using the evolutionary algorithm, second run

8.2.2.2 Gradient Based Algorithm

The same input file as in Fig. 8.10 was used, except that a maximum number of function evaluations was increased to 3000. Three runs where made, each with a different set of initial values as summarized in Tab. 8.7. All the values for x_i and y_i, respectively, where chosen to be the same. The first run represents a reasonable set of values for an initial design, the second and third one represent the lower and upper bounds of the parameters.

	x_i $[mm]$	y_i $[mm]$
1.	10	15
2.	1	1
3.	19	29

Table 8.7: Initial values for the bridge

Fig. 8.20 graphically illustrates the results of the gradient based approach. The top trusses show the initial designs, the bottom one the resulting design. All three simulations show a very reasonable outcome. The first one leads to the best result, although only very marginally. All of them end up in the same arc–like structure as the evolutionary approach, however, they seem much more symmetric. The x–values of the resulting structures are very similar to the initial values, one can see that their angle looks very much the same.

The gradient based approach also leads to a much shorter simulation time, the first one needs 338, the second one 1298 and the third one 306 function evaluations. The second one takes much longer because its initial shape is very distorted, however this is still a reduction of about 75% compared to the evolutionary algorithm.

8.2.2.3 Hybrid Algorithm

Another hybrid simulation was made using the same parameters as given in Fig. 8.13, except that only 500 function evaluations where used. Fig. 8.21 graphically illustrates the results of this optimization. (a) shows the result of the evolutionary algorithm which is then refined by the gradient based algorithm to the final result (b). Fig. 8.21 (a) looks quite random due to the stochastic nature of the evolutionary method, (b) looks very much the same as all the other optimization results, the arc–like shape is clearly visible.

The evolution of the objective function of the hybrid optimization is illustrated in Fig. 8.22. One can clearly see that 500 function evaluations are not enough to result in a noticeable downward trend, but after 500 iterations the gradient based algorithm kicks in and uses the best of the previous evaluations as initial design. The gradient based method then converges within a short amount of time and leads to a very feasible design.

8.2.3 Discussion

The evolutionary algorithm used with this example clearly reaches its limitations. Although a very high number of function evaluations (9000, taking

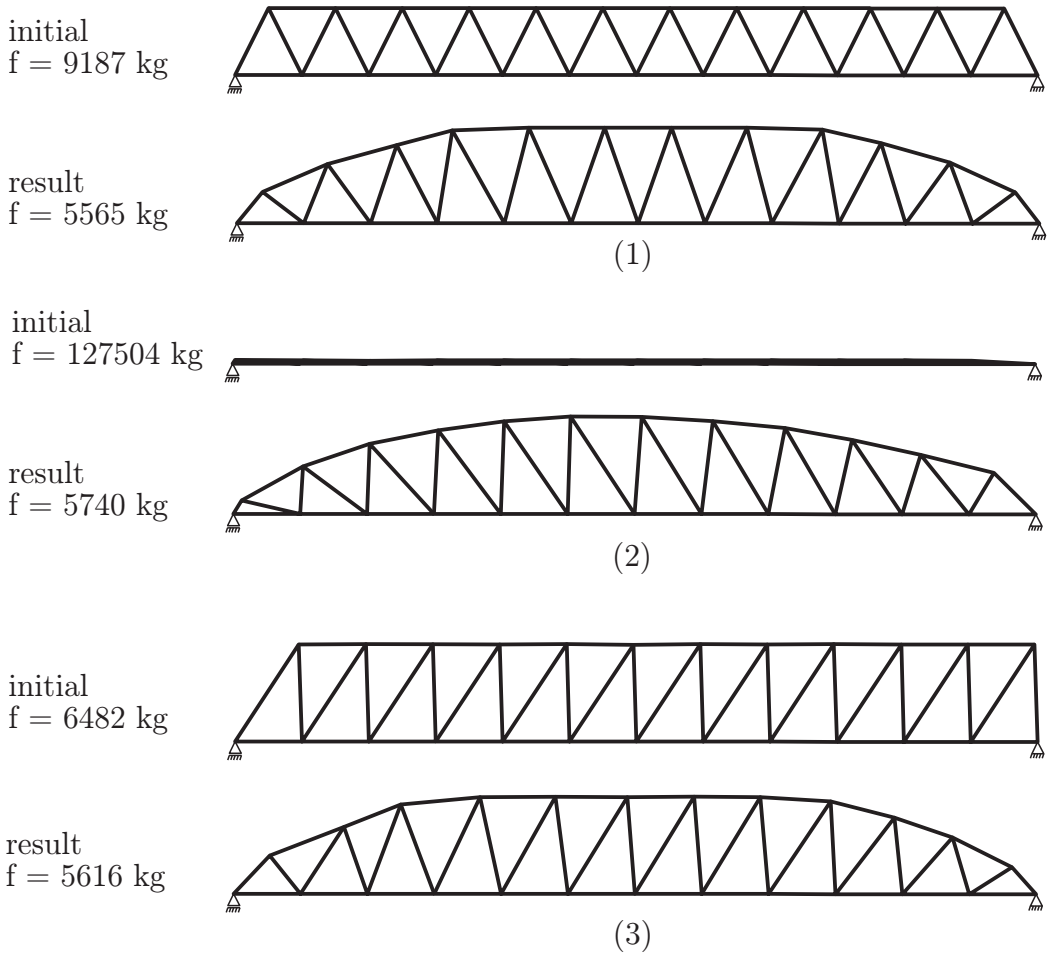

Figure 8.20: Results of the gradient based approach

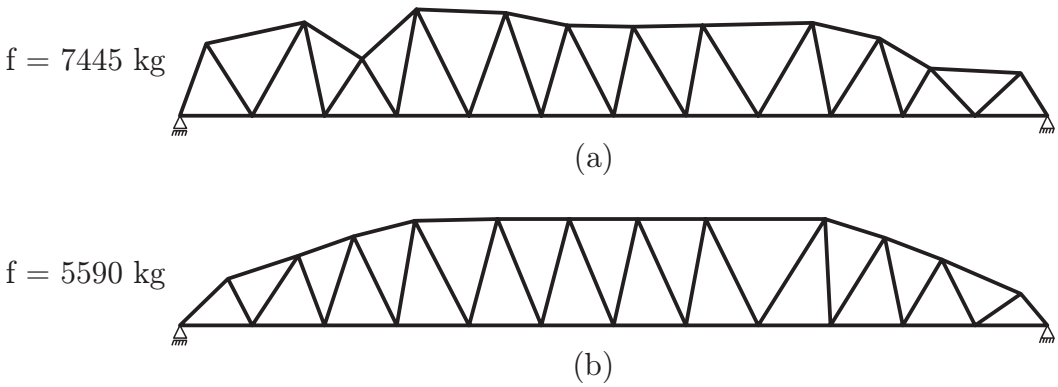

Figure 8.21: Results of the hybrid optimization

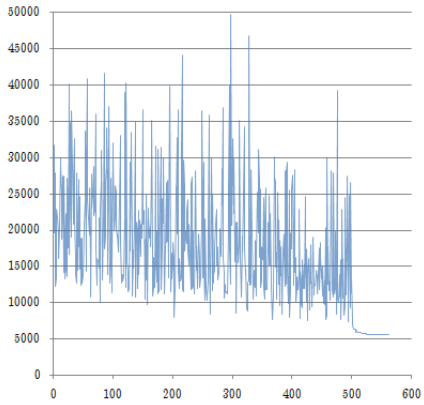

Figure 8.22: Evolution of the objective function during the hybrid optimization

approximately 1.5 days) was chosen, this type of algorithm does not lead to a satisfactory result. The lack of symmetry indicates that the global optimum has not been found. The high number of parameters for this example exceeds the capabilities of the evolutionary algorithm. When choosing the parameters randomly, a feasible design is not found within a reasonable simulation time.

Surprisingly, the gradient based algorithm leads to much more feasible designs, even when using a set of distorted initial values. The best of all results could be achieved by the first run illustrated in Fig. 8.20. The structure looks very symmetric, thus indicating a close call to the global minimum, but this was achieved via a very good initial structure. Even the two other runs with their quite distorted initial values lead to two feasible designs. Although the center beams are leaning to the same side as in the initial design, the objective functions differ by only 3% and 1%, respectively. Again, the gradient based approach extremely fast, taking on average only 647 function evaluations for reaching the optimum.

The last approach for this example was the hybrid one. Fig. 8.21 (a) shows the result of the evolutionary algorithm, which looks quite distorted. Fig. 8.21 (b) illustrates the final design found by the gradient based method using (a) as an initial value. It looks almost symmetric, except the middle right side, and the objective function hardly differs from Fig. 8.20 (1). The

hybrid algorithm seems to be a very feasible method for this example, since it leads to an excellent result while consuming very little time. It takes only about 900 function evaluations, compared to the evolutionary algorithm this is a decrease by 90%. Compared to the gradient based algorithm this is only about 60%, but without the need to chose a suitable initial design first.

Concluding, one can say that the hybrid algorithm is the algorithm of choice if it is not possible to guess a good initial design. If the optimal design is obvious, the gradient based algorithm leads to a faster solution, although one can never be sure that the guessed design is the optimal solution indeed.

Chapter 9

Concluding Remarks

During this thesis the Interface for Parametric Optimization (IPO) between the Open Source optimization library DAKOTA and the finite element solver Abaqus was developed. With this interface one can parametrize finite element models and optimize them regarding an arbitrary objective function. Any mathematical combination of field outputs available in Abaqus can be chosen as objective functions or restrictions.

The interface proves very useful if one needs to run parametric studies and optimizations. In case of parametric studies, one can save a significant amount of time by not having to make each simulation by themselves. One only has to define the parameters within their ranges and IPO completes the task. DAKOTA also provides several optimization routines, which are very useful during the design stage of a production process.

The developed program was then applied to two examples. A simple truss construction for validation and a more sophisticated bridge construction to show the limitations of the different algorithms used. Tree different algorithms where used during this process, an evolutionary one, a gradient based one and a hybrid combination of both previously mentioned. Both examples show excellent results and the interface proves its capabilities.

List of Figures

List of Tables

Bibliography

[1] Linux manual page, 2009. http://linux.die.net/man/2/fork/.

[2] W. Alt. *Nichtlineare Optimierung*. Vieweg, Braunschweig–Wiesbaden, 2002.

[3] DAKOTA. *Design Analysis Kit for Optimization and Terascale Applications*. 2009. http://www.cs.sandia.gov/DAKOTA/.

[4] Scientific Tools for Python. NumPy — N–dimensional Array manipulations, 2001. http://www.scipy.org/NumPy/.

[5] F. Grün. *Form– und Topologieoptimierung unter Berücksichtigung der Betriebsfestigkeit*. 2002.

[6] D.W. Heermann. *Computer Simulation Methods in Theoretical Physics*. Springer, Berlin–Heidelberg, 1986.

[7] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice Hall PTR, Englewood Cliffs, 1988.

[8] B. Kost. *Optimierung mit Evolutionsstrategien*. Harri Deutsch, Frankfurt am Main, 2003.

[9] Sandia National Laboratories. *DAKOTA User's Manual, Version 4.2*. Livermore, 2007.

[10] Sandia National Laboratories. *DAKOTA Reference Manual, Version 4.2*. Livermore, 2008.

[11] E. Laporte and P. Le Tallec. *Numerical Methods in Sensitivity Analysis and Shape Optimization*. Birkhäuser, Boston–Basel–Berlin, 2003.

[12] Mallet and Schmit. Structural Synthesis and Design Parameters. *Hierarchy Journal of the Structural Division*, 89(4):269–299, 1963.

[13] C. Mattheck. *Warum sie wachsen wie sie wachsen — Die Mechanik der Bäume*. Kernforschungszentrum Karlsruhe, 1988.

[14] C. Mattheck. *Design in der Natur — Der Baum als Lehrmeister*. Rombach, 1992.

[15] H. Neuber. *Kerbspannungslehre. Theorie der Spannungskonzentration. Genaue Berechnung der Festigkeit*. Springer, Berlin–Heidelberg, 1957.

[16] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in Fortran 77. The Art of Scientific Computing*. Cambridge University Press, 1992.

[17] E. Schnack. *Ein Iterationsverfahren zur Optimierung von Spannungskonzentrationen*. Number 589. 1978.

[18] A. Schumacher. *Optimierung mechanischer Strukturen*. Springer, Hamburg, 2004.

[19] P. Siarry and Y. Collette. *Multiobjective Optimization*. Springer, Berlin–Heidelberg, 2003.

[20] Dassault Systems. *Abaqus Analysis User's Manual 6.7*. 2007.

[21] Dassault Systems. *Abaqus Scripting User's Manual 6.7*. 2008.

[22] G. N. Vanderplaats. CONMIN, a FORTRAN program for constrained function minimization. *Technical Report TM X–62282, NASA*, 1973.

[23] H. Ziezold and K. Kirckeberg. *Stochastische Methoden*. Springer, Berlin–Heidelberg, 1995.