Optimization of mechanical structures by means of a biological algorithm

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation.

vorgelegt von

Diplom-Biologe Florian Esterhammer

aus

Heidelberg

Berichter:
Universitätsprofessor DI Dr. Werner Baumgartner
Universitätsprofessor Dr. rer. nat. habil. Hermann Wagner

Tag der mündlichen Prüfung: 17. August 2015

Declaration of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon any copyright and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices.

Summary

Steel beams are used to build all manner of structures, for example storage racks for high rack warehouses. They have remained unchanged in their design for decades. At a glance, reductions in production costs could mainly be achieved by one of two approaches. The first one would require the use, and potentially development of, lighter construction materials. The alternative entails the development of a profile that can match the current designs stability whilst reducing the amount of material required. Such profiles can be found by either looking for an inspiration in other fields or by trying to optimize the profile through various methods. The availability of a suitable biological model provided, an improved profile might be found through a so-called top-down process. Alternatively, as all suitable biological model are the result of millions of years of evolution, this process of adaptation and optimization can itself be the inspiration for a technical solution, which in turn, can be used to solve the initial problem. In this context optimization is the improvement of a solution (the profile) to the maximum of a related metric of interest (the reduction of material used). Computational implementation of the evolutionary process were first developed during the 60's and, to this day, are used as an approach to solve optimization problems. Over the years variations of the concept appeared under different names (e.g., evolutionary program, genetic program, evolutionary algorithm). The implementation also differ between the algorithms. For example some omitted crossovers relying solely on mutations to introduce diversity into the population. Other differences include, among others, the encoding of the individuals or the population size. One distinctive feature that hasn't been found in the literature previously and is introduced in the presented algorithm is what has been dubbed "conserved segments". This feature allows part of the individuals' genetic code to be protected from changes from both mutation and crossover operations. This allows the addition of characteristics into the population. It guaranties that a feature that would not necessarily occur in the natural progress of evolution is present in the population and thereby considered in the optimization process. The results gathered while testing the capabilities of the algorithm prove the functionality of the algorithm as well as several of its key features, including the ability to optimize both open and closed profiles, combined with the usage of the novel conserved segments.

All test-cases shown could be improved over the course of 10000 generations. Two of the cases also exemplify the substantial potential for improvements possible for the initial problem by reducing the material requirements by a two-digit percentage while still fulfilling the stability requirements of the reference profiles. Thus the thesis is a proof of concept for both, the concept of conserved segments in genetic algorithms as well as for the potential of improvements to the profiles of beams. After some further optimization of the algorithm and its parameters, it could be used to develop new profiles on demand fitting niche criteria in some fields. To provide even more precise results closer to reality, a fork of the algorithm has been developed that employs a third-party open source finite-element simulation tool to determine the stability of profiles. While this version is fully functional, the prohibitively long runtimes of the complex simulation prevented the running of full tests. It is presented in the appendix.

Zusammenfassung

Stahlträger werden in vielerlei Strukturen verwendet, wie zum Beispiel in Hochregallager. Ihr Profil ist seit Jahrzehnten unverändert. Auf den ersten Blick sind Reduzierung der Produktionskosten durch eine von zwei Herangehensweise erreichbar. Die erste erfordert die Nutzung, und vermutlich vorherige Entwicklung, von leichteren Baumaterial. Die Alternative bedingt die Entwicklung eines Profiles welches dem aktuellen Profil in Stabilität ebenbürtig ist und dabei wenig Material verbraucht. Solche Profile können durch andere Bereiche inspiriert sein oder durch verschiedene Methoden optimiert werden. Sofern ein biologisches Model existiert, könnte ein besseres Profil durch einen sogenannten Top-Down Prozess gefunden werden. Alternativ, da alle biologischen Modelle das Ergebnis von millionen Jahren an Evolution sind, kann dieser Prozess der Anpassung und Optimierung selbst als Inspiration für eine technische Lösung dienen, die selbst wiederum genutzt werden kann um das ursprüngliche Problem zu lösen. In diesem Kontext bezeichnet Optimierung die Verbesserung einer Lösung (das Profil) zur Maximierung einer Metrik von Interesse (die Reduzierung des benötigten Materials). Computergestützte Implementierung des Evolutions Prozesse wurden zuerst in den 60er Jahren entwickelt und werden bis heute als Herangehensweise zur Lösung von Optimierungsproblemen genutzt. Über die Jahre hinweg erschienen Variationen des Konzepts unter verschiedenen Namen (e.g., evolutionary program, genetic program, evolutionary algorithm). Die Implementierung unterscheiden sich auch unter den Algorithmen. Manche verzichten zum Beispiel auf Crossovers und verlassen sich ausschließlich auf Mutationen um eine Diversität in die Population einzuführen. Andere Unterschiede zeigen sich unter anderem in der Enkodierung der Individuen oder die Populationsgröße. Ein besonderes Merkmal welches in der Literatur bislang nicht erwähnt wurde und im vorgestellten Algorithmus vorgestellt wird, wurden "Konservierte Segmente" genannt. Dieses Feature erlaubt es teile des genetischen Codes eines Individuums vor Veränderungen durch Mutationen und Crossover zu schützen. Dies ermöglicht es einer Population Charakteristiken hinzuzufügen. Das garantiert, daß diese Eigenschaft die nicht zwangsläufig im natürlichen Verlauf der Evolution auftritt, dennoch in der Population vorhanden zu sein und somit im Optimierungsprozess berücksichtigt zu werden. Die während der Test des Algorithmus gesam-

melten Ergebnisse beweisen seine Funktionsfähigkeit und die einiger seiner Schlüsselmerkmale, einschließlich der Fähigkeit sowohl offene als auch geschlossene Profile zu optimieren, gepaart mit der Nutzung der neuen Konservierten Segmente. Alle gezeigten Testfälle konnten über 10000 Generationen verbessert werden. Zwei der Fälle zeigen beispielhaft das beeindruckende Verbesserungspotential des intialen Problems, mit Materialeinsparungen im zweistelligen Prozentbereich bei Erhalt der Stabilitätskriterien der Referenzprofile. Die Dissertation erfüllt somit seinen Zweck als Machbarkeitsstudie für die Konservierten Segment im genetischen Algorithmus, aber auch für das Verbesserungspotential von Trägerprofile. Nach einer Optimierung des Algorithmus und seiner Parameter, könnte er genutzt werden nach Bedarf für Nischen zu entwickeln. Um noch genauere Ergebnisse zu erzielen, wurde eine Abspaltung des Algorithmus entwickelt, welches ein open source Finite Element Simulations-Tool eines Drittanbieters nutzt um die Stabilität der Profile zu ermitteln. Diese Version des Algorithmus ist zwar voll funktionsfähig, jedoch konnten aufgrund der langen Laufzeit der komplexen Simulation keine vollständigen Testfälle durchgeführt werden. Dieser Algorithmus wird im Appendix beschrieben.

Contents

List of Figures

Chapter 1

Motivation

To improve is to change; to be perfect is to change often.
- Winston Churchill

1.1 Biological structures

Scientist estimate about 8.7 billion species to be on this planet [30]. Among them we find an amazing diversity of structures on various scales. Each and every one of these structures is highly adapted to its role. These adaptations evolved over centuries and millennia facing and surmounting the same physical constraints and challenges [28]. Many of these challenges of can be encountered in similar form in other places. For example mechanical engineers might be interested in finding a structure matching specified stability criteria while minimizing the resources to be invested in the creation. The field of bionics generally offers two approaches: the bottom-up and the top-down approach. In a bottom-up process, following a discovery of biological principles, others fields are sought in which these might be useful. In the top-down approach other fields seek out biological models that already solved their question and try to understand how the model achieved this. Returning to the example of the mechanical engineer, the top-down approach could be used to search for highly stable biological structures that could be analysed. This could however take a very long time and only solve a single problem. Another approach would be to not look at the result, trying to understand it, but at the solving process instead. Instead of looking at the mechanical structure of various organisms one could instead look at the evolutionary process through which the organisms culminated in their respective forms, and through which they are constantly being refined and improved as time goes on.

1.2 Method for improvement

With the advent of computing technology, the number solvable problems dramatically increased. Many formulas, that were previously deemed too complex to solve within a reasonable amount of time (meaning less than a two digit or more amount of years of continuous computation) or even at all, could now be resolved through the use of clever algorithms. With increasing computing power, the time required shrunk even further and yet more and more complex problems became solvable. While the hardware improved, the mathematicians and software engineers were not idle. A growing number of algorithms was developed to solve the problems while taking advantage of the latest hardware. Where traditional algorithms failed to produce results within acceptable times, heuristics where developed.

The definition of algorithms is a still active field of research. While the fundamental description of a sequence of tasks to be performed to obtain a result are undisputed, there are ongoing attempts at further refining the definition. For example, Donald Ervin Knuth introduced certain requirements like the finiteness (an algorithm can not posses an infinite number of steps), definiteness (each of the steps must be defined as to leave no ambiguousness), input (data given to the algorithm), output (data returned by the algorithm, having a relation to the input), and effectiveness (the algorithm must terminate in a finite amount of time) [21].
Heuristics on the other hand are *programs which are useful to find solutions without guaranteeing success but giving either a good (while not necessarily best) result or at least a good approximation* [37].

Laymen might wonder what use a program is, if running it might not even return a result and if the returned results are not exact. While this question is valid, so is the existence of heuristics. Whenever a formula or problem is encountered that either cannot be solved by an algorithm or cannot be solved within a reasonable time, a decent chance to obtain an approximate result, a quite good answer or in some cases even an answer at all is preferable to having nothing.

The genetic algorithm ([13], [29]; also called evolutionary algorithm, genetic program or evolutionary program, see [26] and [9]) is a quite interesting concept for optimization which did not spring from mathematical concepts but from a biological one. The concept is often credited to J.H. Holland [18] and much work was also done by his student [6]. As such it often garners the attention of biologists in need of optimization but lacking knowledge of higher mathematics mandatory for the understanding of the theories and explanations of alternatives given by the field of mathematical optimization. So it posses a more intuitive alternative for biologically minded users and as such is the focus of this work.

However genetic algorithms are still complex enough to merit a more in-depth explanation than can be given at this point. This blatant shortcoming

will be mended later on.

1.3 Case study: Beams for industrial racks

For practical reasons the investigation of the concept of optimization via genetic algorithm shall be conducted with the help of a concrete case. This will hopefully prevent the reader and the writer from getting lost in obfuscating abstract explanations.

The choice of an example isn't an easy one. Even though genetic algorithms have been applied to a vast number of topics in diverse fields like signal processing [41] or in the banking sector [4] to name two, there yet remain countless issues to be solved. While a purely theoretical case might be in itself interesting, one that is of interest to the industry may be preferable though, as it might be supported or at least rewarded with some funding.

Through fortuitous coincidence a steel company came forth with an interest in the possibility of improving their products. However dreading costly investments without any guarantee of return, a proof-of-concept for one aspect was requested before any investment decision were to be made. The product to be optimized is an industrial rack system. While the company generally wishes to reduce the production costs, primarily only the profile of the beams is to be modified to still support the same weight as the current ones, but requiring less material to make. For preliminary results, the stability of the profiles was to be compared by looking at the *geometrical moment of inertia*. A cooperation with substantial funding as well as more research jobs was announced, should the potential saving of material yield sufficient profit.

1.4 Objective of this study

A preliminary objective of this thesis is to develop a genetic algorithm to optimize the profile of a beam (such as those used in the construction of industrial racks) to use as research environment for all subsequent objectives. As previously mentioned, the first goal is to implement the geometrical moment of inertia as a fitness function. While it is acknowledged, that this function in itself is not sufficient to rate the stability of the beam (as there are many more strains to consider than just axial moments of inertia), it is an acceptable indicator. Therefore it will be used to determine a percentage of material reduction that might be expected by optimization of the profile. Once this is achieved, an alternative fitness function will be developed to provide more precise data concerning the stability of a beam created based on the generated profile.

To add challenge to the task, it has been requested to integrate a possibility to predetermine one or more parts of the profile. This is necessary to

guarantee that certain engineering needs are still met (e.g. a straight surface or a specific angle between two facets that might be essential to fixate the beam with screws, bolts or other mounting systems). This concept will be referred to as *region locking*. Respectively a *region lock* denominates the variable encoding it. Hence the affected part of the profile may be called a *locked region* and therefore considered to be *locked*.

Additionally, the algorithm must support the creation and optimization of both open and closed profiles. Imagining the beam as a construction of a metal sheet repeatedly bent along one axis. A closed profile is one in which the two edges of the sheet along the bending axis are weld together. For an open profile the two edges along the bending axis are left in their bent state without physical connection, not even needing to be near each other.

The ultimate objective of this study, is to provide an algorithm that optimize a set of coordinates which can be interpreted as an open or closed profile of a beam. The stability of those profiles is determined by a third-party finite element software which is controlled by the algorithm through an interface function. This is done to ensure, that the algorithm is reliable and the results trustworthy, to the point of being deployable in the industry.

1.5 Thesis structure

Now that the purpose of this study is clear and its actor is motivated and eager, the tools must be properly introduced. Since the genetic algorithm takes more than just a little inspiration from the biological topics of genetics and evolution, it is quite sensible to first introduce this origin. This chapter will be intentionally kept brief and only cover the aspects relevant to the algorithm without delving into the molecular and biochemical minutiae which, while scientifically interesting, don't add anything useful to the programmatic adaptation of the fundamental principles. Prepared with the basic concepts of the biological motive and some technical terms, the genetic algorithms, as well as related notions by different names, will be described in more details. Definitions that might differ from the biological source as well as new terms will be explained.

Proper introductions being settled, the genetic algorithm developed during the thesis will be presented step by step, one key function at a time, emphasizing code and functionality additions made necessary by special requirements of the case study, as well as noting challenges that arose due to those requirements.

As part of the presentation of the interface to a third-party finite element software, the creation of a 3-dimensional mesh through another third-party software will be covered in as much detail as deemed necessary, before examining the finite element software itself. More specifically, how to control it in an automated fashion from an external program. This chapter will

close on the import of results from the finite element software back into the genetic algorithm.

The chapters to this point should adequately answer all question pertaining to the how and why of the implementation of the genetic algorithm for this case study. Thus the explanatory part of the thesis will conclude in a short summation of all results gathered and lessons learned during the development. Finally, as a closing point, a brief outlook of possible further work on the algorithm shall be given.

1.6 Goal roundup

The core of the work towards the graduation consists of the development process which culminates in this thesis delivering the following:

A basic genetic algorithm

- Optimizes open and closed profiles
- Respectively uses the length and circumference of the profiles to gauge their fitness
- Uses the second moment of area as condition of viability of individuals
- Capable of exempting segments of the profile from modifications during the optimization process
- This algorithm will be used to generate preliminary results shown in the thesis

An advance genetic algorithm

- Optimizes open and closed profiles
- Respectively uses the length and circumference of the profiles to gauge their fitness
- Able to interact with an open-source finite element software (FES)
- Able to generate 3-dimensional geometries and meshes (through third-party software)
- Able to use data from the FES to gauge the viability of individuals
- Capable of exempting segments of the profile from modifications during the optimization process
- Able to distribute workload on several computers

Chapter 2

Genetics and Evolution

2.1 The theory of evolution

Considering the fast pace at which new scientific results are published these days, the theory of evolution can be considered old. Early evidence of this line of thought dates back to the late 18th and early 19th century, for example in Lamarck's *Philosophie Zoologique* (published 1809). However it was truly popularized during the second half of the 19th century after Charles Darwin published *On the origin of species* in 1859, his probably most famous work today.

While Darwin was not the first proponent of the concept of evolution by at least five decades and while some of his explanations have been improved upon if not entirely changed by more recent research, his famed book is to this day oft cited and quoted in high profile journals (e.g. Nature [31]). But what did Darwin write, that remained relevant throughout one and a half century of research?

According to the first chapter of *On the origin of species*, Darwin, while observing both animals and plants, perceived differences between wild and domesticated living beings as well as a higher diversity among domesticated animals and plants. Presumably, this led to further investigations. Domesticated animals, house pets and livestock alike, as well as agricultural crops were good models to observe as they had and to this day still have counterparts in the wild but they were also a subject of interest to people for centuries, offering the experience of those working with them, like farmers, botanists or breeders, on top of Darwin's own observations.

For that matter, differences were noted not only within breeds of pets and livestock and various cultivars of crops, but also to wild types. Let us consider the example of a cow. First domesticated some 10 000 years ago [3], Darwin remarked larger size of the udder compared to its wild congener. Discussion with breeders would reveal to him, that this is one result of careful breeding. This is a process developed by mankind along animal husbandry

and consists of choosing both a male and a female of compatible species, both carrying desirable traits (in this case the term desirable stems solely from the human point of view), and encouraging the mating, thus producing one or more offspring hopefully characterised by the sought traits of both parents. Such offspring would later on be further encouraged to reproduced (i.e. would be presented with the opportunity). Animals not displaying the features needed by humans on the other hand would be prevented from reproduction. Repetition of this process for several generations (even within a single lifetime of a human) would hopefully yield populations dominantly characterised by the initially sought traits, in the case of the example, a larger udder, facilitating an increase in milk production, which in itself is a product of interest to humans for consumption.

Thus came to scientific attention, what already was a common artificially controlled practice for centuries, the first major point of the theory of evolution: the predominance of certain traits and features within a population due to selection of progenitors for said characteristics.

2.1.1 Selection by traits

Both breeding and crop cultivation can be considered artificial in regards to the selection and by extension to evolution. As previously mentioned, in those cases, humans choose what traits are desirable and which ones are undesirable. Based on these choices, mates, which already show tendencies of fulfilling the set criteria, are made available to each other while those individuals of the population available to the breeder are removed from the mating process.

The natural form of selection is one in which humans do not interfere in any way. For plants the principle is fairly simple. If the flower of one plant is inseminated with pollen from another compatible plant, it will result in offspring. There is, to my knowledge, no known mechanism for the pollinated plant to detect unfavourable pollen and prevent the process.

Animals in the wild however, often can, at least to some degree, influence the choice of mate. The following will only concern itself with those animals that do have a mating process and excludes all those that indiscriminately mate with any available individual of their species. First off, lets consider the entirety of a compatible species that is geographically within range and physiologically able to procreate (excluding individuals that are either too young, too old or infertile) the initial mating pool. The first and last condition listed here are straightforward necessities for procreation as incompatibility and inability both logically prevent offspring. The geographically limitation however is worth noting. At first glance it seems to limit mating opportunities during a single generation to those individuals who live close enough to each other for a meeting and subsequent mating to be considered a likely occurrence, also supposing that, some migration provided, the next

generation might encounter a new mating pool due to its new location (still assuming an essentially at least similar habitat). However this geographical limitation also includes natural barriers showing up for several generations. For example, take a mountain growing due to plate tectonics thus separating a previously united population into two distinct ones, or consider an island which, due to a lower sea level during an ice age, for several years becomes a peninsula to which some individuals of a population migrate. A rising of the sea level later on divides this population as well. Such once united populations might over several centuries develop quite differently. This notion is of great interest, especially if despite several obvious distinctions, those populations still retain their compatibility and thus allowing to introduce not just gradual, but also significant changes to a population.

Having established a mating pool, actual selection becomes a key factor in the mating process. Among many animals a behaviour akin to courtship has been observed. While the exact process of this courtship greatly varies between species, in essence one partner attempts to woe, impress or subjugate the other. Some animals fight for dominance over one partner [5] or all partners in a group. Some try to impress by building nests [25] or burrows. Others show off some specific feature like plumage [15] or markings, or perform what to humans appears to be a dance [20].

What most courtship behaviours have in common, is the display of fitness through advantageous traits. The fitness of an individual as having a tremendous impact on its odds of procreating and thus to pass on its features is another major point of evolution. However before discussing it, a closer look at traits, features and attributes that make up an individuals fitness is advised.

2.1.2 Hereditary traits

The words trait, feature and attribute, in regards to individuals, are used synonymous in this thesis and refer to a specific property. They can also refer to both, characteristics that in essence define a species, setting it apart from others, as well as qualities that might be uncommon or even new to a species. While most observed traits mentioned by Darwin were external, he did also consider internal ones, like organs. Hence we should also keep those in mind, even though, for reasons of simplicity and readability, this text almost exclusively mentions external ones.

To be truly considered a trait, as this text uses the word, a characteristic must be hereditary. Some properties, Darwin found to be merely a product of the environment (meaning the nature of a habitat including geographical features, climate and nutritional resources) the individual grew up in. He assumed, and maybe also tested through experiments (although no explicit mention of it could be found), that such characteristics would change, should the individual be put in a vastly differing environment during the main stages

of development.

Two interesting observations concerning the heritability of traits were specially mentioned by Charles Darwin. The first one being, that some features while present in the parental generation of an individual might not be discernible in the current generation but may reappear in the filial generation (essentially "skipping" a generation). What, to Darwin, appeared to be a curios quirk of the hereditary principle was, at least for some cases, explained a few years later during two meetings in 1865 and a published paper in 1866 by Gregor Mendel, a silesian friar highly interested in the heritability of certain traits, performing experiments, most famously, with peas. More on this topic will be covered within the section on genetics (2.2). The second observation made by Darwin concerned linked traits. He found that some features seemingly always appeared together. This oddity of sorts will also be covered later on.

Suffice it to say, that almost all traits found in an individual can be usually found as well in one of its progenitors.

2.1.3 New traits

While most traits can be found in an individuals progenitors, this is not true for all traits. In some rare cases a feature appears that has not been seen in any ancestor or even never at all. This does not mean that these new traits simply appear fully functional and perfectly suited to their function. The appearance of new traits is usually a gradual process over many generations. Evolving from a slight advantage to a refined and potentially optimized feature takes even more generations.

The initial manifestation of such a feature can range from a slight difference to the usual appearance of the individual, to a drastic change that can make the individual stand out in a crowd, so to speak. It can be either advantageous, in which case the odds for a propagation in the population over the next generations increase, or disadvantageous, sometimes to the point of being lethal. Some of the more crass examples are probably what Darwin was talking about when mentioning 'monstrosities'.

Such changes are often the results of a set or series of mutations (rarely, if ever, of a single mutation). Due do the usual necessity of several mutations to result in a change, those unusual features appear seldom, but often enough to be noted and mentioned by Charles Darwin, in individuals.

The keyword of this section would be mutation. However, that being a rather modern concept, which to be properly explained, requires some basic understanding of genetics, it will be properly and fully introduced alongside those in a following section.

2.1.4 Survival of the fittest

As it has now been established how individuals come to possess their respective combination of traits, it is now time to contemplate those individuals from the point of their birth (or germination in the case of plants) to the point following a hopefully successful selection for mating, when they bequeath their traits to a new generation.

The following passages will only mention animals but, with some change to vocabulary and limitations, extends to plants.

Bred livestock, house-pets and any other animal under human care usually has a simple life-path. From birth to death, their human caretaker usually provides shelter, food, security and, traits sought by the owner provided, even mates. The only requirement for the survival of those individuals, is that their combination of traits is not lethal by itself.

In the wild, things are quite different. The first challenge of survival is one against adverse conditions of the environment. Living beings must be adapted to their surroundings. In cold climates they need fur or fat to keep their body temperature within a range in which it can function. In hot climates they need ways to vent excess heat. In dry environments they need to stay hydrated. In wet environments they need to be able to swim. Every environment posses unique challenges to creatures daily struggle. But the mere adaptation is not always sufficient. Occasionally natural habitats are struck with more extreme conditions than usual (e.g. a heatwave in hot environments, prolonged harsh winter in cold ones) or with unexpected conditions such as a drought in a wetland or heavy rainfalls in the desert. Such changes (considering non-permanent short term changes) present an additional challenge to populations, typically well adapted to their habitat over several generations. Individuals possessing traits easing such adverseness stand a much better chance at outlasting it and thereby making it to adulthood and to the next mating season. Thus in regards to the environment, unless it is for some reason highly invariable, a certain degree of flexibility seems to be beneficial.

But the ability to survive rough conditions matters not if an individual fails to prevail against the competition of others living in its habitat. The second challenge is one of competition against ones own kind and those who are neither prey nor predator. It is a fight for nutrition which in some areas and some seasons can be highly limited, but also a fight for territory holding said food, shelter of some sort (depending on the animal: ground for burrows, caves or plenty of trees for nesting) and for some species a sizeable population of its kind, increasing the number of potential mates (see the explanation of mating pool in section 2.1.1). The first point is one of immediate importance to all animals from earliest age on. If they are to young to care for food themselves, it is still indirectly of importance, as the immediateness is merely delegated to a caretaker (this is usually a parent

or member of the group of individuals into which an animal was born and almost exclusively applies to species practising brood care). The territory becomes increasingly important as the individual matures and plays are more active part in the group in securing or expanding its area, or as, in the case of less social species, an individual sets out to find a territory for its own. While for the competition for scarce foods individuals with features improving the capabilities of food-acquisition are favoured, for defence and expansion of territory, more fierce attributes benefit the success and survival of the animal enabling the intimidation of would be competitors or outright fights for dominance over the area.

All throughout their lives, almost all animals but the apex predators in an environment are at risk of ending up as source of nutrition for another species. To surmount this third big challenge, various strategies have developed in different species to deal with this threat. Some animals might be quite apt at fighting their predators while others may excel at escaping them. Those two rather well known options aside, there are two more worth mentioning. First off, camouflage, for which animals blend with the environment through shape and colouration. And second, mimicry, for which one species or variety of animal evolves to look similar to another that is for one reason or another not desirable prey. This may be due to an appearance resembling that of a more dangerous predator or due to a semblance to a poisonous species, to just name a few. No matter how an individual does it, surviving predation well into adulthood and until the mating season is essential to the propagation of its own traits within a population.

What all three of these challenges to animals in the wild have in common, is that the sum of their traits allow them to survive each one of them. This sum of advantageous features is called the fitness of an animal. Since a good fitness increases the odds of survival, fit individuals often become instinctively attractive mates and have thus an increased chance of passing on their traits during mating season. The highest fitness individual within a mating pool accordingly has the highest odds of propagating its attributes into the following generation. Therefore scientist in the field of evolution talk about the *survival of the fittest available yet* in regards to the progeny of positive sets of features.

2.2 The bases of genetics

While this short overview on evolution provides a sufficient foundation to understand the underlying principles of a genetic algorithm, a closer look at the fundamentals of genetics is certain to elucidate the mechanisms that inspired the operators and the *encoding* implemented in the algorithm.

This section is not meant to give an exhaustive or even remotely comprehensive summary of the current knowledge in the field of genetics. It will

however briefly introduce some key points that are relevant to the development of any type of genetic algorithm.

2.2.1 The genetic code

Before sexual reproduction or mutations (henceforth referred to as *genetic operations*) can be explained on a molecular level, it is necessary to outline the basic molecules involved in the storage of hereditary information which is the subject of those genetic operations.

The most basic building blocks of hereditary information are called nucleotides. Each nucleotide consists of a phosphate group, attached to the fifth carbon of a sugar ring (a deoxyribose to be precise), which in turn is connected to one of four nucleobases: guanine (G), cytosine (C), adenine (A) and thymine (T). The phosphate groups connect the single nucleotides to long chains by forming a connection to the third carbon of the deoxyribose of the next nucleotide. Based on the connection point of the phosphate groups, an orientation has been defined as going from the connection to the fifth carbon (noted: 5') to the connection of the third one (noted: 3'). Such a chain of nucleotides is called a strand of deoxyribonucleic acid (*DNA*), or strand for short and is rarely found alone. Usually the nucleobases are connected to complementary nucleobases, cytosine and guanine being complementary to each other, as are adenine and thymine. Those complementary bases are again connected by their phosphate groups and deoxyriboses and so form a second, complementary strand which runs in opposite orientation to the first one (which is oft called *anti-parallel*). These two strands are arranged as a clockwise twisting double helix.

One such double strand can, through its sequence, encode a larger number of features. A segment of a strand, whose sequence can be translated into a functional protein (or ribonucleic acid chain), is usually called *gene*. The entirety of all genes of an organism is called the *genome*.

This already gives sufficient the information and the technical definitions needed to cover both needed genetic operations.

2.2.2 Recombination

Recombination is an umbrella term for a set of genetic operations in which whole segments of genetic code from a source is copied to (conserving the sequence in the source), transferred to (the sequence is excised from the source) or exchanged with (the sequence is excised and replaced by a sequence from the target) a target.

The exchange of sequence parts is the model after which a *crossover*-operator is commonly modelled in genetic algorithms. The underlying molecular mechanism driving this phenomenon is not recreated in the function.

Instead, the phenomenon is abstracted and made to represent, as a gross approximation, sexual reproduction between two individuals.

Before biological sexual reproduction can happen, an individual must produce gametes: cells which contain only a single (*haploid*) set of genes (as opposed to the two redundant sets in *diploid* cells of e.g. most mammals) and which are able to fuse with other compatible cells to produce a zygote (an initial cell from which a multicellular organism develops). The act of sexual reproduction itself has the function of bringing the gametes of both parents together to create this zygote which once again contains two sets of genes (staying with the mammal example). These sets, provided neither is afflicted by defects, are essentially redundant. Due to this redundancy the core functionality of a gene is the same regardless of which copy is expressed. The main differences then lie in the alleles, the exact expression forms of a gen. For example, considering a gen coding for the color of the eyes, the basic function of both copies will relate to the mechanisms by which the color is created and will for those parts be relatively identical. The alleles however can differ and determine whether this expression mechanism is to result in a blue, green or brown colouration.

The principles of heredity of traits in general was the lifework of Gregor Mendel. To this day the principles according to which traits are passed down through generations are taught as the mendelian inheritance. To give a brief simplistic overview, consider two individuals of a species for which two traits are contemplated. They are called the parental generation. For each trait one individual will have dominant alleles, while the other one will possess relatively recessive ones. To make things easier, these individuals are considered to hold two copies of the same alleles for each of these traits, and will therefore produce only one combination of gametes with their exact set of alleles. The first filial generation (offspring of the parental generation) will posses both alleles for each traits and exhibit the respectively dominant phenotype for each trait. Now possessing both alleles and contemplating two traits can result in more variety in the produced gametes. Lets denote one trait with the letters 'A' for dominant and 'a' for the recessive allele, and the second one with 'B' for dominant and 'b' for recessive (it is a common convention to note dominant traits in diagrams and models with capital letters and recessive ones with the same letter in lower case). In this way the filial generation has the genotype 'Aa Bb'. Since gametes only possess a single copy of the genome, they might have any of the following genotypes: 'A B', 'A b', 'a B' or 'a b'. Should two individuals of the first filial generation reproduce with each other, it will result in a second filial generation with 16 different possible genotypes and 4 different phenotypes occurring at a ratio of 9:3:3:1 (double dominant:first trait dominant, second recessive:first trait recessive, second dominant:double recessive).

In this way the genetic code of the parents is mixed throughout the generations while largely remaining conserved. Over the course of many

generations and over many more traits than just two, a huge number of combinations is at least theoretically able to appear. The earlier mentioned selection of desirable traits usually leads to an increased likelihood of those traits occurring instead of the purely statistical ratios of traits determined by Mendel under artificial conditions.

2.2.3 Mutations

While recombination creates diversity by making combination of features possible that haven't been seen before, mutations can affect traits in ways that result in entirely new attributes. As such, mutations are a source of great diversity, albeit a very random one at that.

There are generally speaking two kinds of mutations. Chromosome mutations affect the number, form and/or structure of DNA strands. So called gen- or point-mutations affect a single nucleotide in the sequence, and are the inspiration for the most common mutation operator in genetic algorithms.

Such point-mutations come in three variations. They can either delete a nucleotide, add a new one at a position or replace the nucleotide in a position with one of the other three. Due to the way DNA strands are translated to create proteins, this last mutation can sometimes remain without effect when the new sequence happens to code for the same amino acid as the previous one. The first two types of mutations however often have a more drastic effect, as they shift the entire reading frame for the translation process.

Mutations usually don't happen often and their effect is further reduced by efficient repair mechanisms. While natural occurrence of mutations happen, often attributed to slight glitches in the copying process of DNA strands, there are a number of factors, called *mutagens*, that can cause additional mutations beyond this low number. Among those are various chemical compounds (e.g. reactive oxygen species [2], benzene [1]) as well as several forms of radiation (e.g. ultraviolet light [27], gamma-radiation [38]).

Chapter 3

Genetic Algorithms

Observing the astonishing level of adaptation of some animals to their environment, inspired scientists, looking for ways to solve problems that cannot be precisely answered by a formula due to either complexity or insufficient computing power. Over the years many genetic algorithms have been developed all over the world. Some very generic to be used by others for various kinds of problems, others highly specific to the problem as well as to the hardware they are supposed to run on. This chapter will introduce the programmatic adaptations of each of the previously mentioned biological models.

While there are attempts to use different names, like evolutionary strategy, evolutionary program, genetic program, etc., to differentiate algorithms based on the encoding type, as well as the employed operators, these attempts so far seemingly have yet to be adopted by the scientific community at large. Therefore these denominations are used in their most general sense and should not be used as an indicator for the algorithms structure.

3.1 Concept of the model

Genetic or evolutionary algorithms essentially belong to the group of random exploration solvers. Meaning that they, at the highest level of abstraction of its concept, merely test random solutions from the search space (set of all possible solutions). However, what sets them apart is a degree of guidance within the pseudo-randomness of new potential solutions.

This guidance is achieved by considering sets of possible solutions (which, in light of the inspiration, are called populations of individuals) and not only randomize (read: mutate) those but to mix parts of solutions (read: crossover) to create new ones that might yield better results than the ones used originally. This use of previous solutions to create new ones is a first small step in guiding the search for a good, if not a best answer. The second and often more important step lies in the selection of individuals for

mutations and crossovers. Different selection functions (discussed below) have been developed since the creation of the first genetic algorithms. At their core, they are a system to choose individual from a population on which to perform some other operation. Said system mostly involves a relative or absolute (to the population) gauging of individuals to determine their likelihood of being selected and whether they are eligible at all.

Before considering further details on the selection, crossover and mutation operators, similarly to the biological model, the information storage on which those act will be analysed.

3.2 Encoding an individual

While the definite way of storing data depends on the soft- and hardware employed, the encoding of the information is a user choice. The biological model uses an alphabet of four bases to encode information as sequences forming long chains which, due to the size of the molecules as well as their condensed spacial structure, take up only little physical space. On computers, memory can often be a limiting constraint.

The encoding of an individual is most often set in one of two ways. Either with a series of binary numbers or with a series of decimal numbers. Binary is a base two system on which computers are based. No matter the operating system or software, everything the central processing unit (CPU) does is a set of basic operations on binary numbers. Due to this, binary storage of individuals can be done with very little impact on the memory if programmed well. Decimal numbers are written in a base ten system and commonly used by humans. This familiarity makes them a popular choice as the numbers can be more easily interpreted by the user without requiring additional visualisation functions. However, depending on the precision of floating number (i.e. rational numbers) or integers, this type of encoding can become very demanding on the memory. The choice of encoding is usually more dependent on the data to be optimized. Memory concerns being given in many cases only an afterthought if any at all. For high-performance software however this can be a important decision point.

Once the encoding of single individuals has been decided upon, populations can be created. Those essentially being but a certain number of individuals, they are usually stored in array like structures (depending on the programming language they might be known under a different name, e.g. lists or matrices). The implementation of the different functions of a genetic algorithm depend on the encoding and storage solution of the population. Since both the crossover and the mutation functions require a set amount of individuals to be selected as targets of their respective functionality, the selection operator will be discussed next.

3.3 Selection operators

Selection operators are a relatively popular research topic in the field of genetic algorithms. This leads to a decent choice of functions to implement. Since there are too many to cover them all, only a few examples will be mentioned.

Three of the more common selection operations are:

Roulette Wheel Selection (RWS) is a selection type in which each individual has a chance of being selected that is proportional to its fitness. It gets its name from the comparison to a roulette wheel. In keeping with the analogy, each individual is allotted a number of slots on the wheel equivalent to its own fitness relatively to the sum of the fitnesses of all individuals in the population. As the ball in the roulette game will settle in a slot, determining a number and a color, so will a random number pick a slot and the individual attributed to it will be the one selected. So this selection type improves the odds of selection for individuals possessing a high fitness while not completely discarding average or below-average individuals.

Rank base Selection is a process in which all individuals in a population are ranked by their fitness. The fittest individual is ranked 1, the individual with the second highest fitness 2, and so on. Similarly to the RWS, a random number determines the selection of a rank and through it of an individual, but unlike RWS the selection is not dependent on the fitness ratio of an individual but by its fitness-rank within a population as higher ranks are attributed a higher predetermined chance.

Tournament Selection pits individuals of the entire population or from a random subset of the population against each other. The fitness of two individuals from the chosen set is compared and the individual with the lower fitness is discarded while the higher fitness one is compared to the next fitness. This process is repeated until only the required amount of individuals remain.

As mentioned before, there are more selection strategies, but these three depict some of the most frequently used solutions. No matter what solution one chooses to implement and use, they all serve to emulate a selection process of some sort, be it mate selection for crossover, chance for the mutation or survival for the transition between generations. This then leads to breeding which itself is implement as a function usually called the crossover operator.

3.4 Crossover operator

Due to the simpler encoding and lower amount of information in the genetic algorithm compared to the biological model, the crossover operations implementation attempts to recreate the purpose but not the exact mechanisms. One thing that adds complexity however, is the capability of some crossover functions to involve more than two individuals.

The amount of parent individuals and resulting offspring aside, the crossover function can also be characterised by the number of crossover points. That is, the number of points after which the genome of one parent is exchanged with the same segment of another parent (or in the case of more than two parents, the segments are passed along between the parents). The newly blended genomes essentially become the offspring individuals.
Additionally some crossover operators also automatically apply a mutation to each offspring.

3.5 Mutation operator

The mutation operators main function is to provide a degree of randomization to the genome of an individual and thus to the population.

While the operator essentially adds a random value or multiplies by a random factor to some or all values of the genome, the trick, so to speak, for an effective mutation lies in the random number generator and how it is used to change values. The latter highly depends on the value range valid for the genome and in some cases also on the topology of the search space. A dense amount of local peaks and valleys in most approaches necessitates smaller jumps within the search space via randomization of the genome. In contrast, a rather homogeneous or flat search space lends itself to bigger jumps, which, often, can accelerate the exploration of said space.
Valleys and peaks are search space areas of low and high resulting fitness values respectively.

3.6 Fitness function

The fitness function turns the value set of each genome into a number with which individuals can be compared. The fitness function is what makes the genetic algorithm a solver for a specific problem.

Essentially the fitness function is a mathematical formulation of the problem. The function can be written as a problem of minimization (attaining the lowest fitness value possible), but maximization (obtaining the highest value possible) seems to be the more prevalent choice.

The fitness function determines the topology of the search space. The search space itself is the set of all possible genomes for given restrictions.

Assuming an n-dimensional space, where n is the number of values in a genotype, each point of the search space can be assigned a value by the fitness function. The connection of these values, creates a topological map of the search space. Local maxima are called peaks, while local minima are considered valleys (both denominations assume a maximization problem).

Chapter 4

The core of the Genetic Algorithm

Following the overall introduction of genetic algorithms, we will now introduce in detail the base version developed for the case study. This will be done by analysing all key functions which were previously presented in a more general fashion. Where necessary, additional functions required for the case study will be examined within section following the first encounter and merely referred to, when needed again. The figure 4.1 shows the order of the function calls of all custom functions of the algorithm during the setup-phase. Figure 4.2 gives an overview of the custom function calls during the evolution-phase.

Before getting into the code, the hard- and software involved in the development as well as for the tests should be listed for the sake of completeness.

Hardware

- Processor: Intel(R) Core(TM) i5 CPU 660@3.33GHz, 2 Cores, 4

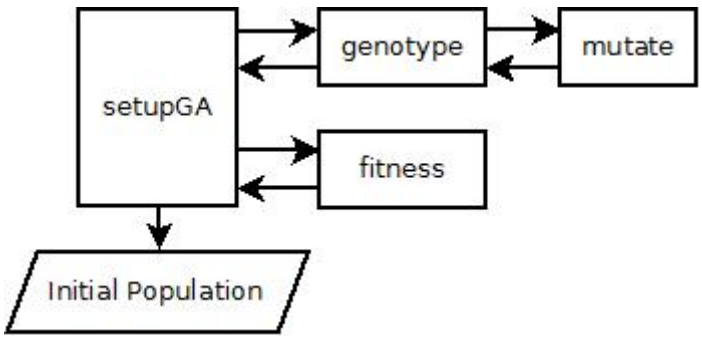

Figure 4.1: Stack flowchart of the setup-phase of the genetic algorithm depicting the call order and dependency of custom functions of the genetic algorithm.

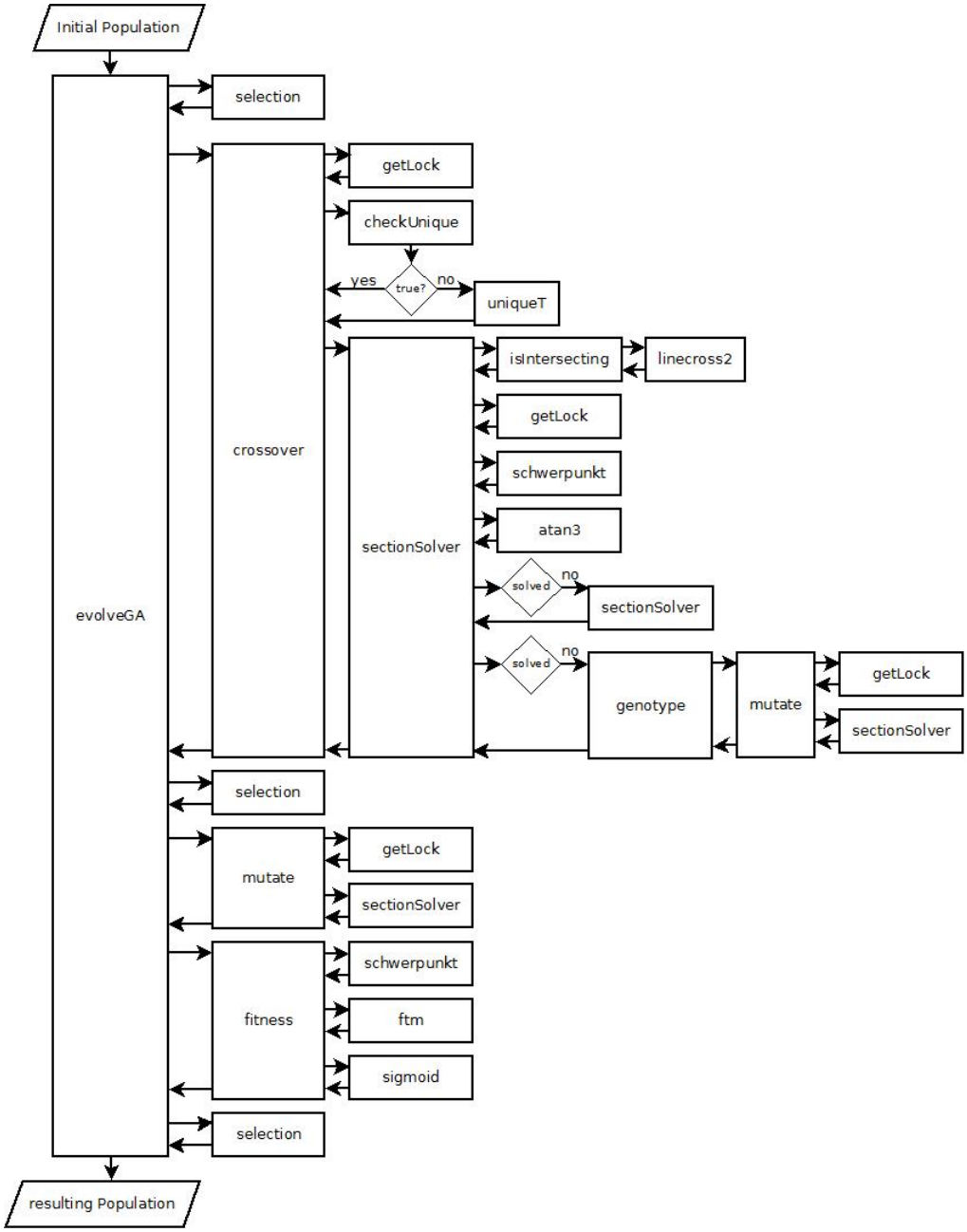

Figure 4.2: Stack flowchart of the evolution-phase of the base genetic algorithm depicting the call order and dependency of custom functions of the genetic algorithm (details of the calls from *sectionSolver* are only shown for the first occurrence but are the same for all).

Logical Processors

- Random Access Memory (RAM): 4.00GB DDR3
- Operating System (OS): Windows 7 Professional (64-bit) Service Pack 1

Software

- Matlab R2012b (8.0.0.783) 64-bit [24]
- Octave 3.6.4 (configured for "i686-pc-mingw32") [7]

While most of the actual development took place in Matlab to take advantage of the many features of its built-in editor, compatibility to the open source solution Octave was a constant concern to ensure usability, even in situations where the costly Matlab Software is unavailable.

4.1 Encoding

As was already mentioned in the first chapter, the test case to be studied essentially concerns itself with the reduction of weight of metal beams in industrial racks, while maintaining their loading capacity.

While material engineers might solve this problem by attempting to improve upon the material used to create the beams, this thesis aims at improving the profile of the beam. An easy way to recreate such a profile is to simply represent each point of it with a set of cartesian coordinates. This unfortunately creates a huge amount of data. But that amount can be greatly reduced by removing all but the first and last point in a straight line and by approximating curves with a limited amount of points.

Depending on the original beam profile, a matrix of varying length and with a height of two can encode any given beam profile. The first row containing the abscissa, while the second one contains the ordinate. A slight drawback to this solution would be the hassle of managing matrices of differing sizes with Matlab as well as some added challenges in implementing a crossover function for such a case. Fortunately these problems can easily be mended by setting an arbitrary but fix length regardless of the original beam profile and deliberately adding additional points between the first and last in a line to fit the matrix size.

4.2 The core function

The role of the core or controlling function, is to set-up all required variables, to maintain the population, perform the crossover and mutation operations on selected individuals and to check the termination conditions.

In this implementation the function is divided into two distinct parts: the set-up *setupGA* and the actual genetic algorithm *evolveGA*. This separation was initially created to allow the genetic algorithm to not only be performed using a newly created population, but to potentially act repeatedly on the some population, allowing (with low generation numbers on each start) the algorithm to run for many generations with interruptions. A user-input driven interruption was also envisioned, but not implemented due to a low priority. Currently, the main use of this system is to test different configurations while excluding the likelihood of better or worse results being caused by the fitness of the initial population. This can be done by simply reusing the same initial population.

The two parts will be examined separately, starting with the set-up. For all following code, a basic understanding of Matlab/Octave syntax will be assumed.

4.2.1 Setting up the genetic algorithm

Code 4.1: setupGA.m source code

```
1  function ga = setupGA(individuals, generations, p_type
       , regionLock, regionValues)
2   try
3    warning('off','Octave:divide-by-zero');
4   end
5   global profile_type
6   if nargin<3
7    profile_type=0;
8   else
9    profile_type=p_type;
10  end
11  if nargin>3
12   if isempty(regionLock)==1
13    regionLock=[[1;1] regionLock];
14    regionValues={[0;0] regionValues{:}};
15   else
16    regionLock=regionLock;
17    regionValues=regionValues;
18   end
19   else
20    regionLock=[1;1];
21    regionValues={[0;0]};
22   end
23   genomeSize=16;
24   genome=zeros(individuals,2,genomeSize);
```

```
25    for j=1:individuals
26     genome(j,:,:)=genotype(genomeSize,regionLock,
           regionValues);
27    end
28    fit=zeros(1,individuals);
29    for j=1:individuals
30     fit(j)=fitness(shiftdim(genome(j,:,:)));
31    end
32    ga={genome,regionValues,fit,generations,profile_type
          };
```

As previously mentioned, *setupGA*s main function is to create an initial population for *evolveGA*. It starts with the self-explanatory keyword *global* on line 5, which makes a variable available to the global scope (by default variables in Matlab have a local scope and are only available within the function in which they have been created) and so to all other functions that are also given access to this variable by adding the same line (*global* followed by a comma separated list of variable names). This approach avoids having to hand over the variable with each function call as well as through nested functions. The implementation of a variable through this global scope is based on convenience and only chosen for those that remain unchanged in value and meaning throughout the algorithm.
The variable in question is *profile_Type*. It determines whether the profile to be optimized is open or closed. Those two cases will be handled slightly differently by some functions. The specific differences will be mentioned in the description of affected functions. The variable is expected to be a boolean. Matlab however does normally neither save nor display boolean data as string but rather as number (the use of the word number here is deliberate as not to infer any detail on the specific internal handling of the format of boolean variables). Thus 0 represents the logical state *FALSE* and 1 represents **TRUE**. Accordingly, if *profile_type* is set as TRUE, an open profile will be considered and subsequently solved by the algorithm. The conditional statement from line 6 to 10, sets the variable *profile_type*. Checking the number of input arguments with *nargin* ensures that required variables are set (in this case the third input variable *p_type* is checked), even if they haven't been specified by an input (automatically set values for attributes are referred to as default values). This allows quicker and simpler function calls for default cases, usually those called most often. While some consider such behaviour as good programming procedure, it is mainly a convenience feature. The default value of *open* is 0 (meaning a closed profile) as can be seen on line 7.

In a further conditional statement between the lines 11 and 19 default values for the *regionLock* and the *regionValues* are set. The first is a two-dimensional matrix of height 2 and variable length. For each regionLock the

starting position is stored in the first row and its total length in the second. The length of the matrix corresponds to the number of regionLocks. The *regionValues* are stored in a cell array. For each regionLock, its values (coordinates as encoded in the genome) are stored verbatim. If neither is given, the tuple of coordinates $[0; 0]$ is set for the first position of the individuals genome. By giving at least one common point to all genomes, the likelihood of the resulting phenotype of different individuals drifting apart in the reference frame of the coordinate system is decreased. Thereby reducing the odds of incorporating spatially distant segments (again in the reference frame of the coordinate system) through the crossover operation which would result in an individual with a low fitness value due to the length of the connection to such a segment.

On line 23, the variable *genomeSize* sets the length of the genome for all individuals. The value of 16 was chosen rather arbitrarily. This variable is set separately instead of in-line to make it easily visible for changes and to permit easy adjustment while still keeping the size consistent over all function calls within *setupGA.m*.

Line 24 creates the zero-variable *genome*. Even though it is more customary to create empty variable when the proper values are note known yet, this is highly inefficient in Matlab when the full set of values cannot be created at once. Instead a variable with the right size is created and simply filled with other values. These values are usually zeros or ones (created respectively by the provided *zeros* and *ones* functions). In theory completely random values would be acceptable too, but generation of those takes more computation resources and is therefore not practised.

The variable *genome* is a three dimensional matrix used to store the genotypes of the individuals. The first dimension size, set by *individuals*, determines the number of individuals in a population. The second dimension has always a fixed size of two. One row for the abscissa and one for the ordinate (see 4.1 on the encoding of the profiles). The third dimension set the genotype length as defined in line 23.

If those two variables have been given, the function *genotype* is called with them as input arguments. Otherwise, both variable *regionLock* and *regionValues* are created as empty matrices (which later on simplifies the code of other functions) and the *genotype* function is called without additional input (a more in-depth look at the *genotype* function is appended to this section). In both cases the function is called once for each individual to be created for the population. The returned genotype is then saved to the previously prepared *genome* variable. Line 28 simply initialises a variable called *fit* as one dimensional zero-matrix. This will hold the fitness value calculated for each individual. The fitness-value and the genotype are linked by their index within their variable. The fitness of the n^{th} genotype in the *genome* variable is stored at the n^{th} position in the *fit* variable.

Line 29 to 31 loop through all individuals to calculate the fitness, which

is returned and saved to the proper position in the variable. The fitness function which will be discussed later (see 4.6) takes a single genotype as input and returns a single integer. To provide a single genotype, the variable j loops through the first dimension of *genome*. This however produces a matrix of the same dimensional size as the original one. In this case a three dimensional matrix with the first one having a size of 1. It would of course have been possible to adapt the receiving function to support such a matrix. Since other functions however do provide the input properly formatted, it was decided to nest the genotype into the *shiftdim* function within the function call. This function, to paraphrase Matlabs "help" command, returns an array sans the unneeded first dimension.

Lastly, on line 32, the variables are added to a cell array which is returned by the function. This is mostly a convenience and a coherence decision. Coherence because, as will be seen in section 4.2.2, it is also used in the *evolveGA* function, and convenience, because it makes it easier to append additional variables for performance analysis without having to adapt the test-functions to each case.

The genotype function

Code 4.2: genotype.m source code

```
1  function genome=genotype(length,regionLock,regionValue
       )
2   global profile_type
3   if ~profile_type
4    minValue=1;
5    maxValue=21;
6    if (mod(length,2))~=0
7     length=length-1;
8     point=1;
9    else
10    point=0;
11   end
12   long=ceil(length/4)+1;
13   short=floor(length/4)-1;
14   long_std=linspace(minValue,maxValue,long);
15   short_low=ones(1,short)*minValue;
16   genome=[long_std,(ones(1,short)*maxValue),linspace(
         maxValue,minValue,long+point),short_low;minValue,
         short_low,long_std,(ones(1,short+point)*maxValue)
         ,long_std(end:-1:2)]+1;
17  else
```

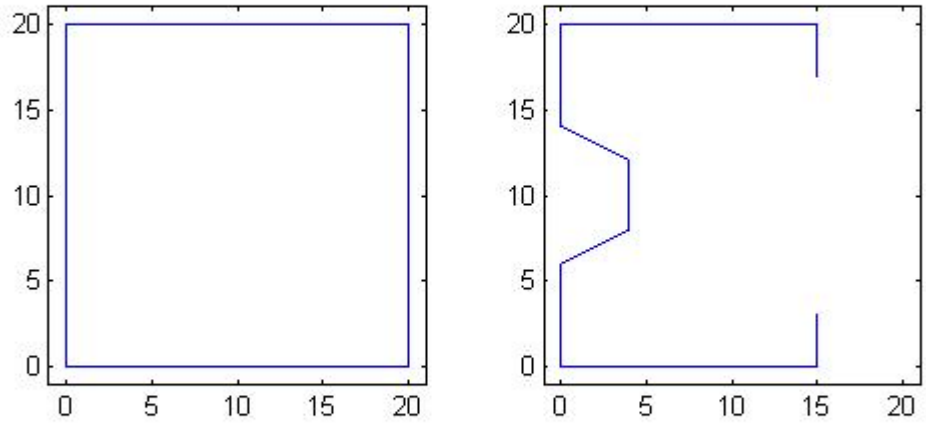

Figure 4.3: Basic shapes from which all genotypes are derived during creation. Left: Square shape as used during the creation of closed profiles. Right: Sigma shape as used during the creation of open profiles.

```
18    genome=[15,15,10,5,0,0,0,4,4,0,0,0,5,10,15,15;
          3,0,0,0,0,3,6,8,12,14,17,20,20,20,20,17];
19  end
20  if nargin>1
21    for i=1:size(regionLock,2)
22      genome(:,regionLock(1,i):regionLock(1,i)+regionLock
            (2,i)-1)=regionValue{i};
23    end
24  else
25    regionValue=[];
26  end
27  genome=mutate({genome,regionValue},0.1);
```

This functions purpose is to create a genotype by slightly mutating base shapes. In the case of a closed profile the code between the lines 3 and 16 creates a square with a side length of 20 units (actual measuring units do not matter at this point and will therefore be forgone). Most of this code segment automatically creates the genotype based on the length given by the input argument and the length of the square sides, while trying to distribute the points evenly among all sides.

In the case of an open profile, which is covered by the lines 17 through 19, it was specifically requested to use a shape resembling the Greek capital letter sigma (Σ). Not being given exact sizes or even proportions or ratios, the form was approximated and deemed adequate.

In either case, if a *regionLock* argument is given, parts of the genotype are replaced according to the code from line 21 to 23. As each column of

the matrix codes for a distinct protected segment of the profile (as discussed in the previous section 4.2.1) this process is accordingly repeated for each column. If an empty *regionLock* matrix was given as input, the for-loop will skip without executing its statement. If the argument was not given at all, the variable will be created empty instead on line 25.

Finally, no matter the base shape (seen in 4.3), all genotypes, before being returned, are subjected to a mutation operation on line 27. This is done to introduce a basic diversity into the population as it is created. The details of this function will be discussed in section 4.5.

4.2.2 Running the genetic algorithm

The *evolveGA* function is the main function (or control function) of this genetic algorithm. Its role is to run the required operators in the proper order (by making the calls to said operators), while keeping track of the population and all parameters. Those are initialised right after the function starts. First the input argument *ga*, which is created by *setupGA.m* is processed as shown in the code listing 4.3.

Code 4.3: Processing of the input argument of evolveGA.m

```
1  global profile_type
2  genome=ga{1};
3  rVals=ga{2};
4  fit=ga{3};
5  generations=ga{4};
6  profile_type=ga{5};
7  individuals=size(genome,1);
8  crossoverRate=0.8;
9  mutationRate=0.3;
10 mutationStrength=0.6;
```

Line 1 sets up *profile_type* as a global variable, which is then assigned a value on line 6. The variables *genome*, *fit*, *individuals* and *generations* are identical to the respective variables of the same name in the *setupGA* function. *rVals* is a shorthand for the *regionValues*. The variable *individuals* is different in its initialization, as it is not simply read from the input argument but derived from *genome*.

Code 4.4: Storage of filial generation data in evolveGA.m

```
1  nextGen=zeros(max(size(genome,1),((crossoverRate*2)+
       mutationRate)),size(genome,2),size(genome,3));
2  nextFit=zeros(1,size(nextGen,1));
```

There are more variables being initialised, since those however pertain to specific function calls, their meaning will be mentioned in the discussion

of said functions. The last two variables necessary for the functionality of the core function are *nextGen* and *nextFit* (see 4.4). *nextGen* is similar in size and purpose to *genome*, as it stores the individuals genomes. The difference in size is the first dimension, as it diverges from the exact number of individuals in the population. Instead, the size matches the number of offspring expected to result from both the crossover and the mutation function, or the size of the population, depending on which number is bigger. The *nextFit* variable has a length equal to this first dimension of *nextGen* in order to store the fitness value of each individual accordingly.

Now that all variables relevant to the overall functionality of the genetic algorithm have been established, it is time to take a closer look at the actual algorithm. Due to the length of the code, which does not fit on a single page, it will be viewed in small cohesive segments which themselves are shown in the same order in which they appear in the actual code. Furthermore, let it be noted, that the code listings 4.10 to 4.15 are all nested within a for-loop with a variable i initialised as 1, incremented by 1 on each cycle and with an ending-condition of being equal to the variable *generations*.

Code 4.5: Crossover function call in evolveGA.m

```
1  selectedC=selection(zeros(2,crossoverRate),0,fit,
       genome);
2  for j=1:crossoverRate
3   [nextGen(j,:,:),nextGen(j+crossoverRate,:,:)]=
        crossover(shiftdim(genome(selectedC(1,j),:,:)),
        shiftdim(genome(selectedC(2,j),:,:)),rVals);
4  end
```

The first operation of the genetic algorithm is the crossover (see 4.10). All individuals taking part in this process in a generation are selected with the *selection* function call on line 1. As this function will be discussed in detail later on, an analysis of its inner functioning and input arguments will be omitted at this point. Let it however be noted, that due to its input argument, a two dimensional matrix containing two rows and a number of columns equal to the number of crossovers to be performed, will be returned to the variable *selectedC*. This matrix contains indices within the range of the population variable. The rest of the code listing loops through the number of crossovers, calling the function each time using the previously determined indices in a column to get the two parent individuals genome for each call. The returned filial individuals are stored in the previously set-up *nextGen* variable.

Code 4.6: Mutation function call in evolveGA.m

```
1  selectedM=selection(zeros(1,mutationRate),0,fit,
       genome);
```

```
2    for j=1:mutationRate
3     nextGen(j+(crossoverRate*2),:,:)=mutate({shiftdim(
          genome(selectedM(1,j),:,:)),rVals},
          mutationStrength);
4     totalMutations=totalMutations+1;
5    end
```

The mutation operation is called quite similarly to the crossover operation. Only two differences exist. First, the matrix of indices returned to *selectedM* is one dimensional since the operation acts on a single individual at a time. Second, a variable *totalMutations* sums up all mutations performed. This variable is not reset each generation. Its role will be discussed alongside the mutation function later on.

Code 4.7: Transition of generations in evolveGA.m

```
1    if ((crossoverRate*2)+mutationRate)<individuals
2     selectedS=selection(zeros(1,individuals-((
          crossoverRate*2)+mutationRate)),0,fit,genome);
3     nextGen(((crossoverRate*2)+mutationRate+1):end,:,:)
          =genome(selectedS,:,:);
4     for j=1:individuals
5      nextFit(j)=fitness(shiftdim(nextGen(j,:,:)));
6     end
7     genome=nextGen;
8     fit=nextFit;
9    elseif ((crossoverRate*2)+mutationRate)>individuals
10    for j=1:size(nextFit,2)
11     nextFit(j)=fitness(shiftdim(nextGen(j,:,:)));
12    end
13    selectedS=selection(zeros(1,individuals),0,nextFit,
          nextGen);
14    genome=nextGen(selectedS,:,:);
15    fit=nextFit(selectedS);
16   else
17    genome=nextGen;
18    for j=1:individuals
19     fit(j)=fitness(shiftdim(nextGen(j,:,:)));
20    end
```

Once the two key operations of the genetic algorithm are terminated, the transition to the next generation can begin. In simpler terms, the individuals from the *nextGen* variable (hence called filial generation) are transferred to the *genome* variable (called parental generation). However, as was already mentioned at the beginning of this section. The size of both does not

necessarily match. Three different cases can occur. First, the filial generation might be smaller than the parental one (code 4.7 lines 1 to 8). Second, both populations are of equal size (code 4.7 lines 16 to 20). Third, the filial generation can be larger than the parental one (code 4.7 lines 9 to 15).

In the first case, an index matrix *selectedS* is created with a length equal to the number of individuals of a population minus the number of individuals created by the crossover and the mutation. With this, the selected individuals are appended to the filial generation. This increases the size of the filial generation to match the parental one. Afterwards, the fitness values of all individuals in the filial generation are calculated. Once both the genotypes and the fitness values are properly determined, the parental generation can simply be overwritten by the filial generation.

The second case is indubitably the simplest. The genotype of the filial generation can directly overwrite the parental generation and the fitness values can immediately be stored in the *fit* variable as they are calculated.

The last case starts by calculating the fitness values of the filial generation as these are needed to create the index matrix with the *selection* function. With the newly created *selectedS* both, selected genotypes and their respective fitness values, are copied to the parental generation.

Generally speaking, this handling of the transition of the population between generations all but guaranties the survival of the fittest individuals of the filial generation. Only in the third case will some of filial individuals be lost. Who survives is then depending on the implementation of the selection operator. Parental individuals can only survive if they are selected in the first case. As selection operations are usually fitness dependent, the lose of parental individuals (or some filial ones) is not considered detrimental to the solution finding process as the fitter individuals are assumed to live on in the filial generation as product of crossovers and mutations.

Still, other approaches exist and could be used just as well. For example, both generations could be joined prior to the selection of individuals for the next parental generation.

4.3 Selection function

The selection of individuals, with or on which actions are to be performed, being a prerequisite for the other two key functions (crossover and mutation), the selection function will be the first to be covered.

The selection function (code 4.8) essentially fills the input matrix *selected* with indices, that are within the range of the population size, and then returns it.

Code 4.8: Source code of selection.m

```
1 function selected=selection(selected,selType,fit,
    genome)
```

```
2  if selType==0
3   if mean(fit)~=0
4    wheel=cumsum(fit)/sum(fit);
5    selected=arrayfun(@(x) find(x<=wheel,1,'first'),
         rand(size(selected)));
6   else
7    selected=selection(selected,1,fit,genome);
8   end
9  elseif selType==1
10  altFit=ones(size(fit));
11  selected=selection(selected,0,altFit,genome);
12 elseif selType==2
13  divVar=diversity(genome,fit);
14  selected(1,:)=selection(selected(1,:),0,fit);
15  for i=1:size(selected,2)
16   candidates=find(divVar(:,selected(1,i),:)==max(
         divVar(:,selected(1,i))));
17   selected(2,i)=candidates(randperm(size(candidates
         ,1),1));
18  end
19 else
20  selected=[];
21  error('selection: no matching selection type found')
       ;
22 end
```

Three distinct selection methods were implemented. An equal selection method (lines 9 to 11), a roulette wheel selection (RWS, lines 2 to 8), and a diversity based selection (DBS, lines 12 to 18). The type of selection used is determined by the second parameter of the function call *selType*. This parameter is expected to be an integer.

Lets first consider the most basic method: equal selection. This will be used if *selType* is set 1. As the name implies, all individuals have an equal chance of being selected by this method. Its main purpose is to serve as reliable selection function during the testing phase, though it could also be used as a safe fall back function. This method creates a one-matrix of the same size as *fit* called *altFit* and calls the selection function with the RWS as selection type. Due to the one-matrix, all individuals are given an equal fitness, therefore the odds of being selected by the RWS method are equal for all.

Mostly, the genetic algorithm uses the RWS. It is a very common fitness-dependent selection method that takes its name from the device used in the famous casino game of roulette. While the gambling device is divided in a number of equally large slots for the ball to fall into, the RWS normalizes

the fitness values of the population. Essentially giving each individual a slot with a size proportional to its fitness relative to the populations total fitness. The RWS per se is made up of two lines of code. Line 4 creates a digital equivalent (*wheel*) of the roulette wheel by calculating the cumulative sum for each fitness value in *fit* and then dividing each by the total sum of the fitness values. The employed functions *cumsum* (to calculate cumulative sums) and *sum* (to calculate total sums) are both Matlab functions. Sticking to the analogy, line 5 is akin to throwing the ball into the roulette wheel, i.e. choosing individuals from *wheel*. This is done through *arrayfun*, a Matlab function, that applies a "function to each element of (an) array" (to quote its Matlab help text). Usually the first input argument expected by this function is a function handle. That is, the name of a function to apply (as opposed to a function call, which includes parenthesis potentially containing arguments). All following arguments are assumed to be matrices of equal size (as they are all iterated through at the same time) on whose contents the function is to be applied. For each input argument expected by the function, one matrix is expected. In this case the *find* function is to be applied. Each element of the array is be compared to *wheel* and only the index of the first element of *wheel* to be smaller or equal to the element is to be returned. This function call however requires three additional variables beside the array element on which to operate: *wheel*, 1 and the string 'first'. Furthermore, all those additional variables are identical for each iteration of the operation. While for the later two arrays, of the same size as the array on which the function operates, could be created, containing at every position the same value (1 and 'first' respectively), handing *wheel* to the function wouldn't be solvable this easily as it is not needed as a separate input argument, but as part of the comparison in the first argument. All of this can however be circumvented by declaring an anonymous function in-line as the first argument of *arrayfun*. As this is not trivial and not part of the expected basic understanding of Matlab syntax, it shall therefore be explained. The at sign (@) declares the following to be an anonymous function (Matlab Documentation defines this as follows: "An anonymous function is a function that is *not* stored in a program file, but is associated with a variable whose data type is *function_handle*."). It is followed by parenthesis in which a comma-separated list of parameter names are given (in this case just one name x). Separated by space, the body of the function is given. The x name stands in for input variables. Variables that are not defined in the list of input names are inherited from the scope of the definition. That means that variables like *wheel* are stored as part of the function when it is defined and in the state it was in at the time of definition. If the variable was to be modified or even deleted, it would still exist and be usable within the anonymous function. The values 1 and 'first' are simply fixed values as they would be in any regular function. Thus an anonymous function taking one input argument, comparing it to *wheel* and returning the one first index smaller

or equal is defined. Since it is defined inside the function call of *arrayfun*, it will only exist within this function instance. Getting back to this original function call and its second argument, a matrix of "uniformly distributed pseudorandom numbers" (Matlab help text of *rand*) of size equal to *selected* is created (the number of columns correlate to the number of operations to perform, the rows permit the selection of multiple individuals for a same operation, e.g. partners for crossover). So each of these random numbers is compared to *wheel*. Since the number are uniformly distributed, they have equal chances to be any number of the interval $]0;1[$. By comparing the numbers to *wheel* they are figuratively arranged into slots of varying sizes (relative fitness), the normalized cumulative fitness' acting as limits between slots. Each of these slots has an index corresponding to the individual whose fitness is represented, which is then returned.
Beside this common case, there is also a second possible process. The first case relies on fitness values. Should there, however, be no fitness with a value greater than 0, *wheel* will fail to create (due to an attempt to divide by zero). To prevent this error from stopping the program, the mean fitness is checked. If it is equal to zero the *selection* function is called again, but with a different selection type (the fall back type: equal selection). Only a non-zero mean fitness will lead to a proper execution of a roulette wheel selection.

The last implemented selection method is one based on diversity (lines 12 to 18). The aim of this process is to increase the diversity within a population by selecting two genetically differing individuals for a crossover operation. To achieve this, a matrix of diversity values is created by the function *diversity* in *divVar*. Next, the first row of *selected* is filled by the previously covered RWS. For each index in the first row, the appropriate column in *divVar* is subjected to the *find* function, looking for the maximum value. The first result is saved to the second row of *selected*.

4.3.1 Assessing diversity

Code 4.9: Source code of diversity.m

```
1 function diversity=diversity(genome,fit)
2  diversity=zeros(size(genome,1));
3  a=1.25;
4  b=1.75;
5  for i=1:size(genome,1)
6   for j=1:size(genome,1)
7    buff=abs(mean((genome(i,1,:)-genome(j,1,:)).^2 + (
       genome(i,2,:)-genome(j,1,:)).^2));
8    if isnan(buff)
9     diversity(i,j)=0;
```

```
10      else
11        diversity(i,j)=buff*(fit(i)^a)*(fit(j)^b);
12      end
13    end
14  end
```

If a partner for the crossover operation is to be selected based on the diversity of the individuals of the population, a measurement of diversity has to be defined first. The main part of the diversity is shown by and saved to the temporary variable *buff* on line 7. It is essentially the absolute value of the mean of the squared euclidean distance of two individuals. However solely basing the selection on the difference between individuals might lead to pairing of highly fit individuals (as previously shown the first partner is selected by RWS) with individuals that although very different, are considered non-viable based on their fitness. To keep a degree of influence of the fitness, the actual diversity value *buff* is multiplied by the fitness values of the two individuals involved, each exponentiated by a factor (*alpha* for the first individual and *beta* for the second one). The separated exponents allows for varying emphasis of the two fitness values.

Through two nested for loops, *diversity* iterates through all combinations of individuals. The results are returned in a square matrix *diversity* of the same height and length as the *genome* is created.

4.4 Crossover function

Code 4.10: Source code of crossover.m

```
1  function [genotype1,genotype2]=crossover(genotype1,
       genotype2,rVals,switches)
2   if nargin<4
3    switches=2;
4   end
5   if ~isempty(rVals)
6    lock1=getLock(genotype1,rVals);
7    lock2=getLock(genotype2,rVals);
8    whitelistS=[1:size(genotype1,2)];
9    whitelistE=whitelistS;
10   for i=1:size(lock1,2)
11    rL1=[lock1(1,i):lock1(1,i)+lock1(2,i)-1];
12    rL2=[lock2(1,i):lock2(1,i)+lock2(2,i)-1];
13    for j=1:size(rL1,2)-1
14     whitelistS(find(whitelistS==rL1(j+1)))=[];
15     whitelistS(find(whitelistS==rL2(j+1)))=[];
16     whitelistE(find(whitelistS==rL1(j)))=[];
```

```
17        whitelistE(find(whitelistS==rL2(j)))=[];
18      end
19    end
20    whitelistS(end)=[];
21    position=zeros(1,switches+mod(switches,2));
22    for i=1:2:switches
23      pS=randperm(size(whitelistS,2),1);
24      pE=randperm(size(whitelistE(find(whitelistE>=
            whitelistS(pS),1,'first'):end),2),1)+find(
            whitelistE>=whitelistS(pS),1,'first')-1;
25      position(i)=whitelistS(pS);
26      position(i+1)=whitelistE(pE);
27      whitelistS(pS)=[];
28      whitelistE(pE)=[];
29      if whitelistS(end)==whitelistE(end)
30        whitelistS(end)=[];
31      end
32    end
33  else
34    position=randperm(size(genotype1,2),switches);
35    if mod(switches,2)~=0
36      position(end+1)=size(genotype1,2);
37    end
38  end
39  for i=1:2:switches
40    buffer=genotype1(:,position(1,i):position(1,i+1));
41    genotype1(:,position(1,i):position(1,i+1))=genotype2
          (:,position(1,i):position(1,i+1));
42    genotype2(:,position(1,i):position(1,i+1))=buffer;
43  end
44  if ~checkUnique(genotype1)||~checkUnique(genotype2)
45    [genotype1,genotype2]=uniqueT(genotype1,genotype2,
          position);
46  end
47  genotype1=sectionSolver(genotype1,rVals);
48  genotype2=sectionSolver(genotype2,rVals);
```

As previously mentioned, the crossover function (code listing 4.10) seeks to emulate the breeding of parent organisms. Or to be more specific, the exchange of genetic information between individuals to create offspring. Hence, the key input arguments are the genotypes of the chosen parent individuals. The variable *rVals* is a cell array containing the values expected in the protected segments. The last input variable *switches* gives the number of transposition points to be used during the crossover operation. If not value

is given, *switches* defaults to 2, based on findings of [13] (also favoured by [34]).
There are other implementations beside multi-point crossover, like uniform crossover [40], punctuated string ([39], [19], or [23]), or the partially matched crossover ([14] or [13]). Seemingly there there is not one clearly superior version ([22]), so the choice was made based on personal choice.

The crossover operation first determines the points of the genome at which the transposition will take place. To do this, two cases are distinguished: regionLocks are used or they are not used. As, for example, the first point belonging to a protected segment may be used as the first point of a transposed segment without cutting it off from the rest of the protected region, it can never be used as the last point of a transposition. The inverse is also true. The last point of a protected segment can be used as the last point of a transposition, but not as the first. Therefore, special attention is needed for the selection of the transposition points in the first case. First the exact positions of the *regionLocks* are determined by the *getLock* functions (lines 6 and 7). Next, two so called white-lists are created containing the indices of all positions in the genome, one for potential starting points of transpositions and one for potential ending points. A for-loop iterates through the region locks *lock1* and *lock2*. Since those are encoded only by their respective starting points and length, they are translated into full lists of indices they cover *rL1* and *rL2*. A nested for-loop iterates through those lists, removing their content from the white-lists (except for the first position for the list of starting points and the last position for the list of ending points). With all these preparations done, the array for the actual transposition points *position* is created as zero-matrix of the length *switches* (if this is an odd number, an additional position is added since the transposition is always performed between a pair of points). For every two points to be chosen, one is picked from the starting white-list *whitelistS* and one from the ending white-list *whitelistE* by the function *randperm*. The points are stored in *position* and removed from their respective list, thus preventing the same point from acting repeatedly as starting or ending point (performing both roles once however is still possible). Also, the last point of *whitelistS* is removed in the original list, as well as in the loop determining the chosen positions if it is equal to the last point of *whitelistE*, as the selection of this point would not leave a second point to set the end of the segment to be switched.
In the second case, the creation of a *whitelist* is forgone, as all indices of the genotype are valid for the transfer of genetic information. A number of points equal to *switches* is chosen by *randperm*. If this number is odd, the last index of the genome is appended to guarantee that the points can be called in pairs.

For each set of indices in *position*, the set of values between them from one genotype is copied to *buffer*. True to its name, its sole function is to

temporarily keep a copy of the information, while the the segment of the first genotype is overwritten by the information held within the same segment of the second genotype. Afterwards the content of *buffer* replaces the segment of the second genome, hence completing the transfer of genetic information between the two genotypes.

The following if-statement (lines 44 to 46) concerns itself with the correction of a rarely occurring problem of redundant points in the genotype that can potentially lead to problems with the advanced algorithm. Points are considered redundant only if another point in the genotype matches both the abscissa and the ordinate.

To get an idea how rare this occurs, a small test has been devised. Three so called base-populations, of 30 individuals each, have been created by *setupGA*. Each of these populations was evolved for 1000 generations (with a crossover-rate of 0.5 and a mutation-rate of 0.1; for information on these rates see section 4.8) and the amount of crossovers as well as the number of crossovers resulting in non unique individuals have been recorded. This test was performed thrice (using the same three base-populations every time) to get some comparable data, statistically speaking. Despite a total of 15000 crossovers performed during *evolveGA* (which adds up to 135000 crossovers for three base-populations tested thrice each) not a single case of genotypes containing redundant points occurred. This of course does no mean, that such cases do not exist. It merely demonstrates the rarity of the event. To prevent problems down the line, the problem is dealt with where it arises. When one of the genomes created by the crossover operation is found to contain duplicates by the *checkUnique* function, both genomes as well as the index list *position* are given to the function call *uniqueT*. Its purpose is to undo the switches that caused the duplicates to appear.

Before returning the newly created offspring genomes, *sectionSolver* checks them for intersections and tries to resolve them (for details see chapter A.1.1). While intersection lines result in a longer circumference and therefore can be expected to be attributed a lower fitness value, thus leading them do disappear over time, the advanced genetic algorithm is unable to handle them. This is why they are eliminated at this point.

To demonstrate the efficiency of the crossover operator, three base-populations created by *setupGA* are evolved. For every crossover performed, the fitness values of the resulting genomes are compared to those of the parent genomes. The crossovers resulting in an improved fitness for the filial genomes relative to the parents are counted. This test is performed three times. The results, as percentage of total crossovers, of each repetition are averaged (results shown in table 4.1).

An approximate improvement rate of 80% can indubitably be considered a tremendous success.

Table 4.1: Average percentage of crossovers resulting in an improved fitness value of offspring relatively to its parents over 1000 generations (with 15 crossovers per generation), and mean of the values.

Base-population	Crossovers improving fitness-values (%)
1	80.24
2	79.57
3	80.27
Mean	80.03

4.4.1 Ensuring unique coordinates

Code 4.11: Source code of checkUnique.m

```
1  function bool=checkUnique(a,point)
2   if nargin<2
3    bool=true;
4    for i=1:size(a,2)
5     point=i;
6     x=find(a(1,:)==a(1,point));
7     y=find(a(2,:)==a(2,point));
8     if size(intersect(x,y),2)>=2
9      bool=false;
10    end
11   end
12   else
13    bool=true;
14    x=find(a(1,:)==a(1,point));
15    y=find(a(2,:)==a(2,point));
16    if size(intersect(x,y),2)>=2
17     bool=false;
18    end
19   end
20  end
```

As previously mentioned, the *checkUnique* function (code listing: 4.11) checks an input genome (input argument a) for duplicate points: matching tuples.

A second argument *point* can be given, which will lead to the use of the second approach on the lines 12 to 18. In this case only the tuple at the position given by the argument is checked for duplicates. In most cases however the first approach on the lines 3 to 11 is used which checks the point coordinates at each position of the input genome. The process for finding any duplicates of a tuple is simple. Separately all positions with

identical values to the abscissa and ordinate respectively are sought out. If the two lists have two or more points in common (determined by the built-in function *intersect*), the reference point is not unique. One common position is of course expected, as the reference point obviously matches itself.

4.4.2 Correcting duplicates

Code 4.12: Source code of uniqueT.m

```
1  function [a,b]=uniqueT(a,b,position)
2   for i=1:2:size(position,2)
3    insertInA=a(:,position(i):position(i+1));
4    insertInB=b(:,position(i):position(i+1));
5    restA=[a(:,1:position(i)-1) a(:,position(i+1)+1:end)
         ];
6    restB=[b(:,1:position(i)-1) b(:,position(i+1)+1:end)
         ];
7    if size(intersect(insertInA',restA','rows'),1)>0||
         size(intersect(insertInB',restB','rows'),1)>0
8     a(:,position(i):position(i+1))=insertInB;
9     b(:,position(i):position(i+1))=insertInA;
10   end
11  end
```

The *uniqueT* functions sole purpose is to remove duplicate points from a genome that arose through the crossover operation.

As, when this function is called, it is already known that at least one of the genomes contains duplicates, it is only a question of finding out which of the segments switched during crossover contains them. Therefore a for-loop cycles through all segments (as with the for-loop in *crossover* one point of the position list is considered the start of a segment while the following one is considered the end and the loop iterates with a step size of two instead of the default of 1). During each cycle, the segment transferred during the original crossover is copied into a temporary variable. The segment from genome *a* into *insertInA* and the segment from genome *b* into *insertInB*. A second set of variables, named *restA* and *restB*, is created which contain the remaining content of the genomes *a* and *b* respectively. Similarly to how *checkUnique* finds duplicates, common points between the corresponding *insertIn** and *rest** sets a identified through the *intersect* function. Since however the segment is not compared to the full genome, but to the genome sans itself, self-hits (false positives from a match to its own values) cannot occur and hence even a single match already indicates a duplicate. If a match is found for either genome, the segments excised from the genomes are used to overwrite the corresponding block of the respectively other genome.

4.4.3 Finding region lock positions within a genome

Code 4.13: Source code of getLock.m

```
1  function rLock=getLock(g,rVal)
2   if isempty(rVal)
3    rLock=[];
4   else
5    rLock=zeros(2,size(rVal,2));
6    for i=1:size(rVal,2)
7     xS=find(g(1,:)==rVal{i}(1,1));
8     xE=find(g(1,:)==rVal{i}(1,end));
9     yS=find(g(2,:)==rVal{i}(2,1));
10    yE=find(g(2,:)==rVal{i}(2,end));
11    mS=intersect(xS,yS);
12    mE=intersect(xE,yE);
13    if ~isempty(mS)&&~isempty(mE)&&(all(all(g(:,min(mS,
          mE):max(mS,mE))==rVal{i}))||all(all(g(:,max(mS,
          mE):-1:min(mS,mE))==rVal{i})))
14     rLock(1,i)=min(mS,mE);
15     rLock(2,i)=size(rVal{i},2);
16    end
17   end
18  end
```

The purpose of the *getLock* function is to find the starting positions of all protected segments within a genome. To do this, two input arguments are required: the genome g and the values of the region locks $rVal$. If $rVal$ is empty, no protected segments were defined, hence an empty matrix $rLock$ is created and returned. If the variable is not empty, $rLock$ is initialised as a zero-matrix of height 2 and length equal to that of $rVal$. A for-loop iterates through the cell array $rVal$ containing the matrix with the values of each protected segment. Each time, matches for the first and the last point of $rVal$ are sought in the genome. This is performed separately for the abscissa and ordinate coordinates.The intersection of matches for both axes makes sure only those points are kept, that completely fit the given model. To determine if the entire sequence matches the conditional statement on line 13 must be true. To avoid errors, the intersection arrays mS and mE are checked to be not-empty. The matching of the full sequence is attempted in both directions. If successful in either case, the starting point is saved in the first row of $rLock$ and the length of the region in the second one.

4.5 Mutation function

Code 4.14: Source code of mutate.m

```
1  function genome=mutate(genInfo,deviation)
2   if nargin<2
3    deviation=0.001;
4   end
5   mRange=deviation*20;
6   genome=cat(1,genInfo{1,1});
7    maxVal=ones(size(genome))*Inf;
8    minVal=ones(size(genome))*-Inf;
9    rVals=cat(1,genInfo{1,2});
10   regionLock=getLock(genome,cat(1,genInfo{1,2}));
11  for i=1:size(regionLock,2)
12   maxVal(:,regionLock(1,i):regionLock(1,i)+regionLock
         (2,i)-1)=rVals{i};
13   minVal(:,regionLock(1,i):regionLock(1,i)+regionLock
         (2,i)-1)=rVals{i};
14  end
15  cU=0;
16  while cU==0
17   modifier=mRange*randn(size(genome));
18   newValue=genome+modifier;
19   newValue(newValue>maxVal)=maxVal(newValue>maxVal);
20   newValue(newValue<minVal)=minVal(newValue<minVal);
21   if checkUnique(newValue)
22    cU=1;
23   end
24  end
25  genome=sectionSolver(newValue,rVals);
```

The *mutate* function, like mutations in nature, acts as an operation of randomization. Thus introducing new points, increasing diversity in the population, and acting as the main source of change in the genome ([17]). This is both so important and powerful, that there is a type of algorithm essentially consisting of a genetic algorithm without a crossover operator (so just the mutation operator), sometimes called naive evolution, that can at times perform quite well ([10], [11], or [11]). The genome upon which this function acts is contained in the mandatory variable *genInfo*. The second variable *deviation* is optional and, if not given, will be set to 0.001 by the conditional statement on the lines 2 to 4. Essentially, it simply acts as a multiplication factor on line 5, when setting up the *mRange* variable. This variable determines the range in which random numbers are generated to modify the genome and is defines as the product of the *deviation* factor and the number 20, which in turn was chosen as it represents the length in units of the square on which all individuals are based when considering the closed

profile. Before actually mutating the genome, the *regionLock* has to be checked and if necessary, translated into an adequate format. The *genInfo* variable is in fact a cell array containing the genome in its first position and the $rVals$ in a second position. For ease of access and the readability of the code, the genome to be mutated is copied into a variable *genome* (line 6). On line 10, $rVals$ is extracted from *genInfo* and *regionLock* is determined on the following line. Before that two one-matrices $maxVal$ and $minVal$ are created and multiplied by separate factors (in the default function by Inf and $-Inf$ respectively). Those act as borders for the search space by respectively determining the maximum and the minimum value for each position in the genome. The following for-loop, from line 11 to 14, replaces the values of both of these matrices by the values from $rVals$ at the positions determined by *regionLock*. The values of the protected values can exceed those set by the factor during the creation of the border matrices. Through a while-loop, checking for the boolean variable cU, a mutated genome without duplicates is created. First a matrix containing values by which the genome is to be modified is generated by multiplying the *mRange* by uniformly distributed random numbers (or, to be precise and quote the help reference of Matlab :" Uniformly distributed pseudorandom numbers", an important distinction for computer scientists and mathematicians, but merely a question of accuracy of the dissertation in this case). Adding the thus created *modifier* matrix to the genome creates a potential new genome *newValue*. Before the uniqueness of its coordinate tuples can be verified in the conditional statement from line 21 to 23, which would then set cU to TRUE, thereby causing the while-loop to terminate, the limits have to be applied. On the lines 19 and 20 all values exceeding the maximum value or falling below the minimum value defined for their respective position by the border matrices are replace by these maxima and minima. As the positions defined by the *regionLock* contains the values of the protected segments in both of the border matrices, any deviation will either exceed or undershoot these values and be reset. Before the new genome can be returned, it is used as an input parameter while calling the *sectionSolver* function (for more information on this functions, see A.1.1).

Before turning the readers attention elsewhere, there is one functionality or feature related to the mutation function that has to be dealt with. This resides in *evolveGA.m*. As previously shown, one of the input arguments of the mutation function, labelled as *deviation* within the function, can have a tremendous effect on the range of numbers that can be generated to modify a given genotype. The following dynamically changes this parameter over the course of many generations during runtime. The idea behind this is essentially based of the rule of the fifth developed by [35]. Said rule professes (backed by empiric data), a slight boost in the performance of genetic algorithms by an adaptation of the mutation rate over the course of time. If the ratio of successful mutations to total mutations exceeds a

specific threshold over a certain number of generations, it is assumed, that bulk of the individuals is converging to quickly toward a local maximum at the cost of exploring the fitness landscape which might hold greater peaks still. To expand on the exploration, the mutation strength should then be increased. Conversely, when same ratio falls short of the threshold, the mutation strength should be decreased to reduce the reach of the mutation operation (the distance between input and resulting genotype) avoid overshooting peaks and, so to speak, narrow down the search. Rechenberg et al., in their work, determined the optimal ratio of successful mutation to be one fifth of the total mutations performed during a selected time frame. Hence the name 'rule of the fifth'.

In this implementation, the ratio is simply gathered from already saved data. By iterating over the *nextFit* matrix slots in which the fitness values of the genotypes resulting from mutations are stored, those can be compared to the values of the original genotypes, which are accessed via the indexes listed in *selectedM*. The total amount of mutations *totalMutations* is already being increase by one after each call to the *mutation* function on line 4 (Code 4.6). While iterating over the fitness values of genotypes resulting from mutations, a difference in favour of the resulting genotype while lead to an increment of the counter *improvedMutations*.

The time between changes of the mutation strength are set by the variables *mutEvoNext* and *mutEvoStep* (see Code 4.15), which are established right after all other variables within *evolveGA* following the code block shown in 4.3. *mutEvoNext* denotes the generation during which the next modification of the mutation strength is to take place. *mutEvoStep* sets the time in generations between changes.

As defined by line 6 (Code 4.15), whenever the current generation counter i matches *mutEvoNext*, the adaptation of the mutation strength shown from line 7 to 15 takes place. First *mutEvoNext* gets redefined by adding *mutEvoStep* to it. The ratio of successful mutations to the total amount is calculated and saved as *successRate*. If more than 20% of the mutations are successful, the mutation strength is increased by dividing the current value by the sum of *mutEvoFactor* (a variable defined alongside *mutEvoNext* and set to 0.75) and a random number that lies between 0 and $1 - mutEvoFactor$. Thus the divisor is garantied to lie between the *mutEvoFactor* and 1. If less than, 20% of the mutations are successful, the formula is very similiar, only instead of dividing the current mutation strength, it is multiplied by the partially randomized factor. The last action of this code segment is to reset both *improvedMutations* and *totalMutations*.

Code 4.15: Adaptation of the mutation in evolveGA.m

```
1       fit(j)=fitness(shiftdim(nextGen(j,:,:)));
2     end
3   end
```

```
4    for j=(crossoverRate+1):(crossoverRate+mutationRate)
5     if nextFit(j)>fit(j-crossoverRate)
6      improvedMutations=improvedMutations+1;
7     end
8    end
9    if i==mutEvoNext
10    mutEvoNext=mutEvoNext+mutEvoStep;
11    successRate=improvedMutations/totalMutations
12    if successRate>0.2
13     mutationStrength=min(mutationStrength/(
           mutEvoFactor+(rand(1)*(1-mutEvoFactor))),1);
14    elseif successRate<0.2
15     mutationStrength=mutationStrength*(mutEvoFactor+(
           rand(1)*(1-mutEvoFactor)));
16    end
```

Code 4.16: Variables defining the adaptation of mutations in evolveGA.m

```
1  crossoverRate=0.8;
2  mutationRate=0.3;
3  mutationStrength=0.6;
```

4.6 Fitness function

As a reminder, the goal of the genetic algorithm is to decrease the weight of beams for industrial racks while still meeting stability criteria. Assuming that the material and the thickness, as well as the length of the beam are constant, the weight can only be reduced if the circumference (for closed profiles) or the length (for open profiles) of the profile is reduced. This is calculated by the function *schwerpunkt* and returned as l alongside the center of mass s. To determine the stability of a profile, the function *profeval* provided by Prof. Dr. Werner Baumgartner is used (only minor changes and amendments were performed). This function uses the *ftm* function(a version also provided by Prof. Dr. Werner Baumgartner to provide functionality beyond the calculation of the second moment of area) to calculate the second moment of area. Additionally it calculates the section modulus of torsion and the modulus of resistance. These are compared to given minimal values. Those exceeding the minimal values will return a positive result, otherwise the returned result will be negative. To avoid losing good candidates, that only missed the reference values by a narrow margin while showing a short circumference or length, a generalised logistic function *sigmoid* is used to smoothen the transition between barely missing the reference values and successfully fulfilling the stability criteria. This function also turns this

stability check into a value between 0 and 1. While all successful profiles receive the value 1, near misses are assigned a value between the two, and clear misses a value of 0. The product of these values is calculated and used as a factor modifying the inverse of the circumference (the key value of the fitness). If the stability requirements are fulfilled, this factor will not influence the key value. However, if some requirements miss their target, the product will penalise the key value.

Code 4.17: Second moment of area based fitness function

```
1 function fit=fitness(genome,long,force,filename)
2   [s,l]=schwerpunkt(genome);
3   fit=(1/l)*prod(sigmoid(profeval(genome,0.2)))*100;
```

4.6.1 Geometric data from *schwerpunkt*

The purpose of this function is to calculate both, the coordinates of the center of mass (*pos*) and the circumference or length (L) of a profile given by a set of coordinates as the parameter g.

Let it be noted that, except for the code on the lines 2 to 5 in listing 4.18, the entire function, as seen in the listing, was provided, courtesy of Professor Doctor Werner Baumgartner.

Code 4.18: Function to determin the circumference and center of mass of profiles *schwerpunkt.m*

```
1  function [pos,L]=schwerpunkt(g)
2   global profile_type
3   if ~profile_type
4    g=g(:,[1:end 1]);
5   end
6   n=length(g)-1;
7   L=0;
8   pos=[0;0];
9   for i=1:n
10   Li=sqrt(sum((g(:,i)-g(:,i+1)).^2));
11   L=L+Li;
12   pos=pos+Li*(g(:,i)+g(:,i+1))/2;
13  end
14  pos=pos/L;
```

The global variable *open* (line 2) was sufficiently covered in section 4.2.1. Of greater interest to us, is its use in the conditional statement on the lines 3 to 5. The condition itself is the negation of *open*, which is used here as if it was a boolean variable, even though the actual data type is the default 'double' as it wasn't specified during the declaration. Therefore the body

of the statement will only be executed when the genetic algorithm is used to generate closed profiles. This body, which only consists of line 4, simply overwrites the input variable g with a copy of itself to which the first column (the coordinates of the first point) have been appended.

First off the end value n for the following for loop is calculated as the largest dimension of the input g minus 1. This will lead to an iteration through all points of the genome but the last. After this, the output variables are initialised with values of 0. Inside the for loop, the length of each segment Li is calculated as the distance between the point currently targeted by the iteration and the following one. This distance is calculated as the square root of the sum of the squares of the differences of the abscissa and ordinates respectively (see formula 4.1). The length of the segments is added to the total length L, giving the full circumference once the loop is finished (formula 4.2).

$$L_i = \sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} \tag{4.1}$$

$$L = \sum_{i=1}^{n} L_i \tag{4.2}$$

The coordinates for the center of mass of the profile is calculated by summing the weighted centroids of each segment. The formula for the centroid (the geometrical center of a figure) for a line is given in formula 4.3. This centroid has to be multiplied by its weight. And even though the material and thickness of the profile are unknown within the scope of this function, we can do so. Since those parameters are identical for all segments of the profile, the can be cancelled out, leaving the length of the segment as the sole multiplicand for its weight. Summing up the weighted centroids gives the center of mass of the profile.

For closed profiles, the appended first coordinates ensure that the last coordinates are iterated through and the segment between that point and the first one is considered in the calculations.

$$C_x = \frac{x_1 + x_2}{2} \qquad C_y = \frac{y_1 + y_2}{2} \tag{4.3}$$

4.6.2 The second moment of area *ftm*

This section covers the original *ftm* function developed by the author (as opposed to the one provided by Prof. Dr. Werner Baugmartner) towards the end doctorate. All formulas within this function (code 4.19) pertaining to the calculation of the second moment of area where taken from [16].

The function expects two mandatory, as well as one optional input parameters. c contains the genome of a single individual. s contains the coordinates of the center of mass. b contains the thickness of the beam-wall

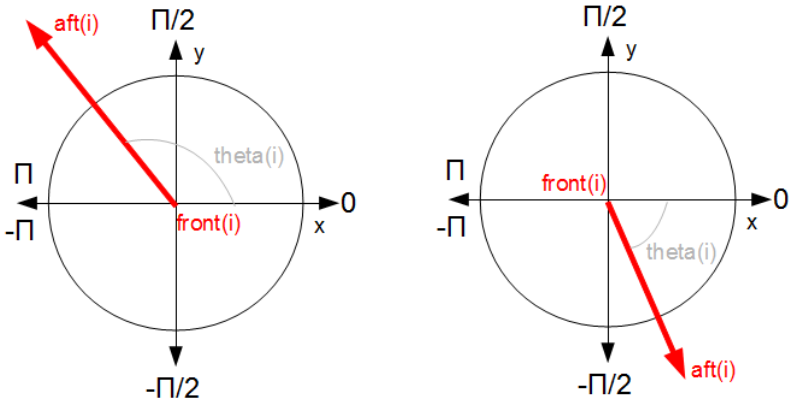

Figure 4.4: Diagram of the four quadrants of *atan2*. Left: example for a segment (red vector) with a positive angle *theta*. Right: example for a segment (red vector) with a negative angle *theta*. Note that the angle in the first point defined by *front* is calculated. The function *atan2* returns values in the interval $[-\pi; \pi]$.

and is optional. If it is not given, it is set to 0.2 by the conditional statement on the lines 3 to 5.

After defining n as the size of the second dimension of the genome (this essentially is equivalent to using the *length* function of Matlab in this case, c.f. code 4.18 line 6), the variables *front* and *aft* are defined. The specifics of those definitions depend on whether the profile to be optimized is open or closed, as defined by the global variable *open*. These two variables are matrices containing series of indices. in both cases the indices in the*front* matrix are shifted by 1 position compared to those in the *aft* matrix. For closed profiles, the first position is appended to the *front* matrix, but not for open profiles. This again defines an additional segment of the profile between the last and the first position, which is only present for the closed profiles.

Next, a series of geometric properties of the segments are calculated. dx and dy are the distance between two consecutive points (forming one segment respectively) on the abscissa and on the ordinate respectively. d determines the absolute distance between those points by calculating the square root of the sum of the squares of the distances in each of the two axis directions. If the formula for dx and dy where directly written into d instead of the variables, the complete formula would correspond to the on shown in 4.1. A simply calculates the surface area of each segment by multiplying the previously determined length by the thickness of the beam wall. *theta* is a matrix containing the angles in radians between each segment and a fictive horizontal line passing through the respective first point of each segment (as defined by the order of the points encoded in the genome). The *atan2*

function itself is part of Matlab and, according to its description in the 'help' text, returns the "four-quadrant inverse tangent (arctangent)". The variables*ys* and *zs* respectively calculate the distance on the abscissa and the ordinate between the closest point to the centroid of the segment that is on its surface and the center of mass. The formulas can be generalized as seen in formula 4.4. C_x and C_y within it, refer to formula 4.3, and s_x and s_y denote the respective coordinate of the center of mass. So far undefined are the elements w_x and w_y.

$$ys = C_x - w_x - s_x \qquad zs = C_y + w_y - s_y \tag{4.4}$$

These account for the distance between the centroid of a segment and its surface. Although it is known that the direct distance is equal to $b/2$, the segments are tilted in various angles (*theta*), therefore the distance within the reference frame of the Cartesian coordinate system used by the genome, and by extension by the center of mass, must be properly calculated. Figure 4.5 gives a more graphical explanation of the geometry involved. Assuming that the inner surface of the profile wall is a line parallel to $\overline{AB}$ through C and that in the given example of the diagram, the center of mass s is positioned somewhere to the lower right of the segment, the distance on the abscissa is equal to the distance to M, minus the distance $\overline{MD}$, and the distance on the ordinate is equal to the distance to M minus the distance $\overline{DC}$. Both of those distances can be calculated. To this point the known variables in the diagram are $\overline{MC}$ (which is equal to $b/2$) and τ (which has been calculated in *theta*). Through a series of complementary and corresponding angles detailed in the diagram caption, the angle τ'' in $\widehat{MCD}$ can be determined to be equal to *theta*. Also, as established during the geometric construction, the angle $\widehat{MDC}$ is a right angle. Hence determining the length of the remaining sides of the triangle MDC boils down to a simple trigonometry problem solved by formula 4.5. w_x solves the distance $\overline{MD}$ and w_y the distance $\overline{DC}$.

$$w_x = sin(theta) * (\frac{b}{2}) \qquad w_y = cos(theta) * (\frac{b}{2}) \tag{4.5}$$

On line 21, *theta* is modified to fit the different reference frame expected by the following formulas.

Subsequently the second moment of area can be calculated in three distinct steps. First, based on the formula for the second moment of area for rectangles ([16], table 4.1), the value for each segment is calculated assuming the centroid to be its center of mass (lines 23 to 25). Secondly, with the previous results, a second moment of area is calculated accounting for the respective tilt of each segment (lines 27 and 28 according to [16] formula 4.14). Thirdly, the previous results are used to calculate a second moment of area of each segment accounting for the segments displacement relative

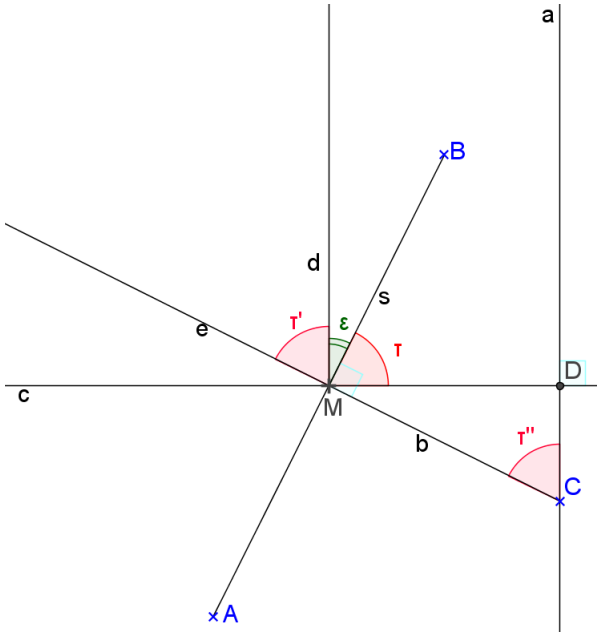

Figure 4.5: Diagram of the geometric construction underlying formula 4.5. $\overline{AB}$ or s: segment of the profile. M: centroid of $\overline{AB}$. $\overline{MC}$ or b: half wall thickness (towards center of mass). c: auxiliary horizontal line through M. a: auxiliary line through C, perpendicular to c. d: auxiliary ray from M, parallel to a. e: auxiliary line extending b. τ: angle *theta*. ϵ: complementary angle to τ. τ':complementary angle to ϵ (due to the right angle between e and s).τ'': corresponding angle to τ' as a and d are parallel and e is an extension to b.

to the center of mass of the profile (lines 31 and 32 according to [16] formula 4.13).These last second moments of area from each segment are summed up and returned as two values. *iy* gives the second moment of area in the direction of the abscissa, while *iz* gives the second moment of area of the ordinate.

It should be noted, that this procedure yields a relatively good approximation, but not exact results. Since segments expand (based on wall thickness) perpendicularly to the line connecting the points defining the segment, there will almost always be some overlapping between two segments close to the point involved in the definition of both. This error was accepted for the sake of development- and runtime-speed as an exact function, able to handle any random profile given, would prove to be much more complex to write and to calculate.

Therefore this function is to be considered a sufficiently precise approximation to reveal optimization potential, if any exists, but makes no claim to be physically accurate.

Code 4.19: Second moment of area *ftm.m*

```
1  function [iy,iz]=ftm(c,s,b)
2   global profile_type
3   if(nargin<3)
4    b=0.2;
5   end
6   n=size(c,2);
7   if ~profile_type
8    front=[2:n 1];
9    aft=(1:n);
10  else
11   front=(2:n);
12   aft=(1:n-1);
13  end
14  dx=c(1,front)-c(1,aft);
15  dy=c(2,front)-c(2,aft);
16  d=sqrt(dx.^2+dy.^2);
17  A=d*b;
18  theta=atan2(dy,dx);
19  ys=((c(1,front)+c(1,aft))/2)-sin(theta).*(b/2)-s(1);
20  zs=((c(2,front)+c(2,aft))/2)+cos(theta).*(b/2)-s(2);
21  theta=theta-(pi/2);
22  Iy=A.*(d.^2)/12;
23  Iz=A.*(b^2)/12;
24  Iyz=0;
25  Iy_g=0.5*(Iy+Iz)+0.5*(Iy-Iz).*cos(2.*theta)+Iyz.*sin
        (2.*theta);
```

```
26  Iz_g=0.5*(Iy+Iz)-0.5*(Iy-Iz).*cos(2.*theta)-Iyz.*sin
        (2.*theta);
27  Iy=Iy_g+A.*(zs.^2);
28  Iz=Iz_g+A.*(ys.^2);
29  iy=sum(Iy);
30  iz=sum(Iz);
```

For completeness, the function provided by Professor Baumgartner is shown in the code listing 4.20

Code 4.20: Function for the calculation of the second moment of area provided by Professor Baumgartner

```
1  % Evaluierung von Traegern
2  % Eingaben: c ... Matrix der Knickpunkte (2 Zeilen, N
       Spalten) x-Werte in 1. Zeile
3  %           wd ... Wandstaerke
4  % Ausgabe: h ... Vektor der relativen Abweichungen der
       Ist- von den Sollwerten positiv = ok
5  %
6  %Wichtig: Alle Eingaben in cm
7  %
       ----------------------------------------------------------------------

8  function h=profeval(c,wd)
9
10
11 Iymin=30.11;      % Minimales Flaechentraegheitsmoment
       um y
12 Izmin=233;        % Minimales Flaechentraegheitsmoment
       um z
13 Itmin=0.06;       % Minimales Torsionswiderstandsmoment
14 wzmin=29.3;       % Minimales Widerstandsmoment um z
15 wymin=2;          % Minimales Widerstandsmoment um y
16 rtymin=3;         % Minimaler Traegheitsradius um y
17 rtzmin=35;        % Minimaler Traegheitsradius um z
18
19
20 U=sum(sqrt(sum((diff(c').^2)')));
21 global profile_type
22 if profile_type
23 %Fuer offene Profile
24 It= U*wd^3/3;
25 else
26 %fuer geschlossene Profile
```

Code 4.21: Function to determin a generalised logistic function *sigmoid.m*

```
function s=sigmoid(x)
 A=0;
 K=1;
 B=64.46;
 v=1/70;
 Q=1;
 M=0;
 s=A+((K-A)./(1+Q.*exp(-B.*(x-M))).^v);
```

```
Am=polyarea(c(1,:),c(2,:));
It=(2*Am)^2*wd/U;
end
s=schwerpunkt(c);
[iy,iz,iyz,wy,wz,rty,rtz]=ftm(c,schwerpunkt(c),wd);


h=[(iz-Izmin)/Izmin;(iy-Iymin)/Iymin;(It-Itmin)/Itmin
    ;(wy-wymin)/wymin;(wz-wzmin)/wzmin;(rty-rtymin)/
    rtymin;(rtz-rtzmin)/rtzmin];
```

4.6.3 The generalised logistic function *sigmoid*

Instead of a implementing a standard sigmoid function, the generalized logistic function as devised by [36] was chosen for its high degree of flexibility, allowing adaptation according to needs (code: 4.21).

A defines the lower limit of the results while K defines the upper limit. B denotes the growth-rate. v defines the maximum-growth bias. Set to 1, the growth taking place below and above $x = 0$ is identical. Lower values will lead to maximum-growth taking places for input values below 0, while higher values increasingly push the maximum-growth towards input values above 0. Q influences the result of the input 0. M is a parameter which only also affect the maximum-growth, however only if $Q = v$.

The values where chosen to approximately result in a curve fitting a set of conditions. An input of 0 should yield a result of 0.99 or above. And results of 0.01 should be attained for inputs smaller than -5. The values shown in the code 4.21 yield 0.010009 for an input of -5 and 0.99015 for 0. These result are deemed sufficient.

4.7 Conserved segments in genetic algorithm

As previously mentioned, the implementation of the genetic algorithm supports so called conserved segments. These are segments that will not be affected by mutations or crossovers. This means that arbitrary features can be defined that are needed (e.g., specific angles between two edges of the profile or straight surfaces of a certain length and inclination for mounting). Those will be considered during the optimization process. The result will therefore be a good solution with the feature included instead of merely adding the feature afterwards and hoping that the solution will still be good in spite of the feature. This is a novel feature.

4.8 Operator parameters

The crossover and the mutation operators are both part of the genetic algorithm (even though some exist that omit either one or the other). But they are not applied to each and every individual of the population. The number of operations performed relative to the total number of individuals in the population, often referred to as rates, can have a huge influence on the quality of the results achieved within a set number of generations.

To determine a good combination of crossover- and mutation-rates, a testing scenario has been established. This consists of three populations (henceforth called base-populations) created by *setupGA* with thirty individuals and a runtime of one-thousand generations. The algorithm is then started once for every combination of rates between 0.1 and 0.9 (incremented in steps of 0.1), and base-populations for a grand total of two hundred and forty-three runs. The range of rates was chosen to get a look at as broad a spectrum as possible while omitting those rates that result in operators not being used or operators being applied to the same amount of individuals as there are in the population. This cut-off value, while seemingly arbitrary, was selected based on the experience gained with various values during the development phase. The step-size for the increment was set to 0.1 to result in a agreeable number of different rates while also keeping the duration of the test to a manageable level. The three base-population were created to have some variety. By reusing those base-populations for each rates-combination, results become comparable as disparities in the test conditions due to different initial conditions related to the fitness of the population being evolved can be precluded.

To gauge the quality of the results, the best fitness value, as well as the mean of all fitness values at the end of the algorithm are compared.

Graph 4.6 depicts the average fitness value of each population after 1000 generation as they evolved for each base-population. Graph 4.7 show the best fitness value of the 1000th generation as result of the evolution of each

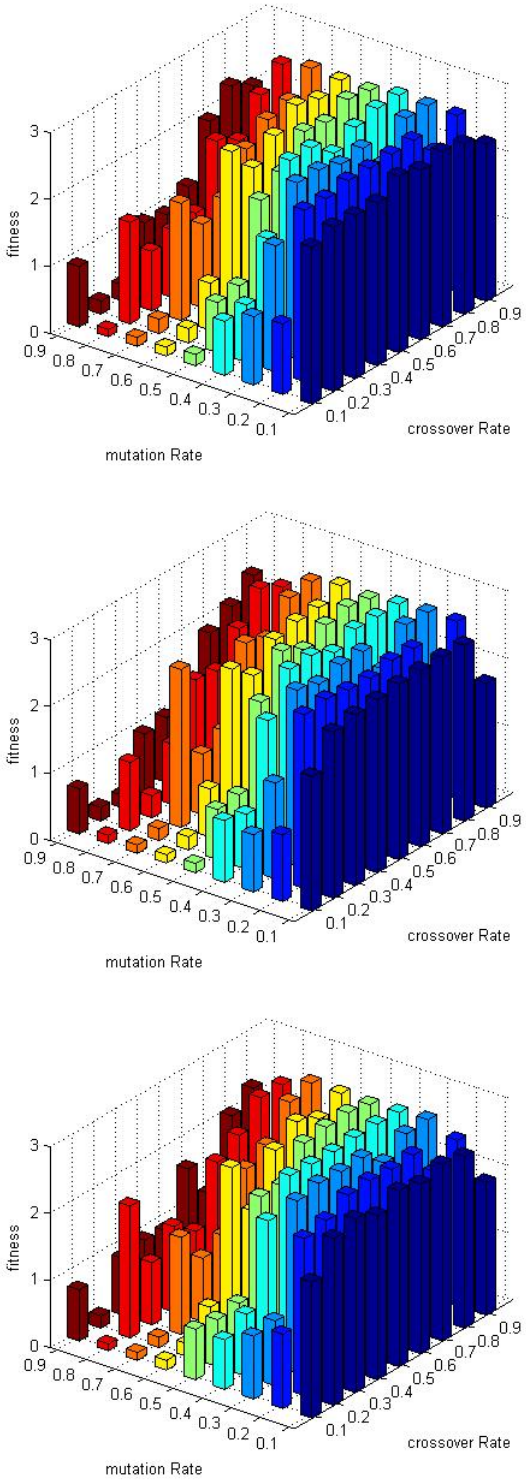

Figure 4.6: Average fitness values of the population after 1000 generations for different crossover- and mutation-rates. Top: Values evolved from the first base-population. Middle: Values evolved from the second base-population. Bottom: Values evolved from the third base-population.

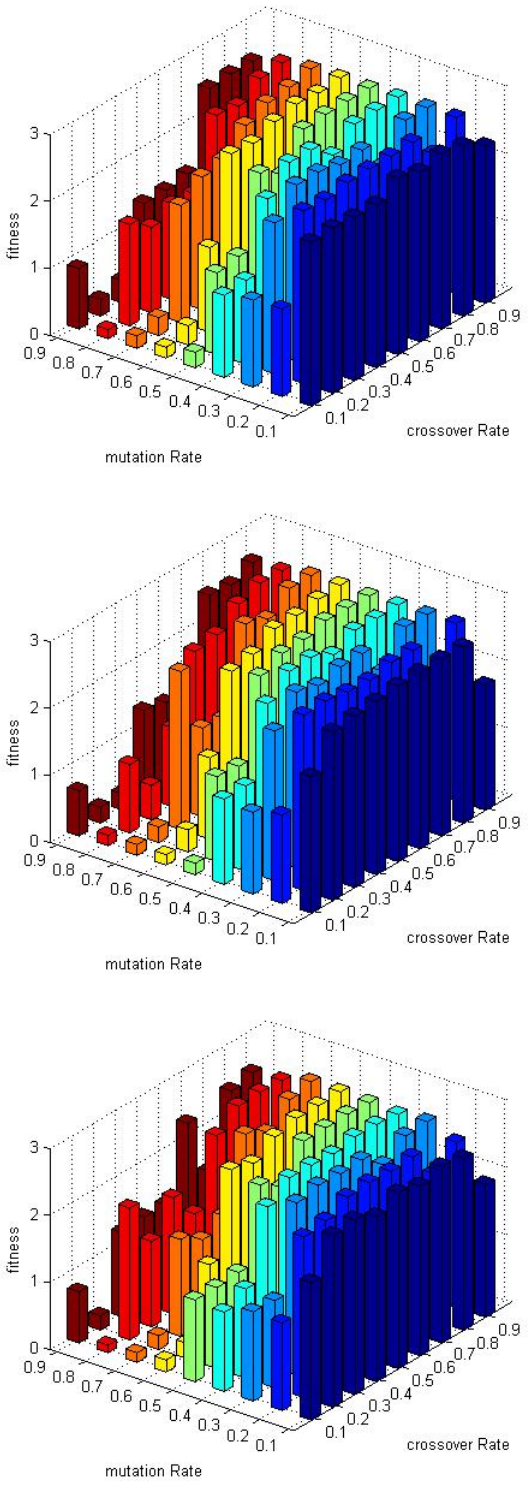

Figure 4.7: Best fitness value of the population after 1000 generations for different crossover- and mutation-rates. Top: Values evolved from the first base-population. Middle: Values evolved from the second base-population. Bottom: Values evolved from the third base-population.

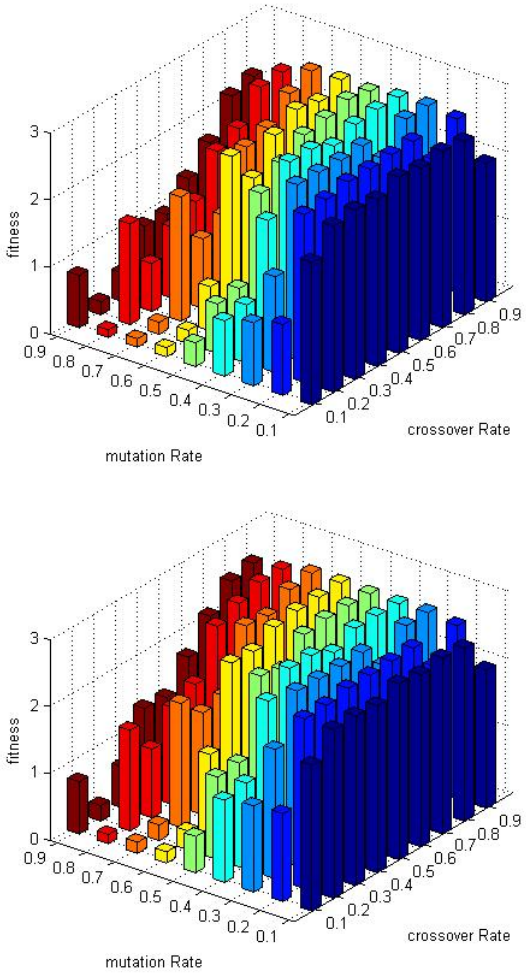

Figure 4.8: Averaged best fitness and average fitness values of the populations after 1000 generations, from the results evolved from the three base-populations. Top: Mean of the average fitness values. Bottom: Mean of the best fitness value.

base-population. The results of those two graphs are averaged in the graph 4.8. As can be seen over all these figures, except for a a few exceptions, there seems to be a pattern dividing the graphs into two distinct areas along an imaginary diagonal approximately running from the mutation-rate 0.1, crossover-rate 0.1 to the respective rates of 0.9 and 0.7. The rate-combinations to the left just reaching above fitness-values of 1, while those on the right exceed fitness-values of 2. The values in the high-area are mostly close and without an obvious, graphically outstanding, outlier. As a matter of fact, the values are too close to claim a noticeable improvement of the fitness values, either the populations average or best, by using a different combination of rates from the set defining the high area. While this could be caused by a proximity to a peak in the fitness-landscape, limiting the further growths of the fitness-value as it is already close to a potential maximum. Whether this is the case or not, as increased rates also require more computing time, the choice of rate for both operations can be based on the lowest resource cost (the prioritized resource being computing time).

To this end the average runtime per generation of each rate combination is determined over 100 generations. The rest of the test is identical to the previous one.

The graphs showing the runtime per generation, both individually for the evolution from each base-population (4.9) and the averaged times (4.10), all show a similar pattern of steadily rising runtime as the mutation-rate increases. The crossover-rate does not seem to have any overly significant influence on the runtime (at least not relatively to the profuse impact of the mutation-rate).

With the data on the average and maximum fitness-values as well as the average runtime, the combination of rates for the two operators yielding the best results while requiring the least runtime can be determined and set as default values for the core algorithm. To do this, the shortest runtime for those rate combinations that yielded the best or maximum average result after 1000 generations, or a result within 1% of it. The rates providing the best results for the least time resource invested are 0.5 for the crossover and 0.1 for the mutation with a fitness value of 2.6258 and an average runtime per generation of 0.2842 seconds. The other combinations yielding results within 1% of the best fitness value **AND** the maximum average result respectively are ([crossover-rate;mutation-rate]): $[0.7; 0.1]$, $[0.6; 0.2]$, $[0.7; 0.2]$, $[0.8; 0.3]$, $[0.9; 0.3]$, $[0.4; 0.4]$, $[0.8; 0.4]$, $[0.9; 0.4]$, and $[0.8; 0.5]$. These combinations can therefore also be considered excellent candidates for default values, for example, if more mutations are desired in an attempt to increase the diversity of the population.

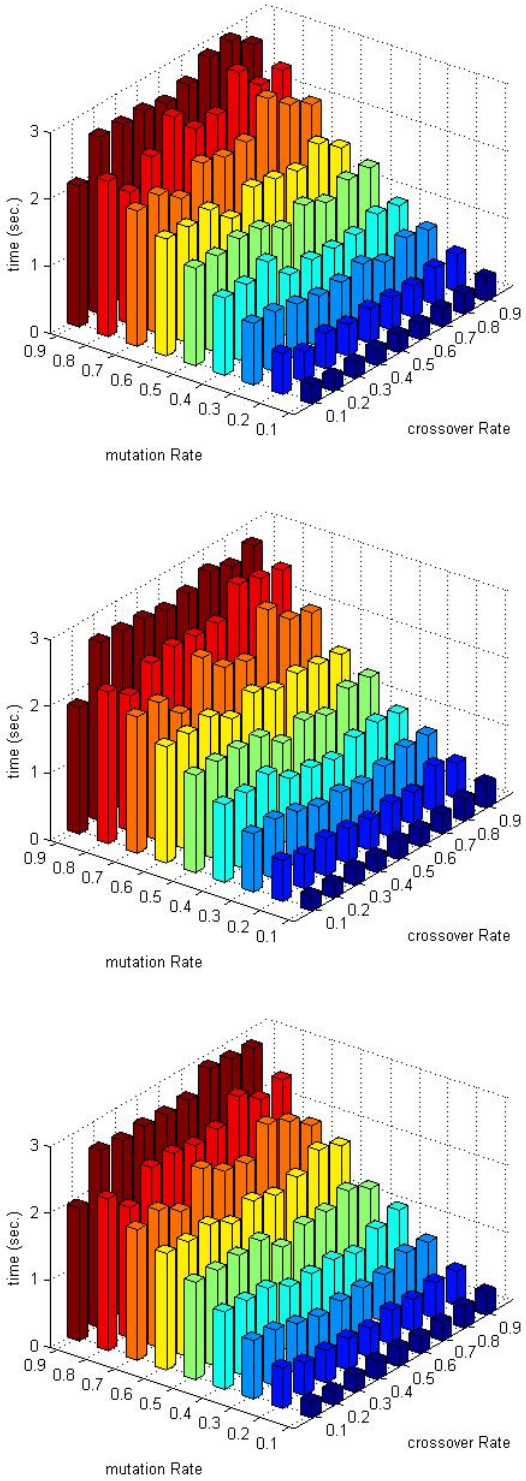

Figure 4.9: Average runtime (in seconds) of the genetic algorithm per generation for different combinations of crossover- and mutation-rates as determined over 100 generations. Top: runtime of evolutions based on the first base-population. Middle: runtime of evolutions based on the second base-population. Bottom: runtime of evolutions based on the third base-population.

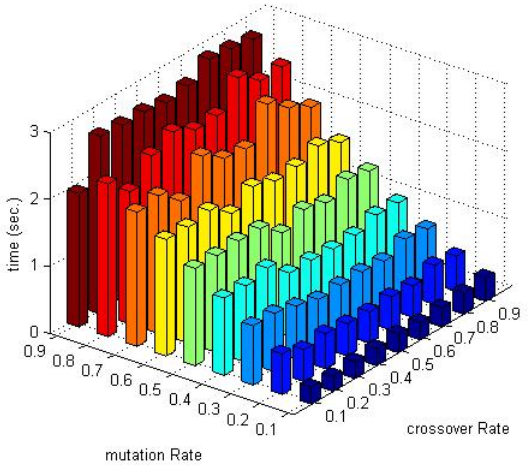

Figure 4.10: Averaged mean runtime (in seconds) of the genetic algorithm per generation for different combinations of crossover- and mutation-rates as determined over 100 generations, from the results evolved from the three base-populations.

Chapter 5

Results

Two versions of a genetic algorithms were presented in detail over the last chapters. They are both characterised by a high degree of flexibility by the modular and general implementation of many functions. They also boast a novel feature in the form of *regionLock*s, permitting a degree of control over features of the optimization problem, thereby avoiding the issue of potential solutions that are optimal but unusable under real-life conditions. This approach is advantageous compared to simply omitting parts during the optimization and adding them afterwards. By directly incorporating such limitations, the solution is optimized for those limitations instead of despite it.

5.1 Results with the base algorithm

As previously mentioned there are multiple configuration of the algorithm itself. It can either be run to optimize open profiles or closed profiles, It can be run with or without conserved regions, and it can determine the second area of moment (SAM) using the originally developed function or one courteously provided by Prof. Baumgartner (see code listing 4.20). To account for all possible combinations of configurations, a large setup was run via a script, starting runs with 100 individuals over 10000 generations. For each combination of profile type, region lock and second area of moment function, three initial populations were created and run thrice each for a total of nine runs per configuration. For the evaluation in this section, only respectively the worst, the best, and the value closest to the calculated mean fitness of the generation shown are pictured and have their values presented.

The region locks used are the values [20,15,10;20,20,20] at position 9 for closed profiles, and the values [0,5,10;20,20,20] at position 12 for open profiles.

For all runs presented in this chapter, a slightly modified fitness function was used. Instead of $\frac{l}{1}$ a linear progression was used: $100 - l$ (in both cases

Generation	Length			Gain		
	worst	mean	best	worst	mean	best
1	292.7284	241.8009	277.9824	-265.9105%	-202.2512%	-247.4780%
10	281.9171	249.1189	266.9024	-252.3963%	-211.3987%	-233.6280%
100	279.9219	241.7506	266.2085	-249.9023%	-202.1883%	-232.7606%
1000	290.7749	241.9822	247.7461	-263.4687%	-168.7278%	-209.6826%
10000	214.1499	264.2628	223.5295	-167.6874%	-230.3284%	-179.4118%

Table 5.1: Length in units and improvement gain in percent for the worst, average and best profile after 10, 100, 1000, 10000 generations for closed profiles, with no conserved regions, using the original SAM function.

l is the length/circumference of the profile).

5.1.1 Case 1: closed profiles, no conserved regions, original SAM function

The figure 5.1 shows respectively snapshots of the evolution of closed profiles without conserved regions using the original SAM functions. Those snapshots were taken after 10, 100, 1000, and 10000 generations. For each, the best profile (left column), the worst profile (right column), and a profile close with a fitness value close to the mean fitness of the population (middle column) are shown. All profiles share a resemblance, possessing a wider upper half that narrows down until the sides intersect to form a narrow base sticking out of the figure to the right.
The values associated to each figure, as presented in the table 5.1, show that not only the form form of the individual figures resemble, but that they subsequently possess similar circumferences. The values however also prove that there actually is a clear progress. The length, meaning the circumference of the profile, steadily decreases for all profiles, worst, mean, and best. This, while being equally far from a usable solution, shows clear improvements for the population as a whole.

Figure 5.2, which shows the best fitness value (in blue) and the calculated mean fitness value of the population (orange) for each generation, additionally confirms that the population constantly improves. Both, the best and the mean fitness, increase slowly but steadily for the first about 4000 generations before reaching a plateau. From then on changes, when they do occur are smaller compared to the improvements of the earlier phases. Also note the spikes in best fitness that occur occasionally for a small amount of generations.

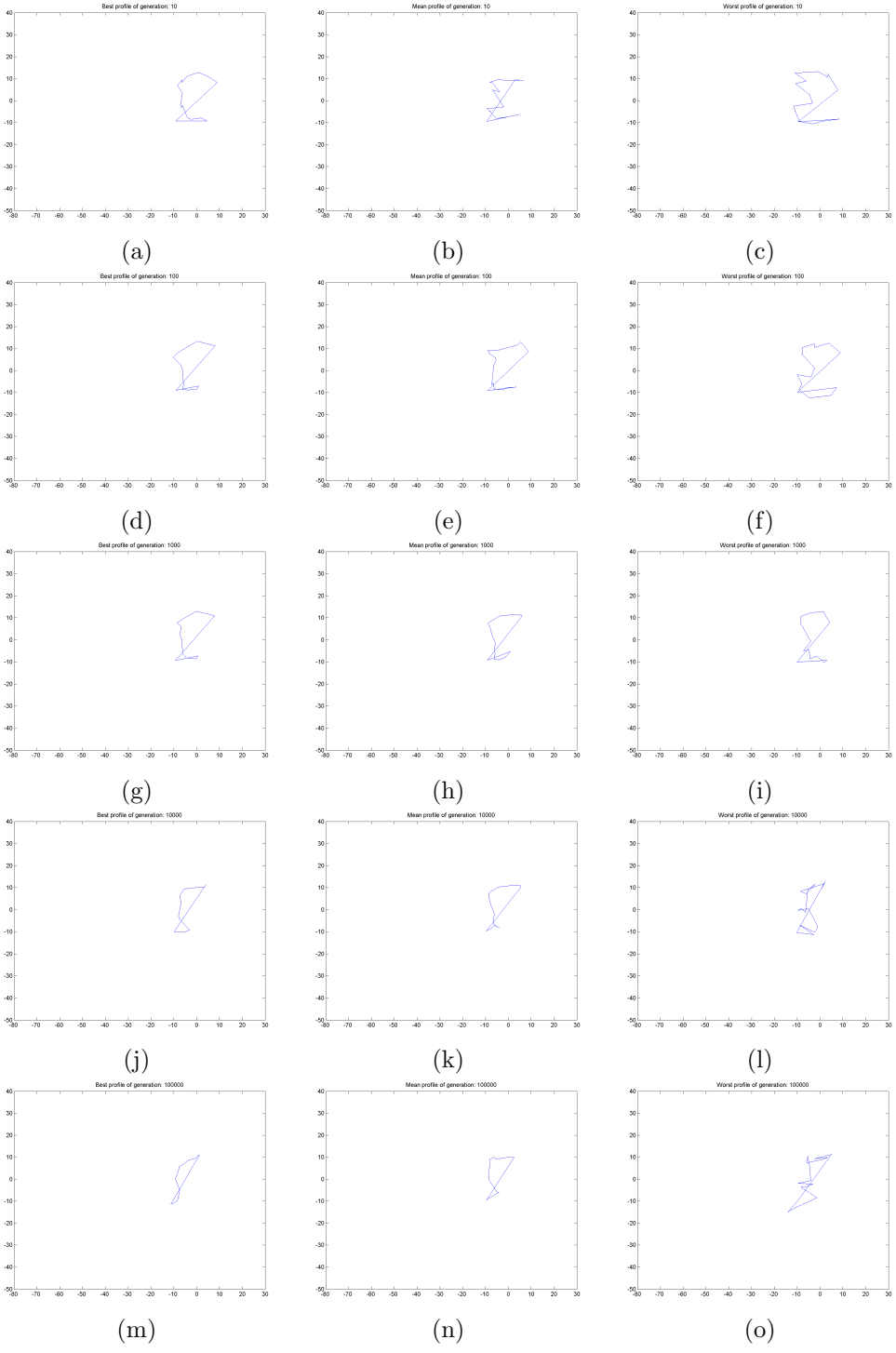

Figure 5.1: Worst, average and best profile after 10, 100, 1000 and 10000 generations for closed profiles, with no conserved regions, using the original SAM function.

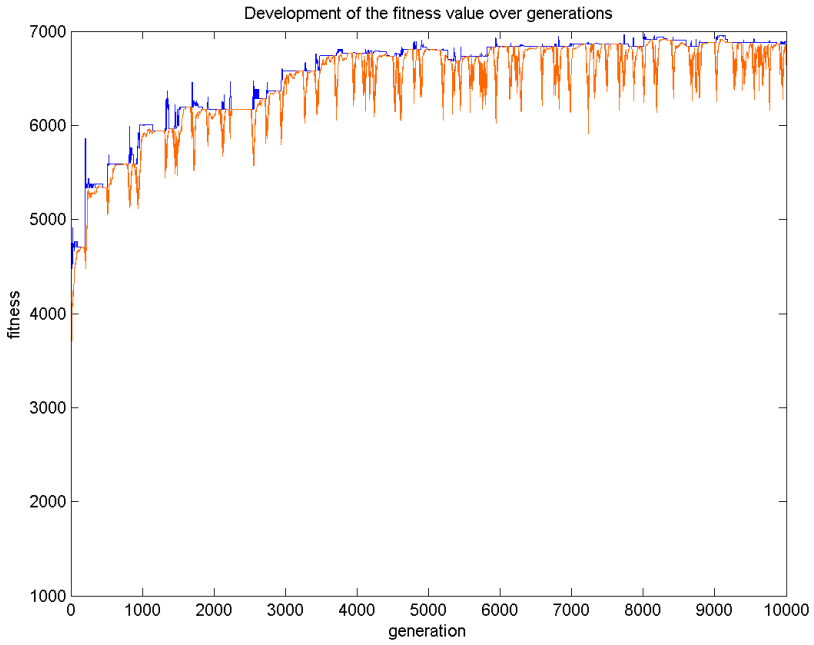

Figure 5.2: Fitness over 10000 generations for closed profiles, with no conserved regions, using the original SAM function. The blue line shows the best fitness value of each generation. The orange line shows the mean fitness value of the population for each generation.

Generation	Length			Gain		
	worst	mean	best	worst	mean	best
1	524.0549	283.3296	273.7854	-0.5551e03%	-254.1620%	-242.2317%
10	604.6041	284.2004	252.2732	-0.6558e03%	-255.2505%	-215.3416%
100	945.2969	276.1702	278.7286	-1.0816e03%	-245.2127%	-248.4107%
1000	850.9408	240.3225	240.3225	-0.9637e03%	-200.4031%	-200.4031%
10000	528.1956	225.8715	189.4198	-0.5602e03%	-182.3393%	-136.7747%

Table 5.2: Length in units and improvement gain in percent for the worst, average and best profile after 10, 100, 1000, 10000 generations for closed profiles, with no conserved regions, using the provided SAM function.

5.1.2 Case 2: closed profiles, no conserved regions, provided SAM function

The figure 5.3 shows respectively snapshots of the evolution of closed profiles without conserved regions using the provided SAM functions. Those snapshots were taken after 10, 100, 1000, and 10000 generations. For each, the best profile (left column), the worst profile (right column), and a profile close with a fitness value close to the mean fitness of the population (middle column) are shown. The best and mean profiles are very similar to the profiles described in the first case. The profiles for the worst profiles however can not be described by using known patterns as they seem to be rather randomized polygons which have nothing but their randomness in common. The values associated to each figure, as presented in the table 5.2. The worst profiles seem to be getting worth until the hundredth generation before improving again to almost reach the starting value by the ten-thousandth generation. The mean values on the other hand are slowly but steadily improving. The best values are improving as well, but are set back in the hundredth generation.

Due to a technical issue a graph for the mean and best fitness values over time is not available for this case.

5.1.3 Case 3: open profiles, no conserved regions, original SAM function

The figure 5.4 shows respectively snapshots of the evolution of open profiles without conserved regions using the original SAM functions. Those snapshots were taken after 10, 100, 1000, and 10000 generations. For each, the best profile (left column), the worst profile (right column), and a profile close with a fitness value close to the mean fitness of the population (middle column) are shown. Up to the thousandth generation, the basic form of all profiles, regardless of whether it is considered worst, mean, or best, look

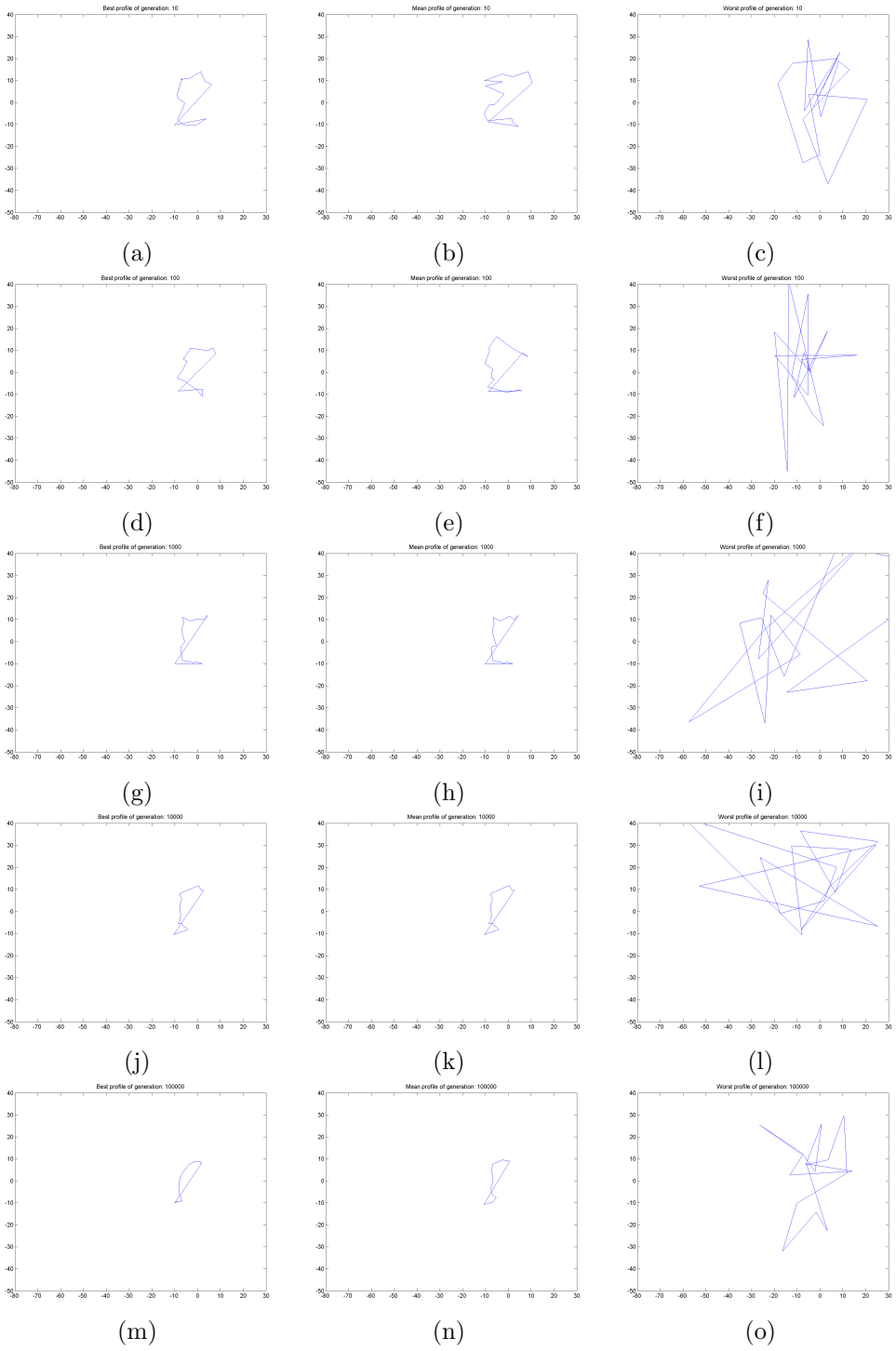

Figure 5.3: Worst, average and best profile after 10, 100, 1000 and 10000 generations for closed profiles, with no conserved regions, using the provided SAM function.

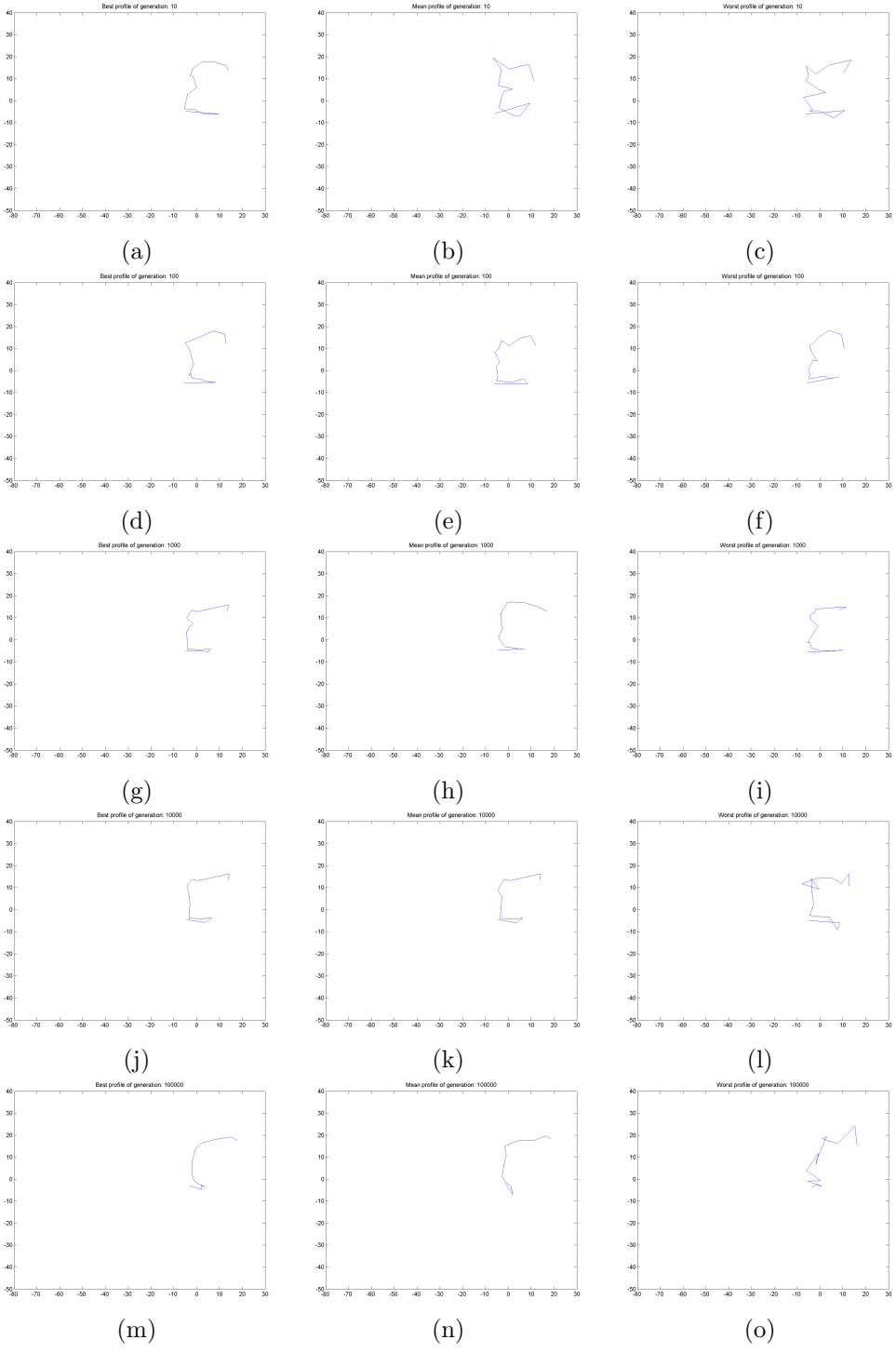

Figure 5.4: Worst, average and best profile after 10, 100, 1000 and 10000 generations for open profiles, with no conserved regions, using the original SAM function.

Generation	Length			Gain		
	worst	mean	best	worst	mean	best
1	83.4930	84.9520	80.0461	-36.9989%	-39.3928%	-31.4331%
10	76.9719	89.5269	74.1130	-26.2988%	-46.8996%	-21.6078%
100	79.3794	87.0840	80.9678	-30.2490%	-42.8911%	-32.8390%
1000	105.9567	75.8422	76.1615	-73.8583%	-24.4452%	-24.9690%
10000	57.1760	78.4444	63.4798	-6.1832%	-28.7149%	-4.1604%

Table 5.3: Length in units and improvement gain in percent for the worst, average and best profile after 10, 100, 1000, 10000 generations for open profiles, with no conserved regions, using the original SAM function.

similar to each other as well as to the basic form. The central indentation of the basic sigma-form gets less and less pronounced over the generation. A notable difference is the vertical line at the bottom which, unlike the base form, does not end in a short upturned segment but in a u-turn that appends a segment to create what looks like a double line at the bottom. For the best and mean profiles in the ten-thousandth generation, both the bottom vertical lines (both of them) are mostly retracted. The worst profile also seems to have a more or less retracted lower vertical line but also seems more intersected along the top half of the profile. Also, unlike the smoother curvature and line course for best and mean profiles, the worst profiles appears to be much more jagged.
The values associated to each figure, as presented in the table 5.3. For the first thousand generation, no clear trend can be discerned. But by the ten-thousandth generation, the best and worst profiles respectively make huge improvements and are within 10% of the reference values.

Figure 5.5, which shows the best fitness value (in blue) and the calculated mean fitness value of the population (orange) for each generation, additionally confirms that the population constantly improves. Both, the best and the mean fitness, increase slowly but steadily for the first about 4000 generations before reaching a plateau. From then on changes, when they do occur are smaller compared to the improvements of the earlier phases. Also note the spikes in best fitness that occur occasionally for a small amount of generations.

5.1.4 Case 4: open profiles, no conserved regions, provided SAM function

The figure 5.6 shows respectively snapshots of the evolution of open profiles without conserved regions using the provided SAM functions. Those snapshots were taken after 10, 100, 1000, and 10000 generations. For each, the worst profile (right column), the best profile (left column), and a profile

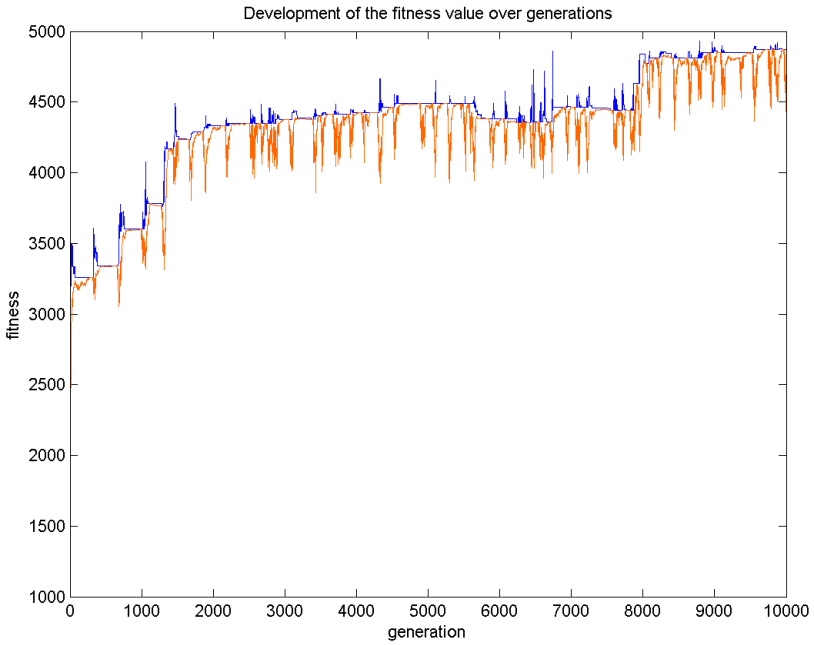

Figure 5.5: Fitness over 10000 generations for open profiles, with no conserved regions, using the original SAM function. The blue line shows the best fitness value of each generation. The orange line shows the mean fitness value of the population for each generation.

Generation	Length			Gain		
	worst	mean	best	worst	mean	best
1	204.7520	103.1464	76.3433	-235.9658%	-69.2470%	-25.2673%
10	366.9619	94.5197	76.1833	-502.1268%	-55.0919%	-25.0048%
100	382.6081	79.4665	80.9812	-527.7996%	-30.3920%	-32.8773%
1000	174.8734	62.4893	62.4893	-186.9397%	-2.5351%	-2.5351%
10000	276.5471	35.3246	32.1769	-353.7702%	-42.0378%	47.2028%

Table 5.4: Length in units and improvement gain in percent for the worst, average and best profile after 10, 100, 1000, 10000 generations for open profiles, with no conserved regions, using the provided SAM function.

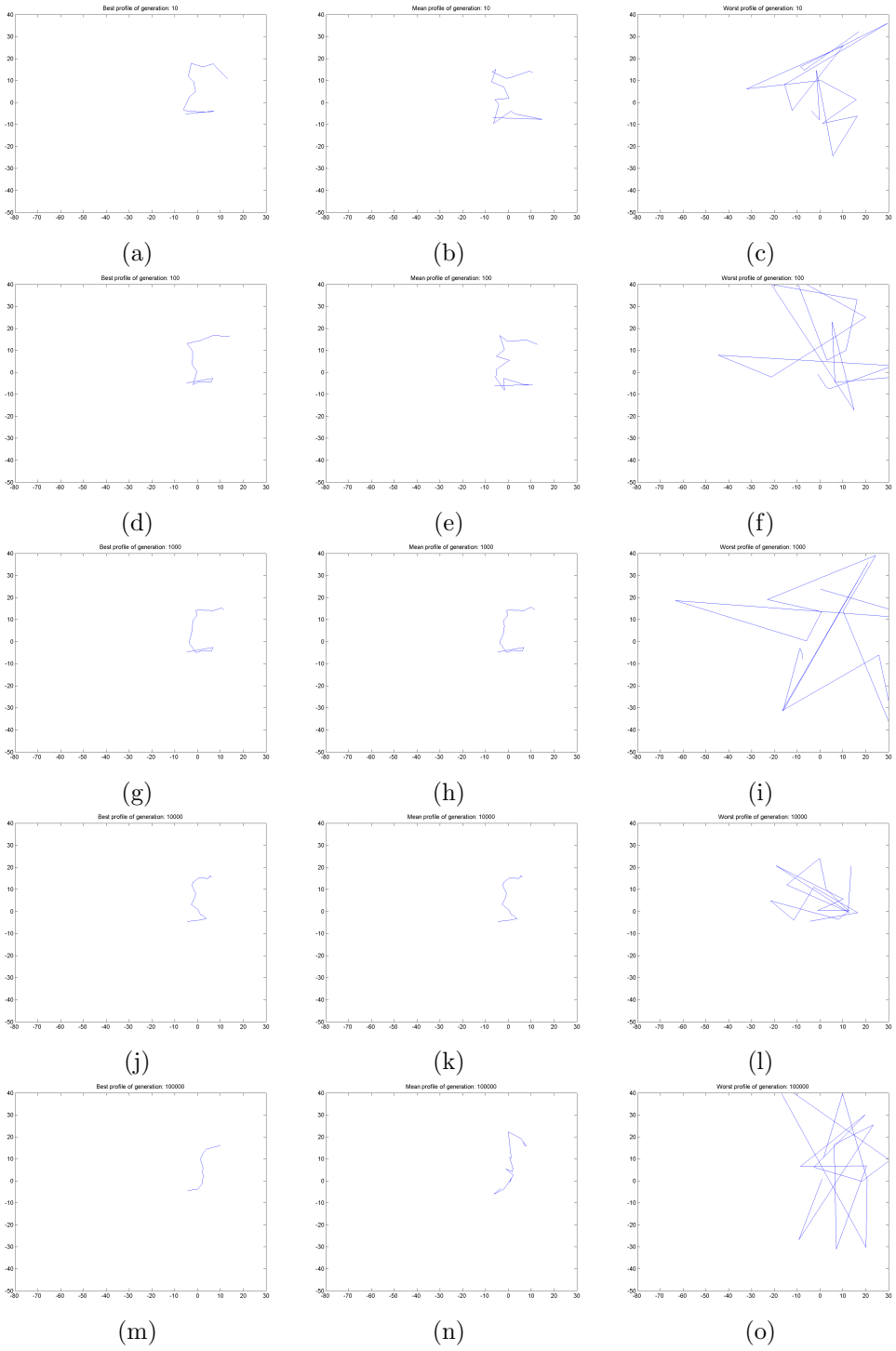

Figure 5.6: Worst, average and best profile after 10, 100, 1000 and 10000 generations for open profiles, with no conserved regions, using the provided SAM function.

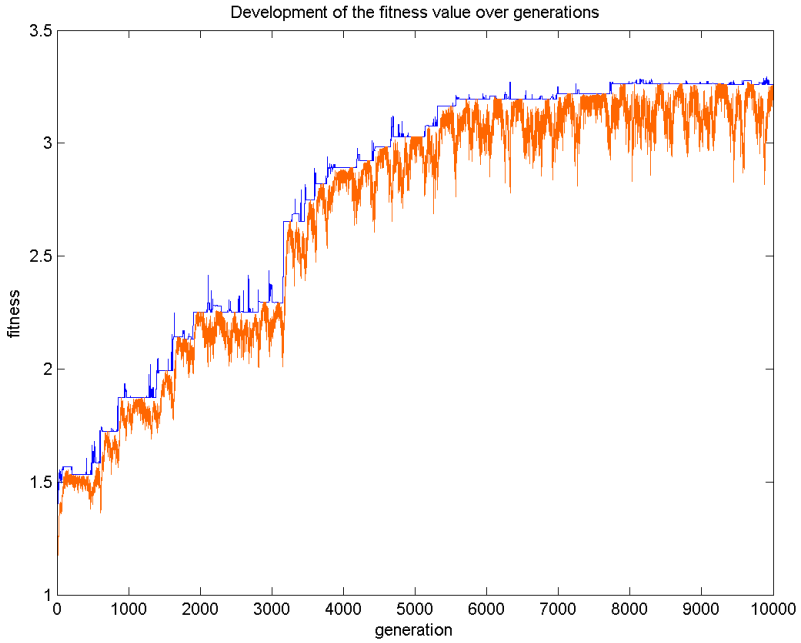

Figure 5.7: Fitness over 10000 generations for open profiles, with no conserved regions, using the provided SAM function. The blue line shows the best fitness value of each generation. The orange line shows the mean fitness value of the population for each generation.

close with a fitness value close to the mean fitness of the population (middle column) are shown.The best and mean profiles at first look similar to the shapes obtained in case 3. By the thousandth generation however they start to push back the double vertical line on the bottom of the profile and develop into a S like shape. The worst profiles on the other hand are, like in case 2 indescribable and merely appear entirely random.
The values associated to each figure, as presented in the table 5.4. There doesn't seem to be a obvious trend as gain sometimes improves and sometimes decreases between generations. The worst profiles indeed become worse even when comparing the values of the first and the ten-thousandth generation.

Figure 5.7, which shows the best fitness value (in blue) and the calculated mean fitness value of the population (orange) for each generation, additionally confirms that the population improves. The fitness improves in jumps every few hundred generations until the six-thousandth generation. After that, the fitness reaches a plateau and only increases slightly if at all.

Generation	Length			Gain		
	worst	mean	best	worst	mean	best
1	229.6495	227.6000	223.8556	-187.0618%	-184.5000%	-179.8195%
10	252.4190	224.0755	212.8259	-215.5238%	-180.0944%	-166.0324%
100	237.8768	229.4928	231.2398	-197.3460%	-186.8661%	-189.0498%
1000	202.1560	168.7871	168.7871	-152.6949%	-110.9839%	-110.9839%
10000	215.5015	253.0904	176.0282	-169.3768%	-216.3631%	-120.0352%

Table 5.5: Length in units and improvement gain in percent for the worst, average and best profile after 10, 100, 1000, 10000 generations for closed profiles, with conserved regions, using the original SAM function.

5.1.5 Case 5: closed profiles, conserved regions, original SAM function

The figure 5.8 shows respectively snapshots of the evolution of closed profiles with conserved regions using the original SAM functions. Those snapshots were taken after 10, 100, 1000, and 10000 generations. For each, the best profile (left column), the worst profile (right column), and a profile close with a fitness value close to the mean fitness of the population (middle column) are shown. The conserved region can be seen in each image as a straight line at the top of the profile. The worst, mean, and best profiles all start out with a similar shape resembling a Z. They also all contain numerous intersections. The mean and best profiles gradually untangle those intersections and lose the horizontal section at their bottom. By the ten-thousandth generation the best profile as achieved an almost triangular shape. The mean profiles, not narrowing quite as much at the bottom are more comparable to an oblique rectangle. The worst profiles also retract the horizontal section at their bottom. But instead of completely loosing them, the section merely becomes scrambled.
The values associated to each figure, as presented in the table 5.5. Even though between the first and last generation an improvement is undeniable, the intermediate values shown (at least for the mean and best profiles) alternate between improvement and worsening.

Figure 5.9, which shows the best fitness value (in blue) and the calculated mean fitness value of the population (orange) for each generation. During the first few hundred generations, for both best and mean values of the population, the fitness values have their strongest increase. After the two-thousandth generation a plateau is reached and the best values only display minor improvements while mean fitness is still oscillating.

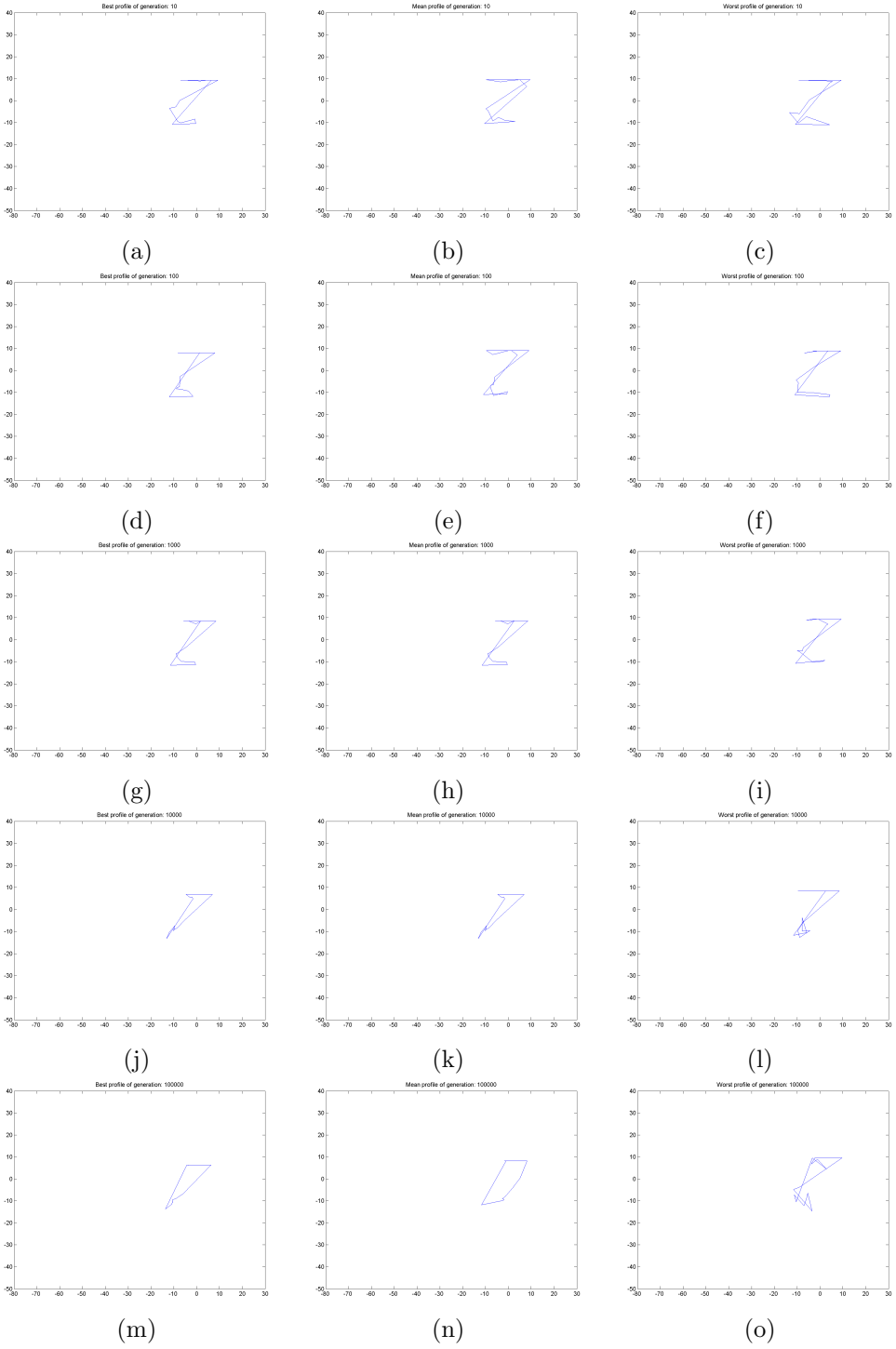

Figure 5.8: Worst, average and best profile after 10, 100, 1000 and 10000 generations for closed profiles, with a conserved region, using the original SAM function.

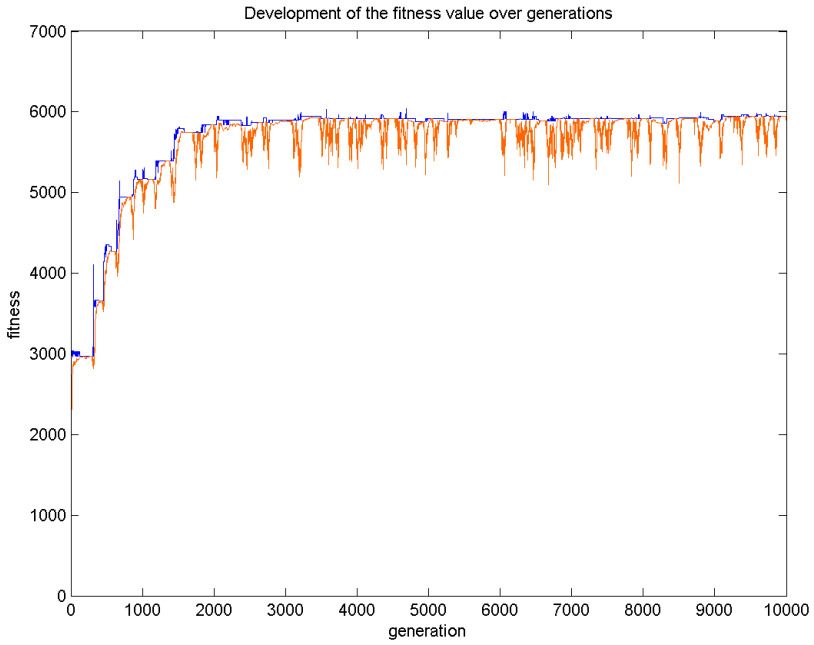

Figure 5.9: Fitness over 10000 generations for closed profiles, with conserved regions, using the original SAM function. The blue line shows the best fitness value of each generation. The orange line shows the mean fitness value of the population for each generation.

Generation	Length			Gain		
	worst	mean	best	worst	mean	best
1	498.4319	253.9887	216.4326	-523.0398%	-217.4859%	-170.5408%
10	458.9111	239.1866	235.0687	-473.6389%	-198.9833%	-193.8359%
100	711.7434	225.2667	214.2673	-789.6792%	-181.5833%	-167.8341%
1000	430.4415	165.4704	165.4704	-438.0518%	-106.8380%	-106.8380%
10000	487.9386	192.0710	179.4990	-509.9233%	-140.0887%	-124.3738%

Table 5.6: Length in units and improvement gain in percent for the worst, average and best profile after 10, 100, 1000, 10000 generations for closed profiles, with conserved regions, using the provided SAM function.

5.1.6 Case 6: closed profiles, conserved regions, provided SAM function

The figure 5.10 shows respectively snapshots of the evolution of closed profiles with conserved regions using the provided SAM functions. Those snapshots were taken after 10, 100, 1000, and 10000 generations. For each, the best profile (left column), the worst profile (right column), and a profile close with a fitness value close to the mean fitness of the population (middle column) are shown. The conserved region can be seen in each image. The conserved region can be seen in each image as a straight line at the top of the profile. The profiles and their evolution in this case like highly similar to those depicted in case 5. The best and mean profiles again start out with a Z-like shape. The best profile evolves into an almost triangular form, while the mean profiles take on the shape of an oblique rectangle. The worst profiles are once again random shapes.
The values associated to each figure, as presented in the table 5.6. While some improvements can be seen between the first and last generation for all profile categories, the intermediate values do not necessarily follow a clear trend. The best values for example worsen during the first ten generations, then improve until the thousandth generation, but end up slightly worsening again by the ten-thousandth generation. The intermediate steps of the mean and worst profiles do not have the exact same trend pattern as the best profiles but interestingly also worsen between the thousandth and ten-thousandth generation.

Due to a technical issue a graph for the mean and best fitness values over time is not available for this case.

5.1.7 Case 7: open profiles, conserved regions, original SAM

The figure 5.11 shows respectively snapshots of the evolution of open profiles with conserved regions using the original SAM functions. Those snapshots

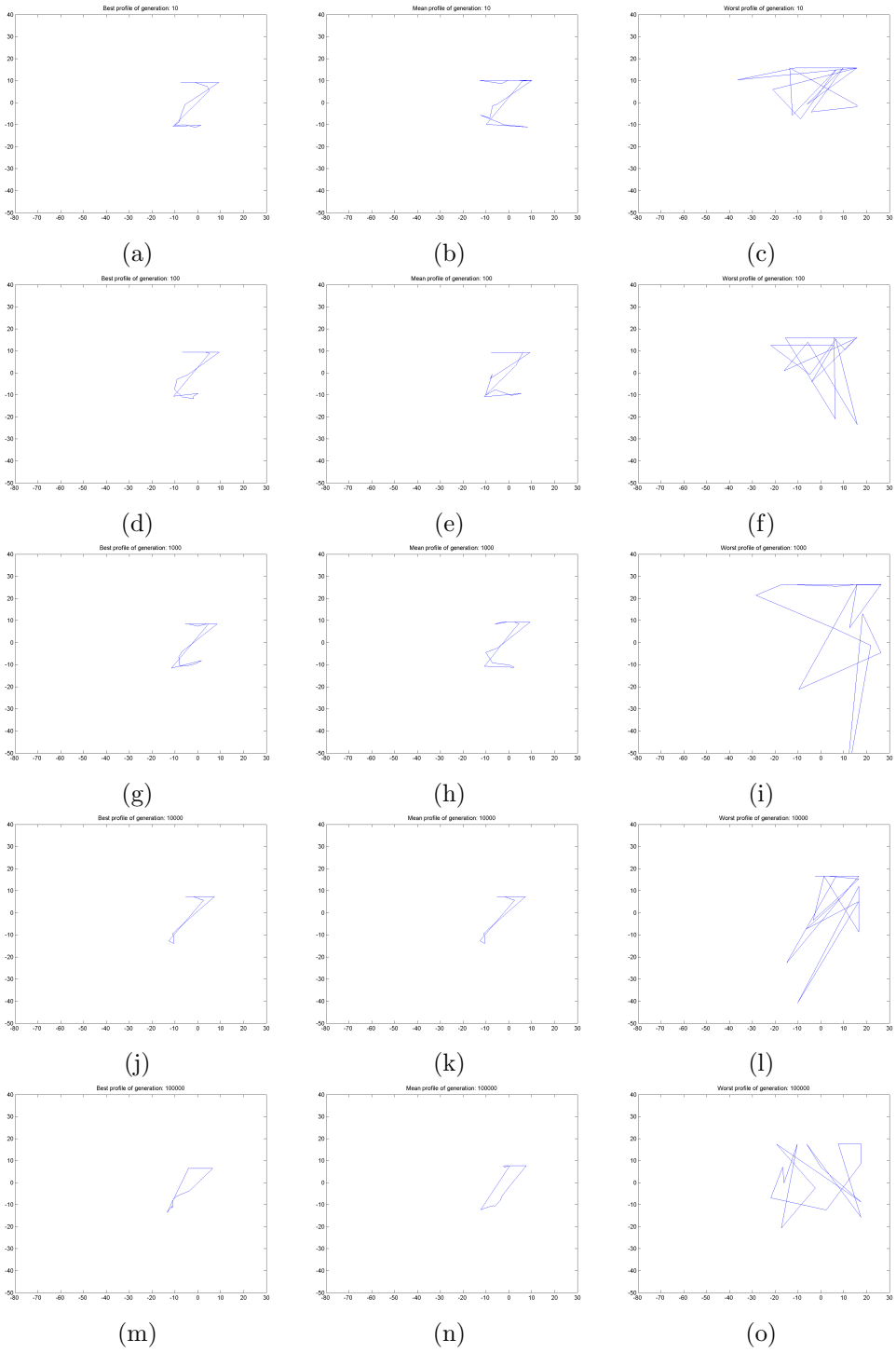

Figure 5.10: Worst, average and best profile after 10, 100, 1000 and 10000 generations for closed profiles, with a conserved region, using the provided SAM function.

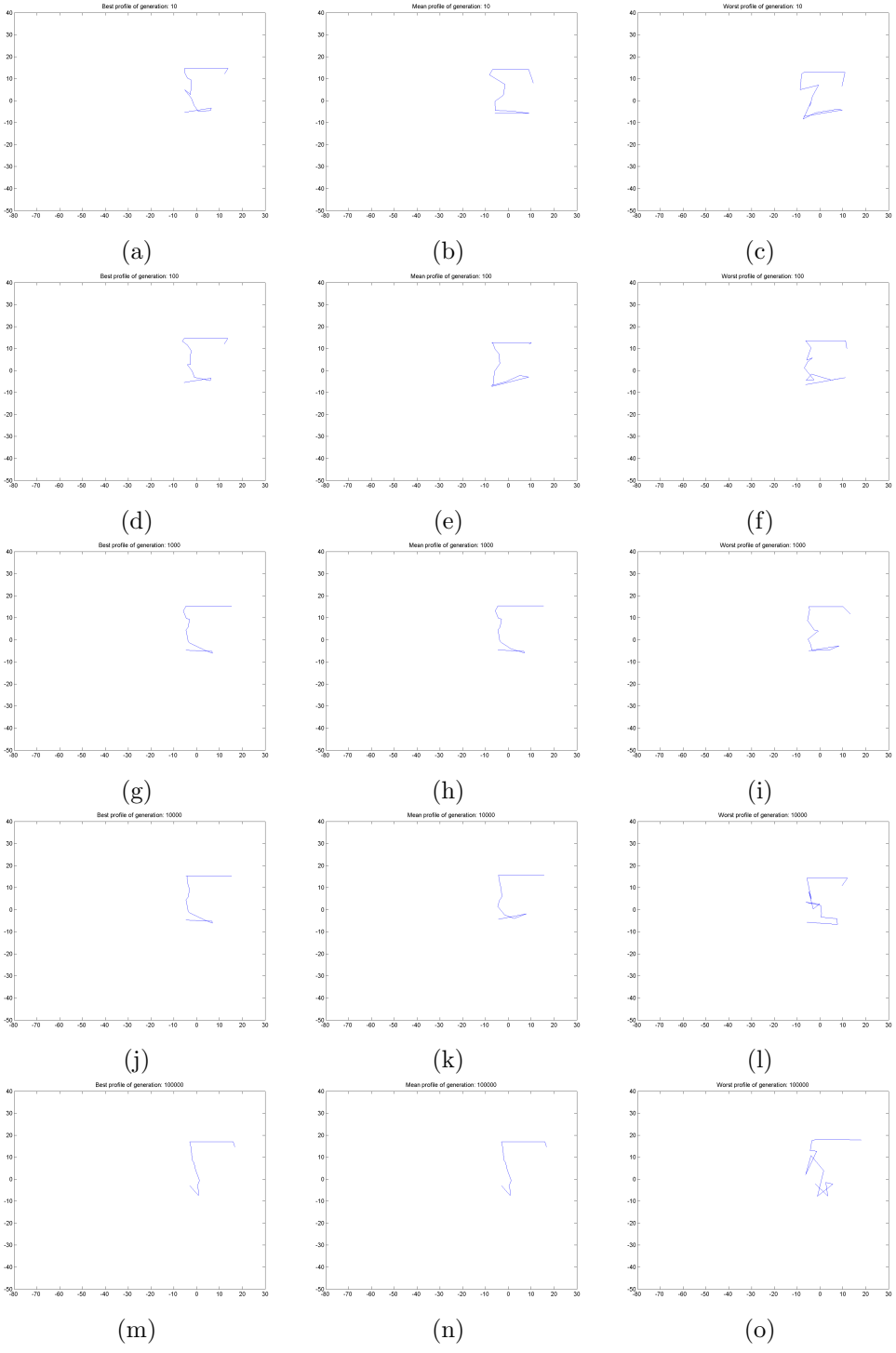

Figure 5.11: Worst, average and best profile after 10, 100, 1000 and 10000 generations for open profiles, with a conserved region, using the original SAM function.

Generation	Length			Gain		
	worst	mean	best	worst	mean	best
1	81.3663	92.1065	90.3539	-33.5093%	-51.1323%	-48.2566%
10	87.8325	68.3545	73.5250	-44.1193%	-12.1590%	-20.6430%
100	71.8022	99.9033	107.3187	-17.8161%	-63.9255%	-76.0932%
1000	95.4295	84.0080	107.3187	-56.5848%	-37.8440%	-76.0932%
10000	112.4869	76.4831	76.4831	-84.5733%	-25.4967%	-25.4967%

Table 5.7: Length in units and improvement gain in percent for the worst, average and best profile after 10, 100, 1000, 10000 generations for open profiles, with conserved regions, using the original SAM function.

were taken after 10, 100, 1000, and 10000 generations. For each, the best profile (left column), the worst profile (right column), and a profile close with a fitness value close to the mean fitness of the population (middle column) are shown. The conserved region is visible as straight line topping each profile in the figure. The best and mean profiles lose the indentation in the middle of the profile that is so characteristic for the sigma base profile early on. As with the previous cases with open profiles, from the very beginning they have a double horizontal line at the bottom. This retracts over time. By the ten-thousandth generation only a short hook to the left remains. The worst profiles retain a more obvious indentation in their middle (again from the sigma base profile) much longer. As the evolution progresses, the non-conserved part of the profile becomes more erratic and less identifiable as a known shape.
The values associated to each figure, as presented in the table 5.7. As in some other cases the improvement is clear when only looking at the first and the last values of the mean and best profiles. The intermediate values do not follow a constant improvement trend. The values of the worst profiles do not follow a constant trend either. However the worst profiles, when only considering the first and last value, worsen over time.

Figure 5.12, which shows the best fitness value (in blue) and the calculated mean fitness value of the population (orange) for each generation. The fitness starts out stagnant until approximately the generation 1800, when they slowly improve. Around the generation 2300 the fitness suddenly improves greatly only to return to a plateau. From about the generation 2700 the improvments are negligible.

5.1.8 Case 8: open profiles, conserved regions, provided SAM

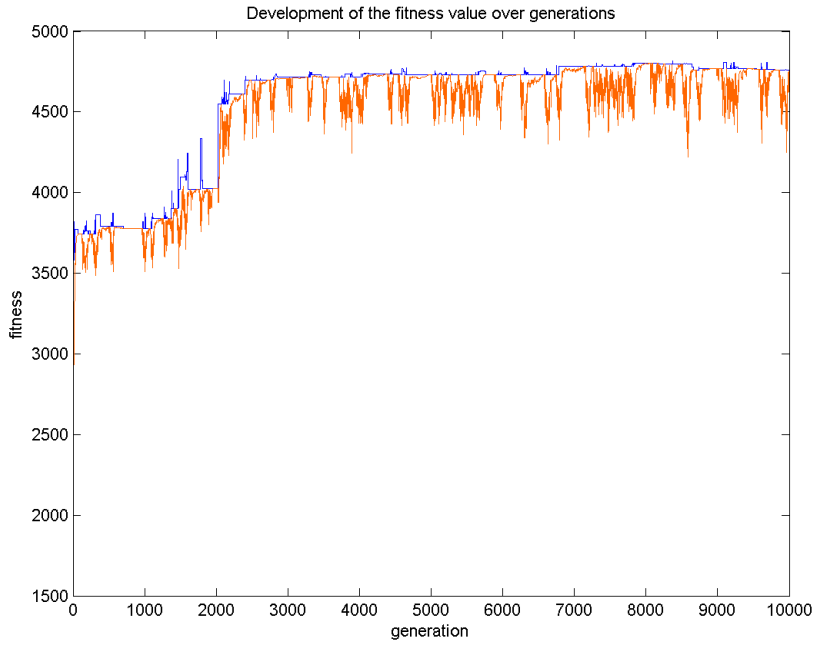

Figure 5.12: Fitness over 10000 generations for open profiles, with conserved regions, using the original SAM function. The blue line shows the best fitness value of each generation. The orange line shows the mean fitness value of the population for each generation.

Generation	Length			Gain		
	worst	mean	best	worst	mean	best
1	178.6423	82.7638	74.4861	-193.1239%	-35.8023%	-22.2199%
10	275.5778	73.8210	75.8943	-352.1798%	-21.1286%	-24.5306%
100	299.5880	55.5134	69.1518	-391.5768%	8.9113%	-13.4673%
1000	329.0414	24.9639	25.4603	-439.9052%	59.0382%	58.2237%
10000	235.8740	81.9966	31.6130	-287.0321%	-34.5435%	48.1281%

Table 5.8: Length in units and improvement gain in percent for the worst, average and best profile after 10, 100, 1000, 10000 generations for open profiles, with conserved regions, using the provided SAM function.

The figure 5.13 shows respectively snapshots of the evolution of open profiles with conserved regions using the original SAM functions. Those snapshots were taken after 10, 100, 1000, and 10000 generations. For each, the best profile (left column), the worst profile (right column), and a profile close with a fitness value close to the mean fitness of the population (middle column) are shown. The conserved region is visible as straight line topping each profile in the figure. The results in this case look very similar to the results from case 7. The best and mean profiles start out looking similar to the base profile with an added double horizontal line at the bottom. Over the course of several thousand generations, this double line retracts to leave only a slight hook to the left. The mean profile from the ten-thousandth generation has a random looking cluster of lines. This can be assumed to be an artefact which happened to appear but is not symbolic of a new evolutionary path. The worst profiles once again are naught but an indescribable random polygon.
The values associated to each figure, as presented in the table 5.8. The improvements for the mean and best profiles show a clear trend of improvement over the first thousand generations but drop slightly by the ten-thousandth generation for the best results, and sharply for the mean results. The worst results show opposite behavior, the gains steadily decreasing over the first thousand generation, then significantly improving until the ten-thousandth generation.

Figure 5.14, which shows the best fitness value (in blue) and the calculated mean fitness value of the population (orange) for each generation. The fitness of the population increases in several large jumps (e.g., generation 1100, 1600, 1900, 3100, and 3800) before reaching a plateau around generation 4000. After this point the improvements are marginal.

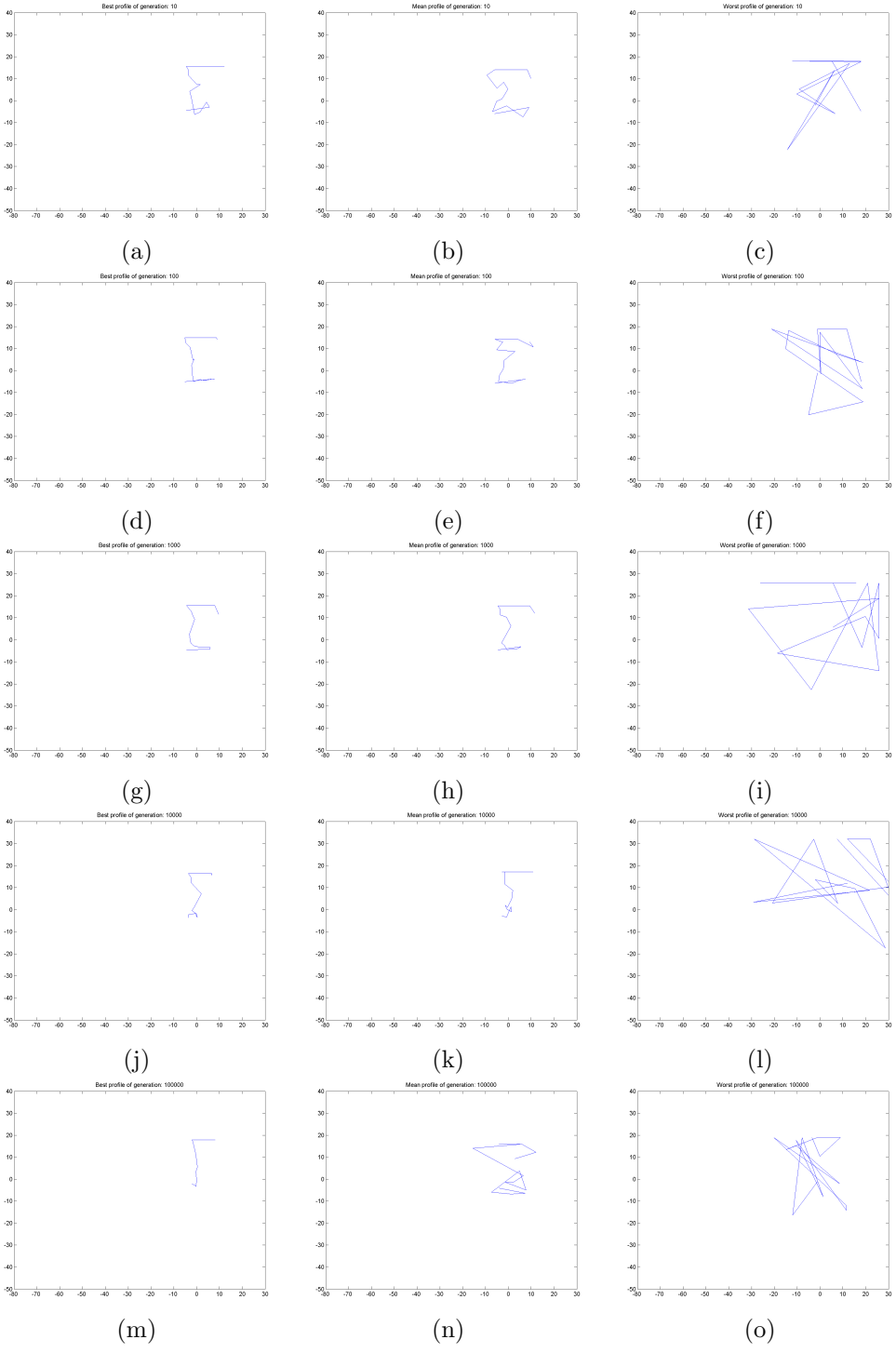

Figure 5.13: Worst, average and best profile after 10, 100, 1000 and 10000 generations for open profiles, with a conserved region, using the provided SAM function.

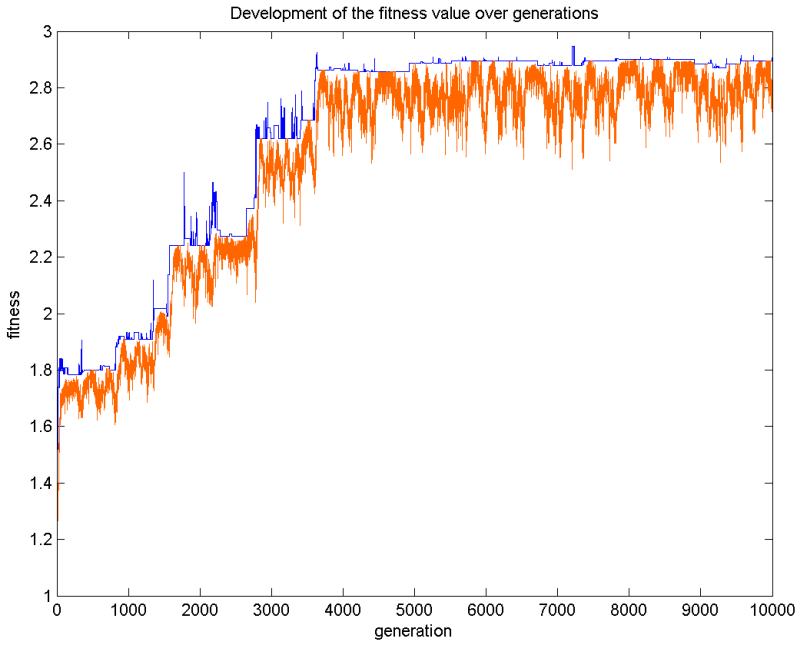

Figure 5.14: Fitness over 10000 generations for open profiles, with conserved regions, using the provided SAM function. The blue line shows the best fitness value of each generation. The orange line shows the mean fitness value of the population for each generation.

5.2 Results with the advanced algorithm

As the hardware for the advanced algorithm was available only late in the development process, with very limited access, there was no time to perform tests after the basic debugging.

Chapter 6

Discussion

6.1 Discussion of the base algorithm

When looking at the values displayed in the tables,there are several occurrences where the worst values are superior to the mean, or even the best values. This is due to the fact, that the quality of profiles (whether they are worst, mean, or best) is performed based on their actual fitness value. The tables however only display the length of the profile edges and the percentage of improvement relative to the reference profiles (the base profiles). Some profiles that are considered the worst in the population posses a short edge-length but also do not satisfy the stability conditions. This leads to a low multiplication factor for the length and subsequently a low fitness value which however does not reflect in the values displayed in the table.
When comparing the cases using the original second area of moment (SAM) function to those using the provided SAM function, two things can be observed. First, while there is no obvious difference between the best profiles in both cases and the difference between the mean profiles of both cases is negligible, the worst profiles are highly distinguishable. In cases using the original SAM function, the worst profiles often share at least some similarities. The worst profiles generated using the provided SAM however are are highly random polygon that do not resemble any familiar shape. The second observation is that the improvements obtained are slightly higher for the cases using the provided SAM function. For example, comparing the cases 7 and 8. The original SAM case starts at around -50% and, over 10 generations, reaches -20%. The provided SAM case starts out better at -22% and actually worsens to -24%. However by generation 10000 the original SAM worsens to -25% whereas the provided SAM case manages to improve to an impressive 48% gain over the reference profile. Going over all cases, those using the provided SAM function either perform at about the same level as those using the original SAM function or better. Considering the fact that the SAM functions are only responsible for the determination

of the stability of a profile in the form of a multiplication factor in the range of 0 to 1, this is quite interesting. It is assumed that the provided SAM, being more demanding to return a high factor, leads to more jagged profiles that fulfil the stability criterion are created and improved upon, potentially leading to a more diverse population and a higher chance of finding a high potential niche. So while both functions satisfy their given purpose of evaluating the stability of profiles, the provided SAM function, through it stringent stability requirements, exerts a presumably diversifying influence on the population as a whole which favours better results in the considered cases. As no negative impact of using this function could be identified during the tests, it is tentatively recommended to use the provided function over the original one.

The other points differentiating the various cases lend themselves less to comparison. The open and closed profiles as well as the conserved regions were considered in the cases as prove of feasibility. For both features, improvements were achieved and while those weren't substantial over the reference profiles, the ability of the presented genetic algorithm to improve upon such profiles is clearly visible between the initial and final generation. While there were at least two runs for open profiles that yielded a positive gain relative to the reference, for the closed profile on the other hand the cases presented in the chapter results (chapter 5) did not provide any examples with a positive improvement percentage. This suggests that the optimization of closed profiles is more challenging. One might speculate as to the reasons for this difficulty lying in an already very good relation of circumference and stability for rectangular profiles.

Even though for two cases positive gains were achieved, most cases did not. However this doesn't mean that other cases cannot be improved to the point of achieving positive gains. During the development a few cases were found. Those were rare and usually occurred during tests aimed at optimizing the configuration of the algorithm. It can be assumed that such results may be reproduced providing two conditions are met. First, the algorithm in its current form must be configured optimally. The current configuration was determined during the development process. However due to a significant number of changes in the final phase which were necessary for the port of the advanced algorithm to the server cluster and subsequent bug fixes, this configuration probably is no longer optimal. The second condition would be the performance of a larger number of runs. Those two conditions could not be met as the sudden availability of a server cluster and the work this entailed disrupted the initial planning of polishing the algorithm during the final phase.

Chapter 7

Conclusion

Event though the results presented in chapter 5 were not quite as good as hoped, this could be explaining. More importantly, everything that this thesis set out to accomplished (see section 1.6) was achieved. A genetic algorithm, capable of optimizing profiles with consideration of its stability was developed. Both open and closed profiles could be and were calculated. Most importantly however was the novel and successful addition of conserved segments to the algorithm which allow users to add their own constraints regarding the form of the profile. The migration of the algorithm to run on distributed systems was also successfully achieved (see the appendix). All challenges presented during the development and all goals achieved, all that remains is the fine tuning of the configuration in order to improve the success rate, which is mostly naught but a time consuming work of diligence.

Chapter 8

Outlook

While the algorithm is considered complete for both versions, base and advanced, there is always room for improvement.

8.1 Configuring genetic algorithm

While some factors, as the mutation and crossover rate have been tested, many more can influence the performance of the genetic algorithm. One frequently arising issue is the problem of premature convergence. The diversity based selection operator was an attempt at countering this. Another solution that could not be implemented in time would be the use of "pseudo random numbers by Lorenz chaotic system" as suggested by [8]. One solution that was considered but could not be developed beyond the stage of an idea was already thoroughly analysed by [32]: usage of interbreeding population of which one emphasises diversity over the fitness-value.

8.2 Code optimization

While the quality of the code developed significantly improved over time due to the experience gathered and large parts of the code already have been replace by more efficient functions, code optimization was at no point a requirement or goal. But now that the algorithm is considered finished and no longer benefits from short development cycles, resource-efficiency becomes more of an issue. One possibility for improvement certainly lies in the code itself. For instance, some of the loops, especially for-loops, might be linearised (that is replace by a single line of code able to perform the same functionality) as Matlab is known for inefficient loop handling. The best way to increase performance would be by changing the programming language to one that is pre-compiled (instead of rum time compilation as in Matlab and Octave) and natively more efficient. C++ or a variation thereof

comes to mind, or, as it is an algorithm consisting mostly of mathematics, FORTRAN variants should prove an excellent choice.

8.3 Expanding to new fields

With the flexibility of the modular functions the algorithm can be quite easily adapted to more, completely new fields. Potentially in cooperation with companies that have already been persuaded of the possibilities of optimization with genetic algorithms. For example optimizing the production process of the beam-profiles generated by the genetic algorithm presented in this thesis.

8.4 Checking the validity of ELMER results

The advanced algorithm (presented in the appendix) uses the finite element method to determine several factors that are incorporated into the fitness formula. As a colleague found out with simple profiles, the accuracy of the results returned by ELMER seems to be dependent on the quality of the 3D mesh. Simple, manually created meshes consisting of simple and regular polyhedron apparently can be used to produce highly precise results. Such meshes are extremely difficult, potentially nigh impossible to create automatically. The more the meshes deviate from simple forms and the complexity grows, the less accurate the results that are generated from them are. If no workaround can be found for this major issue, the usefulness of the advanced genetic algorithm is all but spoiled.

Appendix A

Interface to third-party software

The code so far presents a reliable basic genetic algorithm. As the fitness-calculation are relatively simple, the runtime on an average desktop computer is within an acceptable range. However the precision of the second moment of area by itself is only sufficient to get a first impression of the potential lying in the optimization of a beam profile. To garner more significant results, usable by engineers, a more complex approach must be adopted. This approach culminates in the development of an advanced genetic algorithm presented in this chapter.

This new algorithm, which is based on the previous one and, as a matter of fact, still employs a great number of its functions, distinguishes itself mainly by its fitness-function and some related functions and features. The previous simple formula for the fitness is replaced by one that employs a finite element software (FES) to calculate some values of interest. To manage the drastically increased computational workload of this approach, the algorithm is run on a dedicated server and the fitness function is parallelized by distribution of function calls to several connected server nodes.

This chapter will attempt to elucidate the preprocessing of data in anticipation of the requirements of the finite element software before addressing the initialisation, running and evaluation the results of said software. Once the functioning of a single instance is explained, the manual parallelization will be explained.

The figure A.1 shows the order of the function calls of all custom functions of the algorithm on the headnode. Figure A.2 gives an overview of the custom function on individual nodes.

Before getting into the code, the hard- and software of the servers should be listed for the sake of completeness.

Hardware

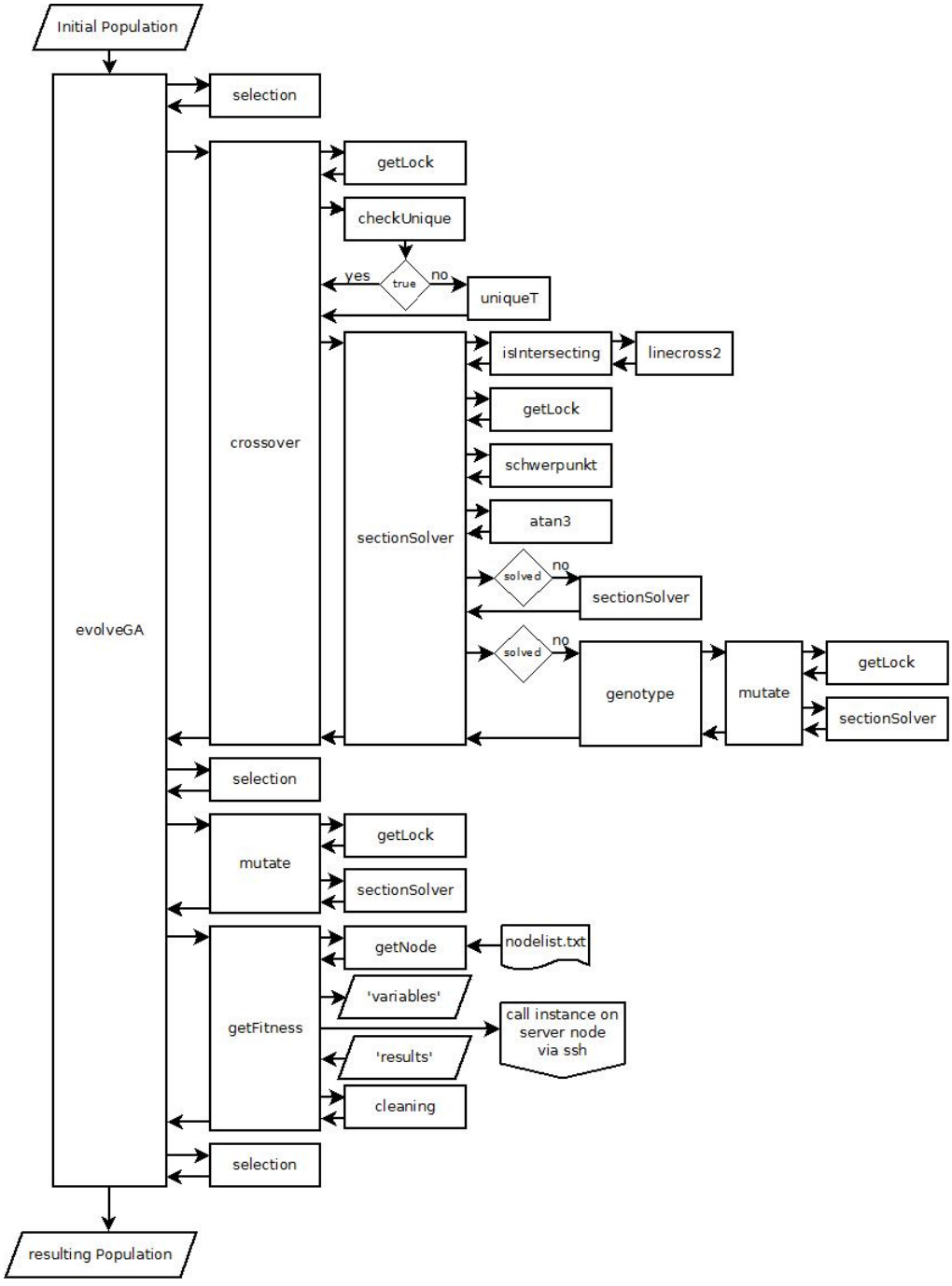

Figure A.1: Stack flowchart of the advanced genetic algorithm depicting the call order and dependency of custom functions on the headnode.

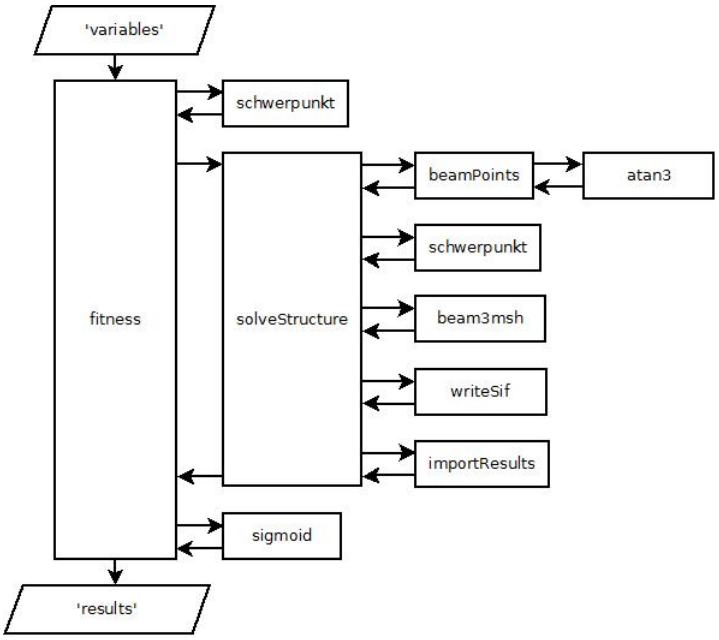

Figure A.2: Stack flowchart of the advanced genetic algorithm depicting the call order and dependency of custom functions on individual nodes.

- Chassis: IBM Bladecenter E
- Headnode: IBM System x3650
- Headnode processor: Intel Xeon 5160 3,0GHz Dual Core (x2)
- Headnode Random Access Memory (RAM): 4GB
- Blades: 14
- Blade processor: Intel Xeon 2,8GHz Dual Core (x2)
- Blade Random Access Memory (RAM): 4GB
- Operating System (OS): Ubuntu Server 12.04 LTS

Software

- Octave 3.6 [7]
- gmsh 2.8.4 [12]
- gmsh 2.8.2 [12]
- ELMER 6.1 [33]

The server is controlled by a headnode that connects to the fourteen individual nodes (or blades) contained in the chassis. The connection is realised via gigabit Ethernet. All of them share their file system via the NFS (network file system) protocol. The NIS (network information service)

protocol enables account sharing over the server-nodes. With this, all nodes can run on the same account. Consequently files created by individual nodes and saved on their default workspace will be available to all other nodes (including the headnode). Commands to the individual nodes are send via the SSH (secure shell) network protocol. There are two versions of gmsh installed as there appears to be a bug in version 2.8.4 occasionally occurring that does not appear with version 2.8.2.

A.1 gmsh

The first step of setting up the profile of the beams for the FES is the conversion of a two dimensional profile to a three dimensional model of the beam. Antedating possible errors during the generation of the mesh due to intersecting profile-lines (this **will** lead to an endless loop in gmsh and is therefore considered unsolvable) is done in two steps: detection of intersections and attempt to solve intersections.

A.1.1 Line intersection

Detecting intersections

As the name implies, this function simply answers the question, whether the profile generated by a specific genome g contains intersecting lines.

Code A.1: isIntersecting.m source code

```
1  function crossed=isIntersecting(g,h)
2   global profile_type
3   if nargin==1
4    if ~profile_type
5     g=g(:,[1:end 1]);
6    end
7    for i=1:length(g)-3
8     for j=i+2:length(g)-1
9      crossed=linecross2(g(:,i),g(:,i+1),g(:,j),g(:,j+1)
           );
10     if crossed, break, end
11    end
12    if crossed, break, end
13   end
14  else
15   if ~profile_type
16    g=g(:,[1:end 1]);
17    h=h(:,[1:end 1]);
18   end
```

```
19    for i=1:length(g)-1
20     for j=1:length(h)-1
21      crossed=linecross2(g(:,i),g(:,i+1),h(:,j),h(:,j+1)
            );
22      if crossed, break, end
23     end
24     if crossed, break, end
25    end
26   end
```

The first few lines (2 to 5) of code A.1 should look familiar by now. Access to the variable *profile_type* is enabled by declaring it in the global scope. It is then used to create the 'wrap-around' by appending the first point if the profile is to be closed.

The actual code is on the lines 7 to 13. The nested for-loops essentially compare each line of the profile to all subsequent lines (except for the immediate following one as it could only intersect by overlapping). The outer-loop iterates through all point but the last three, as one is used with the current point to form one line, and the next two form the subsequent line. The inner loop starts two points beyond the outer-loop and iterates through to the next to last point, skipping the first two points forming the line to which the inner-loop is compared. Each iteration-step calls the function *linecross2* with the four points defining the two lines to be checked as input arguments. The returned value is saved to *crossed* and, as it is expected to be a boolean, is used as the condition of a short-form conditional statement (line 10 and again line 12). If it is **TRUE**, the *break* statement is executed which ends the current loop. The sequence of conditional statements in both loops ensures, that the function is terminated upon encountering even one intersection. As this first intersection is sufficient to cause problems later on, it is considered to be a waste of computing time to investigate further line combinations beyond the first hit.

Code A.2: linecross2.m source code

```
1  function bool=linecross2(a1,a2,b1,b2)
2   bool=false;
3   m1=(a2(2,1)-a1(2,1))./(a2(1,1)-a1(1,1));
4   m2=(b2(2,1)-b1(2,1))./(b2(1,1)-b1(1,1));
5   if checkUnique([a1,a2,b1,b2])
6    if m1==m2
7     o1=a1(2,1)-m1.*a1(1,1);
8     o2=b1(2,1)-m2.*b1(1,1);
9     if (abs(m1)==Inf&&a1(1,1)==b1(1,1)||(o1==o2&&abs(m1
          )~=Inf))&&((all(sign(b1-a1)+sign(b1-a2)==0)||all
          (sign(b2-a1)+sign(b2-a2)==0)||all(sign(a1-b1)+
```

```
        sign(a1-b2)==0)||all(sign(a2-b1)+sign(a2-b2)==0)
        ))
10      bool=true;
11    end
12    elseif abs(m1)==Inf
13    y=m2*a1(1,1)+b1(2,1)-m2*b1(1,1);
14    if y>=min(a1(2,1),a2(2,1))&&y<=max(a1(2,1),a2(2,1))
        &&y>=min(b1(2,1),b2(2,1))&&y<=max(b1(2,1),b2
        (2,1))
15      bool=true;
16    end
17    elseif abs(m2)==Inf
18    y=m1*b1(1,1)+a1(2,1)-m1*a1(1,1);
19    if y>=min(a1(2,1),a2(2,1))&&y<=max(a1(2,1),a2(2,1))
        &&y>=min(b1(2,1),b2(2,1))&&y<=max(b1(2,1),b2
        (2,1))
20      bool=true;
21    end
22    else
23    o1=a1(2,1)-m1.*a1(1,1);
24    o2=b1(2,1)-m2.*b1(1,1);
25    x=(o1-o2)/(m2-m1);
26    if x>=min(a1(1,1),a2(1,1))&&x<=max(a1(1,1),a2(1,1))
        &&x>=min(b1(1,1),b2(1,1))&&x<=max(b1(1,1),b2
        (1,1))
27      y=m1*x+o1;
28      if y>=min(a1(2,1),a2(2,1))&&y<=max(a1(2,1),a2(2,1)
          )&&y>=min(b1(2,1),b2(2,1))&&y<=max(b1(2,1),b2
          (2,1))
29        bool=true;
30      end
31    end
32  end
33 end
```

Now, how does the *linecross2* function actually determine whether the two segments defined by four points intersect? To begin, the return-value *bool* is created and set to **FALSE**. In preparation the slopes of the lines (projections of the respective segments) are determined (lines 3 and 4). Based on the resulting values, four cases are distinguished. First off: the slopes are identical. This means the lines are parallel. To clearly ascertain possible overlaps, the intersection of the lines with the ordinate is calculated (lines 7 and 8). Intersection of the lines is checked by a seemingly complicated conditional statement. This, however, can be broken down into two sets of

conditions, connected by an **AND**-operator and presenting respectively two and four alternatives separated by an **OR**-operator.

Conditions set 1:

$abs(m1) == Inf \&\& a1(1,1) == b1(1,1)$ If $m1$ (and, due to the parallel nature $m2$) is $Infinite$ and the abscissa-coordinates of at least one point from each line coincide, then the two lines are identical and only the overlap of the two segments remains to be determined.

$o1 == o2 \&\& abs(m1) = Inf$ If $m1$ (and, due to the parallel nature $m2$) is not $Infinite$ but the points of intersection with the ordinate are identical, the lines are again shown to be identical and intersection of the segments can be tested by determining the overlap.

Both cases essentially ascertain that the lines are not only parallel, but in fact overlapping. However these two cases must be distinguished as $Infinite$ values for the slopes result in values for the intersection with the ordinate of either $Infinite$ or NaN (Not a Number). But if the slope-values are not $Infinite$, then a match of abscissa-coordinates of points on the two segments is not sufficient geometric constraint.

Conditions set 2:

$all(sign(b1 - a1) + sign(b1 - a2) == 0)$ First point of second segment between points of first segment.

$all(sign(b2 - a1) + sign(b2 - a2) == 0)$ Second point of second segment between points of first segment.

$all(sign(a1 - b1) + sign(a1 - b2) == 0)$ First point of first segment between points of second segment.

$all(sign(a2 - b1) + sign(a2 - b2) == 0)$ Second point of first segment between points of second segment.

As the first condition set already establishes, that both segments are located on the same line, the second set merely checks whether at least one point from one segment lies between the two points defining the other. Subtracting the coordinates of the segment boundaries from a prospective point will yield different leading signs for each. Adding those up will result in 0, thereby confirming the intermediate position of the candidate. Sole exception to this would be either horizontal or vertical segments for which the sign of one coordinate will itself result in 0. Since this however is to be expected for the subtraction of each boundary, the sum will again be 0, satisfying the check for equality to 0. Performing this check in-line within

the *all* function provided by Matlab, ensures that the condition is satisfied for both the abscissa and the ordinate.
Verifying whether the second segments points lie between the points of the first one is not sufficient, as the first segment may be comprised by the second segment entirely. That is why the third and fourth condition were added to the second set.

On to the second case: the slope of the first line is $Infinite$. The intersection y of the second line with the vertical first line is calculated on line 13. As the first line is vertical, this can simply be achieved by inserting the value the first lines abscissa into the function of the second line. Since the intersection with the ordinate has not yet been calculated for this function, this is performed in-line. The conditional statement then verifies, that the intersection point lies within both segments. This is the case if it is bigger than the smaller boundary but smaller than the larger one, for both segments.

The third case is analogous to the second one. Only the second line is $Infinite$ instead of the first one. Accordingly the formula to determine y is adapted. The conditional statement however can be kept without modification.

The last case is the one expected to be encountered most often, as it covers every case that does not involve horizontal, vertical or parallel lines. Once again the ordinate-intersection of each linear function is calculated (lines 23 and 24). Knowing, that the intersection of both lines is the point for which the linear functions of both lines are equal, an equation is written down and solved for x (see A.1).

$$f_1(x) = m_1 * x + o_1 \qquad f_2(x) = m_2 * x + o_2$$

$$\begin{aligned} & f_1(x) = f_2(x) \\ \Rightarrow\ & m_1 * x + o_1 = m_2 * x + o_2 \\ \Leftrightarrow\ & o_1 - o_2 = m_2 * x - m_1 * x \\ \Leftrightarrow\ & o_1 - o_2 = (m_2 - m_1) * x \\ \Leftrightarrow\ & x = \frac{o_1 - o_2}{m_2 - m_1} \end{aligned} \tag{A.1}$$

The resulting value is confirmed to be within the ranges defined by the segments (conditional statement line 26), before inserting it into the linear function of the first line to calculate the corresponding y value. This is then checked to be within the ranges defined by the segments (as a matter of fact the conditional statements on line 14 and 28 are identical) as well.

At the core of each nested conditional statement, the return-value *bool* is set to **TRUE**. When the function ends, only in cases where the system defined by the four input points satisfied all requirements of the various

nested conditionals, will it return **TRUE**. In all other cases, the initial **FALSE**-value is retained.

Solution attempt

Now that intersecting segments in a profile can be found, the question of how to handle those intersections remains. This algorithm opts to attempt to solve such cases. While this at first glance might seem a trivial task, the added constraint of the *regionLock* turns it into a bit of a challenge. Over the course of several weeks, a number of possible solvers were tested with three exemplary individuals (each manual confirmed to be technically solvable) as well as with some random individuals possessing intersecting lines.

Among all tested solutions, not one was able to solve all examples with *regionLock*. Furthermore a few random individuals tested were found to be unsolvable, even manually. Even though interesting from a mathematical point of view, the question of absolute resolvability of point ordering problems in a random polygon with partially invariable ordering exceeds the scope of this thesis. That is why the function *sectionSolver* is only considered an attempt at solving intersections. Among all tested solutions, one was chosen (see code A.3) based on the speed and the number of test-cases it could successfully solve.

Code A.3: sectionSolver.m source code

```
1  function g=sectionSolver(g,v,order)
2   crossed=isIntersecting(g);
3   if crossed
4    if nargin<3
5     order=1;
6    end
7    h=getLock(g,v);
8    [s,~]=schwerpunkt(g);
9    a=atan3(g(2,:)-s(2),g(1,:)-s(1));
10   [a,ind]=sortrows(a',order);
11   a=a';
12   ind=ind';
13   for i=1:length(g)
14    pos=find(find(a(1,:)==a(1,i)));
15    if length(pos)>1
16     dist=zeros(3,length(pos));
17     dist(1:2,1:length(pos))=g(:,pos);
18     dist(3,:)=sqrt(((g(2,pos)-s(2)).^2)+((g(1,pos)-s
           (2)).^2));
19     [dist,ind2]=sortrows(dist',3)';
```

```
20      dist=dist';
21      a(:,pos)=dist(:,:);
22      ind(1,pos)=ind2;
23     end
24    end
25    for i=1:size(h,2)
26     fi=h(1,i);
27     li=h(1,i)+h(2,i)-1;
28     fsi=find(ind==fi,1,'first');
29     lsi=find(ind==li,1,'first');
30     if fsi<lsi
31      for j=fi+1:li
32       ind(find(ind==j))=[];
33      end
34      fsi=find(ind==fi,1,'first');
35      ind=[ind(1:fsi) (fi+1:1:li) ind(fsi+1:end)];
36     else
37      for j=li-1:-1:fi
38       ind(find(ind==j))=[];
39      end
40      lsi=find(ind==li,1,'first');
41      ind=[ind(1:lsi) (li-1:-1:fi) ind(lsi+1:end)];
42     end
43    end
44    if isIntersecting(g(:,ind))&&order~=-1
45     g=sectionSolver(g,v,-1);
46    elseif isIntersecting(g(:,ind))&&order==-1
47     while isIntersecting(g)
48      g=genotype(length(g),h,v);
49     end
50    else
51     g=g(:,ind);
52    end
53   end
```

This function has one mandatory input arguments g (a genome). The argument v contains the *regionValues*) which are translated into the corresponding regionLock h on line 7. The third argument *order* is optional. It is expected to be either -1 or 1. If this argument is not given, it is set to 1 as a default value on line 5. The center of mass is calculated (equivalent to the centroid as no weights for the points or connecting segments are defined) and the angle between the horizontal and each point in the center of mass. The angle is determined by a custom function *atan3*. This function calculates angles in radians with Matlab's own *atan2* function and converts

the results from a domain of $[-\pi, \pi]$ to $[0, 2\pi]$ for convenience of use. The angles are then sorted by the Matlab function *sortrows*. As its name implies, it can only sort rows, therefore the matrix containing the angles must be transposed prior to its used (which is done in the function call here). The variable *order* determines whether the sorting is ascending ($order == 1$) or descending ($order == -1$). By giving the function two output variables, not only the sorted input matrix is returned (a) but also a list containing for each position the index of the value prior to the sorting (ind).
Next, identical angles are sought out by iterating through the genome. If the value of the angle at the current position of the iteration is found more than once, the euclidean distance for each is calculated. Those values are again sorted. The values in the matrix of angles as well as the indices are then updated accordingly.
So far the function is quite simple and straight forward. The second for-loop solely concerns itself with the *regionLock*. It iterates through the columns of the *regionLock* matrix, creating *fi*, a variable containing the index of the first position to be protected, and *li*, a variable holding the last position to be protected. Their current position in the sorted matrix is determined by Matlab's *find* function and saved to the variables *fsi* and *lsi* respectively. If *fsi* is smaller than *lsi*, a for-loop iterates through all intermediate points between those two and deletes them from the *ind* variable. Afterwards *fsi* is redetermined (in case the removal of interposed indices affected this value). *ind* is then rewritten by creating a matrix receiving its current values up to the position *fsi*, then the indices of the positions protected by *regionLock*, and finally all of its remaining previous values.
If *fsi* should be larger than *lsi*, an essentially similar statement is executed, with the slight difference, that the use of these two variables is reversed. This is done in a bid to follow an assumed orientation of the points to decrease the risk of creating new intersections.
The results of the solver are checked in a conditional statement with three distinguished cases. If the reordered genome is still intersecting and the sorting order was ascending ($order == 1$) the function is called again, however with a third input argument -1 (descending sort-order). If the reordered genome is still intersecting and the order was descending, a new genome is created with the *genotype* function. This is repeated until a new genotype is created, that does not intersect. This is a drastic solution, that guarantees that no intersecting genomes are kept as those would lead to errors preventing the algorithm from terminating properly. The last case assumes that the genome is no longer intersecting, as it can only be reached if the previous two cases do not apply. Then the genome g is overwritten with the newly ordered values and returned.

A.1.2 Inner and outer wall of a beam

As was mentioned earlier during the explanation of the *ftm* function (see section 4.6.2), the profile as given by the genome is considered to be the middle line of the walls of a beam with a specific thickness. While the *ftm* function simply extends the wall perpendicularly from this middle-line and tolerates the small error in the results created by overlapping corners of walls, the advance algorithm approach aims at creating as exact and precise a result as possible. Therefore it is important to properly create the actual walls. This function still extends the walls perpendicularly from the middle line, but instead of just assuming a rectangular form for each segment, the lines of the walls are extended to or cut at their first intersection with the wall-lines of neighbouring segments.

Code A.4: beamPoints.m source code

```
1  function [outer,inner]=beamPoints(frame,wall)
2   if nargin<2
3    wall=0.002;
4   end
5   frame=frame/100;
6   front=[2:length(frame) 1];
7   previous=[length(frame) 1:length(frame)-1];
8   theta=atan3(frame(2,front)-frame(2,:),frame(1,front)-
        frame(1,:));
9   gamma=atan3(frame(2,previous)-frame(2,:),frame(1,
        previous)-frame(1,:));
10  alpha=gamma-theta;
11  beta=(alpha/2)-(pi/2);
12  hypotenuse=(wall/2)./(cos(beta));
13  delta=theta+(alpha/2);
14  x=cos(delta).*hypotenuse;
15  y=sin(delta).*hypotenuse;
16  outer=frame-[x;y];
17  inner=frame+[x;y];
```

The function *beamPoints* has one mandatory input argument: *frame* (a genome), and an optional one:*wall* (single number declaring the desired wall-thickness in metres), and returns the variables *outer* and *inner* (respectively matrices containing the coordinates for an outer and an inner wall with encoding identical to the genomes). If the optional argument is not given, *wall* is set to 0.002. As standard units will be needed later on, 1 is assumed to be one metre, correspondingly this value represents 2 millimetre. Assuming, that the units of the genome so far have been centimetres, *frame* is divided by 100 to scale its values to the metre standard unit. With *front* and *previous* two indices-lists are created offset from each other by one position. Once

again this is used to perform operations between neighbouring positions of matrices for which the lists are used as pointers. *theta* calculates the angle in each point from a horizontal line to the following point, while *gamma* determines the angle in each point from a horizontal line to the previous point. Both use the custom function *atan3* (which was already covered in section A.1.1). *alpha* determines the angle between the two segments of which a point is a part. *beta* measures the angle bisector and adapts the value to a vertical reference instead of a horizontal line (as can be seen in figure A.3, this is done to get the angle $\widehat{onw_2}$). As can be seen in figure A.3 o, n and w_2 form a right triangle in w_2. The length of the segment $\overline{n \quad o}$ can be determined by calculating the length of said triangles hypotenuse. This is done on line 12 of the code listing A.4 and saved in a variable conveniently named *hypotenuse*. *delta* gives the angle between the segment $\overline{n \quad o}$ and the horizontal. Now picturing an additional right triangles o, o_x, n (right angle in o_x). The distances $|o_x, n|$ and $|o, o_x|$ (equivalent to $|o_y, n|$) can be calculated by the trigonometric formulas solved for the respective side of the triangles (see formulas A.2 and A.3). The resulting values for x and y can be either subtracted to get the coordinates for the *outer* wall or added to get the coordinates for the *inner* wall of the profile.

$$cos(\delta) = \frac{|o_x, n|}{hypotenuse} \Leftrightarrow |o_x, n| = cos(\delta) * hypotenuse \tag{A.2}$$

$$sin(\delta) = \frac{|o, o_x|}{hypotenuse} \Leftrightarrow |o, o_x| = sin(\delta) * hypotenuse \tag{A.3}$$

A.1.3 Structure of a *.geo file

The geometry file type (*.geo) is a plain-text file containing specifically formatted information that can be interpreted by software supporting the file type. It is also one of the file types supported by gmsh (and subjectively the easiest to write with Matlab and Octave). Thus this section will break down the data formatting. Afterwards the process of automatically generating such a file from a set of input arguments will be explained.

Writing a *.geo file is akin to drawing or constructing the figure in a CAD-software (computer aided design). Although the software supports a great number of commands, this section will only introduce the structure employed by the advanced genetic algorithm. This doesn't fully exploit the full possibilities or capabilities of gmsh, but presents an easy to understand concept that suits the needs of the algorithm, namely being functioning and fairly easy to debug without hindering the overall performance.

First, all points that are to be used must be defined. This is done with lines of code following the pattern 'Point(n) = x,y,z,l;'. n being a continuously counted integer by which each point will be uniquely identifiable and referable later. x, y, and z are numbers (either integers or floating point

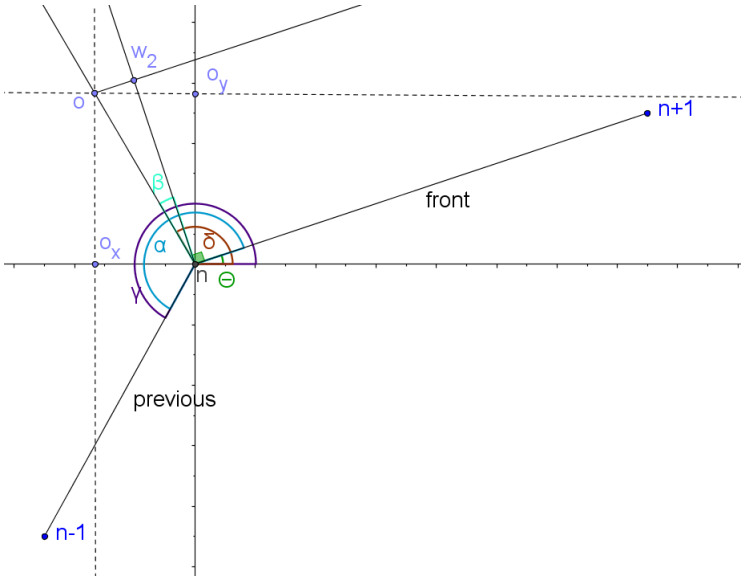

Figure A.3: Diagram of the geometric construction underlying the *beamPoints* function exemplified for the outer wall. n is representative for the point who's outer wall projection is currently being determined. Accordingly *n-1* is the previous point and *n+1* is the next one. The marked horizontal and vertical lines through n are axes of a Cartesian coordinate system. Θ is the angle of the segment $\overline{n \quad n+1}$ (*front*) to abscissa in n. γ is the angle of the segment $\overline{n \quad n-1}$ (*previous*)to the abscissa in n. α is the angle between the two segments connected by n (despite its representation here it is never a reflex angle due to the way it is calculated in the function). δ is the angle bisector of γ. β is the angle between the angle bisector of α and an auxiliary ray perpendicular to the segment *front*. This ray crosses another ray parallel to *front* in w_2 (this is a representation of the outer wall of the segment *front* situated at half a wall-thickness from the actual segment. It intersects with the auxiliary ray marking the bisector angle α in o. The dashed horizontal and vertical lines project this point to the axes. Their respective intersections are marked with o_x and o_y showing the distances by which n must be modified to get the coordinates for o.

numbers of variable length) that give the three dimensional coordinates of each point. The last variable l is expected to be a number (or a variable name previously defined in the file) and defines a characteristic length (also called field size). This can be seen as a target size for elements connected to the point. Of course, depending on the geometry, this target can not always be matched and calculations comparing it to the actual attained values are used to gauge the quality of the mesh. Smaller values will result in more meticulous, but also complexer meshes, leading in turn to more rigorous results at the cost of an increased computing time. A good compromise would be to use smaller characteristic lengths close to intricate, complicated, or narrow structure elements, while handling large continuous areas and volumes with large field size. The values can indeed be defined differently for each point.
Lines are then defined by the line of code 'Line(m) = i,j;'. m being a separate numeration from n (it can start at 1 like n without causing errors, as apparently points and numbers a stored distinctly even if they essentially have an identical variable). i and j are to be replaced with integers of previously defined points. Thus each line is defined as an **oriented** connection between two points in space.
The creation of surfaces in itself is simple, but requires so called line-loops. The code sequence for both is as follows 'Line Loop(m) = i,i+1...,j-1,j;Plane Surface(m)=m;'. The counter m of the line-loops is simply continued from the count of the lines. i to j is a comma separated list of all lines defining the loop. It is of utmost importance to pay close attention to the orientation of each line and to ensure a proper connection between the lines. If a line ends with the point i for instance, then the following line **must** begin with i as well. The last line in the list must end on the same point on which the first line begins. As each line is oriented, it should be noted, that the orientation can be inverted by prefacing the number with a minus symbol. The plane surface itself can take on the same number as the line-loop defining it (making it easier to write.
Just as surfaces needed a line-loop to be defined, a volume requires a surface-loop. The code 'Surface(m)=i,i+1,...,j-1,j;Volume(m)=m;' is similar enough to the definition of plane surfaces to consider additional explanations to be gratuitous.
This concludes the geometric construction of a three dimensional model. However since the FES cannot operate on geometric entities, physical surfaces and volumes must be defined. Surfaces can be created by the line 'Physical Surface(o)=u,u+1,...,v-1,v;' while volumes are defined by 'Physical Volume(o)=m'. Their name o can be a number or a string delimited by quotation marks. Physical surfaces are created by a comma separated list of plane surfaces (or in some cases a single plane surface). Physical volumes are created by equating it to a volume (as previously defined by surface loops).

This concludes the structure of a file that can serve as input for the creation of a mesh.

A.1.4 Automation of the *.geo file creation

The function *beam3msh* is the implementation of the automated creation of *.geo files. It does not return anything, and requires four mandatory arguments and can take one optional argument (see code A.5). *inner* and *outer* are two genomes respectively containing the coordinates of the inner and the outer wall of the profile as created by the function *beamPoints*. *l* is expected to be a number (integer of floating point) giving the length of the beam to create from the profile. This value is assumed to be scaled to the unit of metres. *filename* must be a string that will be used as the file-name (the file-type is added by the function itself). The optional argument *edge* determines the characteristic length to be used. Even though, as was discussed in the previous section, variable size fields can be set for each point, this implementation sets the same value for all points. If the argument is omitted, a default value of 0.1 is given. Besides the given input arguments, access to the variable *profile_type* in the global scope is once again given.

Code A.5: Function declaration of beam3msh.m

```
1 function beam3msh(inner,outer,l,filename,edge)
2  if nargin<5
3   edge=0.1;
4  end
5  global profile_type
```

To write a file the code A.6 is used. The function *fopen* is given by Matlab. Its first input argument is the file-name of the file (in this case the proper file-type is appended in-line). If a file with the name already exists, it is opened, if it doesn't exist it is created. The second argument is optional and determines the access permission to the file. It is always a string, though the possible values are limited (refer to the help page of *fopen* for a comprehensive list of possible values). The character 'w' is used to request the right to write to, or overwrite (if the file already exists and has content), the file. The function returns an identifier to the opened file. The content of the file is written by the Matlab function *fprintf* using two input arguments. First the identifier is given, providing a target for the function. The second argument is the string to be written. It is important to ensure that all data given in this argument is properly formatted as a string. Numbers must be converted using the *num2str* function of Matlab.

Code A.6: Creation of a new writeable file in beam3msh.m

```
1 fid=fopen([filename '.geo'],'w','native');
```

As explained in the previous section, the points are the first geometrical elements to be written. This is done by four consecutive for-loops (see code A.7). The first one creates the points forming the outer wall of the profile using the abscissa and ordinate coordinates from *outer* as values for x and y and setting z to 0. The characteristic length is set by *edge*. The line terminates with the control characters '\n', thus imposing a newline (geometry files written entirely in a single line will not be converted properly by gmsh).
The second loop continues the numeration of the points where the previous loop left of and creates the points of the inner wall from the coordinates in *inner*. The z value is set to 0 in this loop as well.
The third and fourth loops are essentially repetitions of the first two with continued numbering but with z set to the value given by the input argument *l*.

Code A.7: Adding points to the geometry with beam3msh.m

```
1  for i=1:length(outer)
2   fprintf(fid,['Point(' num2str(i) ')␣=␣{' num2str(
       outer(1,i)) ',␣' num2str(outer(2,i)) ',␣0,␣'
       num2str(edge) '};\n'])
3  end
4  for i=length(outer)+1:length(outer)*2
5   fprintf(fid,['Point(' num2str(i) ')␣=␣{' num2str(
       inner(1,i-length(outer))) ',␣' num2str(inner(2,i-
       length(outer))) ',␣0,␣' num2str(edge) '};\n'])
6  end
7  for i=(2*length(outer))+1:length(outer)*3
8   fprintf(fid,['Point(' num2str(i) ')␣=␣{' num2str(
       outer(1,i-(2*length(outer)))) ',␣' num2str(outer
       (2,i-(2*length(outer)))) ',␣' num2str(l) ',␣'
       num2str(edge) '};\n'])
9  end
10 for i=(3*length(outer))+1:(length(outer)*4)
11  fprintf(fid,['Point(' num2str(i) ')␣=␣{' num2str(
       inner(1,i-(3*length(outer)))) ',␣' num2str(inner
       (2,i-(3*length(outer)))) ',␣' num2str(l) ',␣'
       num2str(edge) '};\n'])
12 end
```

The rest of the code writing the file is nested in a conditional statement depending on the value of *profile_type*. The principles will be described using the case of a closed profile. Differences will be noted where they occur. First the lines forming the profiles at $z = 0$ are written, the outer wall before the inner one ((for open profiles, this is done in one step as both profiles

are actually connected). This is done by iterating through each line to be created with for-loops (see code A.8). To make the return to the first point easier, lists of indices named *points* are created before each for-loop. Next the same is done for the profiles at $z = l$. Once all profiles are created, the lines connecting them are created, starting with the first point of the outer wall, proceeding in the order of numbering.

Code A.8: Adding lines to the geometry with beam3msh.m

```
1  points=[1:length(outer) 1];
2  for i=1:length(outer)
3   fprintf(fid,['Line(' num2str(i) ')␣=␣{' num2str(
       points(i)) ',␣' num2str(points(i+1)) '};\n']);
4  end
5  points=[length(outer)+1:(2*length(outer)) length(
     outer)+1];
6  for i=length(outer)+1:length(outer)+length(inner)
7   fprintf(fid,['Line(' num2str(i) ')␣=␣{' num2str(
       points(i-length(outer))) ',␣' num2str(points(i-
       length(outer)+1)) '};\n']);
8  end
9  points=[(2*length(outer))+1:(3*length(outer)) (2*
     length(outer))+1];
10 for i=(2*length(outer))+1:(3*length(outer))
11  fprintf(fid,['Line(' num2str(i) ')␣=␣{' num2str(
       points(i-(2*length(outer)))) ',␣' num2str(points
       (i+1-(2*length(outer)))) '};\n']);
12 end
13 points=[(3*length(outer))+1:(4*length(outer)) (3*
     length(outer))+1];
14 for i=(3*length(outer))+1:(4*length(outer))
15  fprintf(fid,['Line(' num2str(i) ')␣=␣{' num2str(
       points(i-(3*length(outer)))) ',␣' num2str(points
       (i-(3*length(outer))+1)) '};\n']);
16 end
17 for i=1:length(outer)
18  fprintf(fid,['Line(' num2str(i+(4*length(outer))) '
       )␣=␣{' num2str(i) ',␣' num2str(i+(2*length(outer
       ))) '};\n']);
19 end
20 for i=length(outer)+1:(2*length(outer))
21  fprintf(fid,['Line(' num2str(i+(4*length(outer))) '
       )␣=␣{' num2str(i) ',␣' num2str(i+(2*length(outer
       ))) '};\n']);
22 end
```

The third element type, the line-loops are slightly more complicated to write as they consist of a larger number of lines to be referenced that do not necessarily come in the same order as their numbering. Before each line-loop, an empty matrix *textstream* is created (see code A.9). Through for-loops the numbers (referencing the lines) are added as strings separated by commas. Once all lines, save one, are added, the line loop is added with the *textstream* being inserted at the appropriate place followed by the last line (this one is added outside the for-loop since it doesn't need to be followed by a comma). The creation of plane surfaces immediately follows the creation of the line loops they are based upon. For convenience their numbering does not increment from the line loops but uses the same number respectively. The first loop and therefore surface to be created is the face of the profile at $z = 0$ (called left side), followed by the face of the profile at $z = l$ (called right side). Each profile of the left side is then connected to the corresponding segment on the right side (each segment is done individually as curved surfaces can only be created in situations satisfying certain strict constraints, which is not guaranteed to be the case for every profile). This is first done for the outer wall (also called outer shell), then for the inner wall (similarly also called inner shell).

Code A.9: Adding line-loops and their corresponding plane surfaces to the geometry with beam3msh.m

```
1  textstream=[];
2  for i=1:length(outer)
3   textstream=[textstream num2str(i) ', '];
4  end
5  for i=length(outer)+1:length(outer)+length(inner)-1
6   textstream=[textstream num2str(-i) ', '];
7  end
8  fprintf(fid,['Line Loop(' num2str((6*length(outer))
     +1) ') = {' textstream num2str(-(length(outer)+
     length(inner))    ) '};\n']);
9  fprintf(fid,['Plane Surface(' num2str((6*length(
     outer))+1) ') = {' num2str((6*length(outer))+1) '
     };\n']);
10 textstream=[];
11 for i=(2*length(outer))+1:(3*length(outer))
12  textstream=[textstream num2str(i) ', '];
13 end
14 for i=(3*length(outer))+1:(4*length(outer))-1
15  textstream=[textstream num2str(-i) ', '];
16 end
```

```
17   fprintf(fid,['Line Loop(' num2str((6*length(outer))
       +2) ') = {' textstream num2str(-(4*length(outer))
         ) '};\n']);
18   fprintf(fid,['Plane Surface(' num2str((6*length(
       outer))+2) ') = {' num2str((6*length(outer))+2) '
       };\n']);
19   points=[1:length(outer) 1];
20   for i=(6*length(outer))+3:(7*length(outer))+2
21    pos=i-((6*length(outer))+2);
22    fprintf(fid,['Line Loop(' num2str(i) ') = {'
        num2str(points(pos)) ', ' num2str(points(pos+1)
        +(4*length(outer))) ', ' num2str(-(points(pos)
        +(2*length(outer)))) ', ' num2str(-(points(pos)
        +(4*length(outer)))) '};\n']);
23    fprintf(fid,['Plane Surface(' num2str(i) ') = {'
        num2str(i) '};\n']);
24   end
25   for i=(7*length(outer))+3:(8*length(outer))+2
26    pos=i-((7*length(outer))+2);
27    fprintf(fid,['Line Loop(' num2str(i) ') = {'
        num2str(points(pos)+length(outer)) ', ' num2str(
        points(pos+1)+(5*length(outer))) ', ' num2str(-(
        points(pos)+(3*length(outer)))) ', ' num2str(-(
        points(pos)+(5*length(outer)))) '};\n']);
28    fprintf(fid,['Plane Surface(' num2str(i) ') = {'
        num2str(i) '};\n']);
29   end
```

The volume is created in a similar way, using a *textstream* listing all the plane surfaces enclosing it to create the surface loop which is then used to define the volume itself (see code A.10). This concludes the creation of the geometric elements. The creation of the physical elements is much easier as it merely requires to attribute the geometric element to a definition of physical surfaces and a physical volume respectively (see code A.11).

Code A.10: Adding a surface-loop and its corresponding volume to the geometry with beam3msh.m

```
1  textstream=[num2str((6*length(outer))+1)];
2  for i=2:(2*length(outer))+2
3   textstream=[textstream ', ' num2str((6*length(outer)
      )+i)];
4  end
5  fprintf(fid,['Surface Loop(' num2str((8*length(outer)
     )+3) ') = {' textstream '};\n']);
```

```
6  fprintf(fid,['Volume(' num2str((8*length(outer))+3) ')=-{' num2str((8*length(outer))+3) '};\n']);
```

Code A.11: Creating physical entities from geometric elements with beam3msh.m

```
1  fprintf(fid,['Physical Surface("bottom")=-{' num2str((6*length(outer))+1) '};\n']);
2  fprintf(fid,['Physical Surface("top")=-{' num2str((6*length(outer))+2) '};\n']);
3  textstream=[num2str((6*length(outer))+3)];
4  for i=4:length(outer)+2
5   textstream=[textstream ',-' num2str((6*length(outer))+i)];
6  end
7  fprintf(fid,['Physical Surface("outer shell")=-{' textstream '};\n']);
8  textstream=[num2str((7*length(outer))+3)];
9  for i=4:length(outer)+2
10  textstream=[textstream ',-' num2str((7*length(outer))+i)];
11 end
12 fprintf(fid,['Physical Surface("inner shell")=-{' textstream '};\n']);
13 fprintf(fid,['Physical Volume("beam")=-{' num2str((8*length(outer))+3) '};\n']);
```

Once all information has been written to the file, it must be closed before being fully accessible to other programs. Gmsh is executed from within the *beam3msh* function via a system-call with the string composed of 'gmsh ', the file name (without the type-ending) and '.geo -3' (see code A.12). This calls the program gmsh with the file name (the type being appended to it) and '-3' as starting parameters. The file name simply determines which file to upon upon program-start. The parameter '-3' determines the type of mesh to be generated to be 3D.

Code A.12: File closure and starting gmsh

```
1  fclose(fid);
2  try
3   system(['gmsh_old/bin/gmsh- ' filename '.geo -3']);
4   while exist([filename '.msh'])==0
5    system(['gmsh_stable/bin/gmsh- ' filename '.geo -3']);
6   end
```

```
7  catch
8    error('beam3msh: An error occured during the
         creation of the mesh');
9  end
```

A.2 ELMER

The underlying principle of the finite element approach to approximate the solution of differential problems (often of a physical nature) is the discretisation of its actors. This means discretising the complex large domain into smaller and simpler elements and performing the analysis for each before determining an approximate result based on the individual results. In structural engineering for example a three-dimensional model of the structure to be analysed can be turned into a mesh. Each polyhedron of the mesh can then be solved with relative ease due to the simpler geometry. Based on the results of these individual elements the approximate result for the whole model can be determined (often by integration).

ELMER is an open source finite element software that, according to its website, was originally developed by the CSC - IT Center for Science (a Finnish non-profit organisation directed by the nations ministry of education) in collaboration with Finnish research institutions (academic as well as industrial). While it does support a wide selection of physical model, the support of structural mechanics is the main reason for choosing this program over alternatives (among the considered programs, ELMER was the only one to mention this feature explicitly on its front-page).

ELMER consists of several independent modules: ElmerGUI (the graphical user interface), ElmerSolver (the actual finite element software), ElmerPost (a post-processor to view the results), ElmerGrid (a basic mesh manipulation program), ElmerFront (a deprecated GUI), Mesh2D (a mesh generation tool used by ElmerFront), and ViewFactors (a program usually only required by some radiation problems).

A.2.1 The command file

While ELMER does come with a graphical user interface (GUI), it can be run from a command line (i.e. terminal on linux operating systems). To do this, however a so called command-file is necessary. This file contains the configuration for ELMER, e.g. paths to the mesh files, type of calculations to be performed, values of constants, solver configurations, definitions for equations, materials, bodies, and boundary conditions. This file is of the type *.sif ("solver input file") and in our case receives its name from the input parameter *filename*.

The command-file consists of various sections. The sections are started by printing their name (for a comprehensive list of all names supported by the software see the ElmerSolver manual), and ends at the first occurrence of the keyword 'End'. Inside of each section a set of keywords and numbers or strings as their given values, define the configuration of the section. The supported keywords depend on the section.

This chapter will elucidate the role of each section of the command file used in the algorithm and the meaning of many, if not most of the keywords and their setting.

The header contains mostly path-information, including load-paths for the meshes and paths for the default-output. The first line inside the section 'Mesh DB "." "."' sets the location of the mesh files. In between the first set of quotation marks a preceding path (if the files are not located in the working directory or its sub-folders) is given. In between the second set of quotation marks the name of the directory containing the files is set. This line is actually mandatory in the header section. The second line 'Results Directory ""' permits the specification of a directory in which to write the result-files. If left empty, the working directory is used. Even though this line is not explicitly required and is left empty, it was kept in case it would become necessary after the transfer of the algorithm to a server-cluster.

Code A.13: Code to write the header section of a *.sif file with *writeSif.m*

```
1 fdisp(fid,'Header');
2 fdisp(fid,'Mesh␣DB␣"."␣"."');
3 fdisp(fid,['Mesh␣DB␣"."␣"./mesh-' filename '"']);
4 fdisp(fid,'End');
```

The simulation section contains configurations pertaining to the system of the solver, that is, more specific than the information in the header but still common enough to affect all (or at least most) solvers. 'Max Output Level' determines how much information is output to the computer terminal. All messages of ELMER are attributed a level with 1 being very important and 42 (seemingly the highest level) being unimportant. The maximum level defines the highest level of information still being shown. To paraphrase a CSC course on command files [42], setting it to 1 makes the solver 'talkative like a Finnish lumberjack' while setting it to 42 will make it return 'all and everything'. The 'Coordinate System' is self-explanatory and set to Cartesian. The 'Coordinate Mapping(3)' determines the direction of the axes of the coordinate system (the number in parenthesis turn the variable into an array of length three). The order '1 2 3' will result in the same orientation of the mesh as the one given by the coordinates of the geometry file. 'x' being the width of the profile, 'y' its height and 'z' the length of the beam. Note that within ElmerSolver forces are usually introduced by their keyword followed by either of the numbers 1, 2, or 3, with those following

the orientation defined by the coordinate mapping. The 'Simulation Type = Steady state' determines that a solution is sought after all changes occurred and that can be assumed to be stable and unchanging over time. This is opposed to the 'transient' option that would seek results as the system changes. 'Steady State Max Iterations' defines how often the calculations are to be performed (setting this to more than 1 is of more interest when simulating transiently). The 'Output Intervals' define how often results should be generated. As the number of iterations is set to 1 having more than 1 output is not necessary. 'Timestepping Method' defines the way time is discretised. It can be either be set to 'Crank-Nicolson' or to 'BDF' (backward differentiation formula). The second option was mainly chosen in following the example of a tutorial for linear elasticity simulation in two dimensions. The same goes for the 'BDF Order'. 'Post File' sets the file-name for the file containing the data for the post-processor (ElmerPost). It is generated using the input string *filename*.

Code A.14: Code to write the simulation section of a *.sif file with *writeSif.m*

```
1  fdisp(fid,'Simulation');
2  fdisp(fid,'Max Output Level = 5');
3  fdisp(fid,'Coordinate System = Cartesian');
4  fdisp(fid,'Coordinate Mapping(3) = 1 2 3');
5  fdisp(fid,'Simulation Type = Steady state');
6  fdisp(fid,'Steady State Max Iterations = 1');
7  fdisp(fid,'Output Intervals = 1');
8  fdisp(fid,'Timestepping Method = BDF');
9  fdisp(fid,'BDF Order = 1');
10 fdisp(fid,['Solver Input File = ' filename '.sif']);
11 fdisp(fid,'Post File = case.ep');
12 fdisp(fid,'End');
```

The constant section is quite self explanatory. The only constant needed for linear elasticity is gravity. It is created as array of length four. The first three values essentially give a sign for each orientation (see coordinate mapping earlier). The first and third (equivalent to width and length) are set to 0, meaning that the constant does not act in those orientations. The second value is set to -1. So gravity will act in that orientation, but due to the negative algebraic sign it will act in opposite direction (iëdownwards). The value of the constant itself is given by the last value 9.82 (units are not given as Standard Units are expected).

Code A.15: Code to write the constants section of a *.sif file with *writeSif.m*

```
1  fdisp(fid,'Constants');
2  fdisp(fid,'Gravity(4) = 0 -1 0 9.82');
3  fdisp(fid,'End');
```

The body section defines physical bodies in the model. There can be more than one at a time. That is why the keyword is followed by a number (numbering starts at one and should be continuous). 'Target Bodies' associates a body existing in the model to the section. The number in the parenthesis turns the variable into an array (in this case of size one). The body 5 is associated. Technically all physical entities defined in the geometry are available as body for such sections and in the same order. As the order and the number of physical entities created are always the same it is known that the first four are surfaces (left profile, right profile, outer wall, and inner wall) and that the fifth one is the volume. 'Equation' determines which equation acts upon this body. The same goes for the 'Body Force' and for 'Material' (although this does not act upon the body but applies properties). The number always references the count of the section of the same name (so here the first 'Equation' section is applied).

Code A.16: Code to write a body section of a *.sif file with *writeSif.m*

```
1  fdisp(fid,'Body 1');
2  fdisp(fid,'Target Bodies(1) = 5');
3  fdisp(fid,'Name = "Body 1"');
4  fdisp(fid,'Equation = 1');
5  fdisp(fid,'Material = 1');
6  fdisp(fid,'Body Force = 1');
7  fdisp(fid,'End');
```

The first solver section selects and configures a stress solver for a linear elasticity equation. This entire section is taken verbatim from the tutorial on linear elasticity in two dimensions.

Code A.17: Code to write the linear elasticity solver in a *.sif file with *writeSif.m*

```
1  fdisp(fid,'Solver 1');
2  fdisp(fid,'Equation = Linear elasticity');
3  fdisp(fid,'Procedure = "StressSolve" "StressSolver"')
       ;
4  fdisp(fid,'Variable = -dofs 3 Displacement');
5  fdisp(fid,'Exec Solver = Always');
6  fdisp(fid,'Stabilize = True');
7  fdisp(fid,'Bubbles = False');
8  fdisp(fid,'Lumped Mass Matrix = False');
9  fdisp(fid,'Optimize Bandwidth = True');
10 fdisp(fid,'Steady State Convergence Tolerance = 1.0e
       -5');
11 fdisp(fid,'Nonlinear System Convergence Tolerance =
       1.0e-7');
```

```
12 fdisp(fid,'Nonlinear System Max Iterations = 1');
13 fdisp(fid,'Nonlinear System Newton After Iterations =
       3');
14 fdisp(fid,'Nonlinear System Newton After Tolerance =
       1.0e-3');
15 fdisp(fid,'Nonlinear System Relaxation Factor = 1');
16 fdisp(fid,'Linear System Solver = Iterative');
17 fdisp(fid,'Linear System Iterative Method = GCR');
18 fdisp(fid,'Linear System Max Iterations = 500');
19 fdisp(fid,'Linear System Convergence Tolerance = 1.0e
       -10');
20 fdisp(fid,'Linear System Preconditioning = ILU1');
21 fdisp(fid,'Linear System ILUT Tolerance = 1.0e-3');
22 fdisp(fid,'Linear System Abort Not Converged = False'
       );
23 fdisp(fid,'Linear System Residual Output = 1');
24 fdisp(fid,'Linear System Precondition Recompute = 1')
       ;
25 fdisp(fid,'End');
```

This second solver section creates an additional output for the results. While the file for the ELMER post-processor (*.ep file) already is a result output, it contains every single value at every node. This is far more information than needed and exceedingly difficult to process without error. This section on the other hand produces a much simpler XML formatted plain-text file. 'Exec Solver' sets this solver to only execute at the given time 'after timestep'. Since the simulation is steady state with only one iteration. This means results will be written after the simulation terminates. 'Equation' and 'Procedure' select the needed methods (from a pre-existing set implemented in ELMER, though it is possible to write additional solvers). 'Output File Name' set the name of the file, while 'Output Format' sets the file type. The variable type of the 'Output Format' is specified to be a String. The file type is set to 'vtu' (one of the supported plain-text types). To ensure plain-text results, 'Binary Output' must be set to 'False'. 'Single Precision' sets the precision of the numbers in the file.

Code A.18: Code to write an additional custom result output in a *.sif file with *writeSif.m*

```
1 fdisp(fid,'Solver 2');
2 fdisp(fid,'Exec Solver = after timestep');
3 fdisp(fid,'Equation = "result output"');
4 fdisp(fid,'Procedure = "ResultOutputSolve"
      "ResultOutputSolver"');
5 fdisp(fid,['Output File Name = "' filename '"']);
```

```
6  fdisp(fid,'Output Format = String "vtu"');
7  fdisp(fid,'Binary Output = False');
8  fdisp(fid,'Single Precision = True');
9  fdisp(fid,'End');
```

The body section already specified that the first equation section is to be used. Here, this section gets defined (showing that there is not necessarily a specific order for the sections apart from the header). The 'Name' is merely a convenience. Setting 'Calculate Stresses' to 'True' however is very important as those are the key informations sought by the advanced genetic algorithm. This activates the calculation of all stresses components. 'Active Solvers' is an array associating solver sections to the equation.

Code A.19: Code to write an equation section of a *.sif file with *writeSif.m*

```
1  fdisp(fid,'Equation 1');
2  fdisp(fid,'Name = "Elasticity"');
3  fdisp(fid,'Calculate Stresses = True');
4  fdisp(fid,'Active Solvers(1) = 1');
5  fdisp(fid,'End');
```

The material section adds physical properties necessary for the calculations to the bodies referring to it. The example shown gives values for generic stainless steel and is one of the choices available by default when using ElmerGui. The 'Name' is once again merely a convenience. The properties 'Density', 'Poisson ratio', and 'Youngs modulus' are all given in standard units.

Code A.20: Code to write a material section of a *.sif file with *writeSif.m*

```
1  fdisp(fid,'Material 1');
2  fdisp(fid,'Name = "Steel (stainless - generic)"');
3  fdisp(fid,'Density = 7925.0');
4  fdisp(fid,'Poisson ratio = 0.285');
5  fdisp(fid,'Youngs modulus = 200.0e9');
6  fdisp(fid,'End');
```

The Body Force sections define general forces that act on the entire body. Gravity is a good example. As before, the name is a convenience. The key information is the 'Stress Bodyforce 2'. The number '2' already defines that the following force (or one of its components) acts along the height axis of the body (as defined by the coordinate mapping). The value '$-9.81*7925.0' warrants a closer look. It begins not with a number or even a mathematical symbol but with a dollar-sign. This sign flags the following information for interpretation by MATC (an internal mathematical language used by ELMER). This is necessary as the rest of the line is actually a multiplication, albeit a very simple one. The negative sign determines the orientation along

the axis. The product multiplies the standard gravity (rounded up at the second decimal place) with the density of the material (stainless steel - generic). Both values are given in standard units. The result gives the force per volume for the body. This being the format expected by the variable is actually not mentioned in the manual. It is found out by scrutinising the instructions of tutorials.

Code A.21: Code to write a body force section of a *.sif file with *writeSif.m*

```
1 fdisp(fid,'Body Force 1')
2 fdisp(fid,'Name = "Gravity"');
3 fdisp(fid,'Stress Bodyforce 2 = $ -9.81*7925.0');
4 fdisp(fid,'End');
```

The boundary condition sections determine how the boundaries of bodies are affected. The first of two boundary conditions presented is the fixation condition. It is used to set immovable surfaces such as those by which a beam is affixed to a wall or a warehouse shelf. The 'Target Boundaries' is an array of size two, filled with the boundaries '1' and '2'. As the first surfaces created in the geometry file are the left and right profile surface, those will be targeted. There are three 'Displacement' variables, one for each axis direction. All are set to 0.0, thereby preventing any displacement from occurring for the affected surfaces, thus representing the fixation.

Code A.22: Code to write a boundary condition section for fixation in a *.sif file with *writeSif.m*

```
1 fdisp(fid,'Boundary Condition 1');
2 fdisp(fid,'Target Boundaries(2) = 1 2');
3 fdisp(fid,'Name = "Wall"');
4 fdisp(fid,'Displacement 3 = 0.0');
5 fdisp(fid,'Displacement 2 = 0.0');
6 fdisp(fid,'Displacement 1 = 0.0');
7 fdisp(fid,'End');
```

The second boundary condition is used to define a force applied to the model. As boundary conditions can not affect bodies, only surfaces, the 'Target Boundaries' variable is set to only '3' (the outer surface of the profile). Only 'Force 2' is set, as the load applied is assumed to only affect the beam in one orientation. The value is set by the input variable *force*. Merely the orientation is modified by prepending it with a negative algebraic sign

Code A.23: Code to write a boundary condition applying a force to the body in a *.sif file with *writeSif.m*

```
1 fdisp(fid,'Boundary Condition 2');
2 fdisp(fid,'Target Boundaries(1) = 3');
3 fdisp(fid,'Name = "Mass"');
```

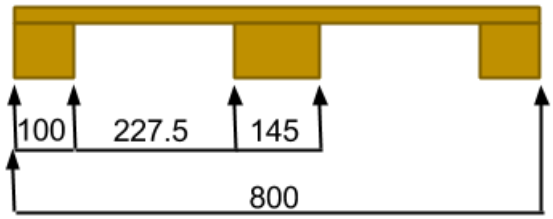

Figure A.4: Schematic of the front view of a standard EUR-pallet (or EPAL-pallet) with the width specifications in millimetres.

```
4 fdisp(fid,['Force 2 = ' num2str(force)]);
5 fdisp(fid,'End');
```

Originally, to simulate a realistic situation, the force was to only be applied to the contact surface of EUR-paletts resting on the beam. An algorithm was developed that would divid the length of the beam into equal segments with a minimum length of 0.8m (the width of a standard EUR-pallet () filling the beam with the maximum amounts of pallets without overlaps. The remaining space between the pallets would be determined by dividing the length of the beam by the number of pallets, subtracting one pallet-width and halving the result. The result was a space added to both sides of the pallets position (two pallets placed next to each other will accordingly be separated by twice this space). A conditional statement would evaluate as **TRUE** for all coordinates along the length of the beam covered by a pallet and set the force to an input value. All other coordinates would receive a force of 0 as their boundary condition.
Despite being fully implemented, this approach was removed to reduce the complexity of the FEM-system and thereby possible sources of errors.

A.2.2 Reading results

Once ElmerSolver ran with the command file generated by *writeSif*, the results must be imported into Matlab. While going over the sections of the command file, two output files were encountered: a *.ep and a *.vtu file with input argument dependent file names. The first one is the file for the post processor and created by default. The second one is specifically requested and contains the data in a XML-like format, making it much easier to read, and parse. This is performed by the function *importResults*. The function requires a string containing the file name variable used to create several files over various functions (usually differentiated by appending various character sequences or file type endings) and returns a cell array containing all values of all stress components available as well as the absolute displacement of all points.

The function starts out by loading the result file by not only appending the file type to the name but also preceding this by the number sequence '0001' . This sequence is automatically added by ElmerSolver. The file is loaded as a single one dimensional character matrix, which allows the content to be manipulated like any other matrix. With the Matlab function *strfind* indices within a matrix matching a given character sequence can be found (the position matching the first sought character is given if the entire sequence matches). This is used on the lines 3 and 4 to find a match for '<PointData>' (saved to *pointDataS*) and one for '</PointData>' (saved to *pointDataE*). These sequences delimit the section of the file containing the data concerning the individual points of the simulation. The variable of the result file *fid* is then overwritten with the sequence defined by the two previously determined variables. This excerpt of the file is parsed for matches to the sequences '<DataArray' and '</DataArray>' which are saved to *dataStart* and *dataEnd* respectively. This must be done after reducing the result to the point data section as the full file would return many matches for the data array subsections that are irrelevant to this genetic algorithm. Each of these data arrays contains one of the stress or the displacement components. The for-loop from line 8 to 40 therefore loops through the *dataStart* matrix containing the starting point of each array. Within the loop, the subsection is copied to a separate variable. This is parsed for the name attribute and the returned index is increased by 6 to skip the string 'Name="' and adjust it to the beginning of the attributes value. The end of the attribute value is found by searching for the first occurrence of a quotation mark beyond the beginning index. With this information the name of the variable stored in the data array can be extracted and saved to the variable *idName*. Next the actual data is separated from the opening 'DataArray' tag in a separate variable *dataProper*. With the Matlab function *strsplit* the individual values in the string are stored in separate matrix positions. This is done by splitting the string every time the separator character (or character sequence) is encountered. For the data in the *.vtu file the separator is a space character. The values are then converted to numbers (of format double) with the *str2double* function of Matlab. To avoid problems with calculations all matrix cells with content that is non a number (as determined by the Matlab function *isnan*) is deleted. With the help of a switch statement two cases are distinguished. If the name of the data array is 'displacement' the data requires a bit more processing. As the displacement values for each point are given as the components in each orientation of the coordinate system, an absolute displacement for each point must be calculated. To make things easier, then one dimensional matrix containing the data is reshaped (the *reshape* function is given by Matlab) into a two dimensional matrix of three columns (as there are three axes) and a number of rows equal to a third of the length of the one dimensional matrix. The absolute displacement at each point is then calculated by determining the square root of the sum of

the squares of each component. The last line for this case (line 22) contains two nested Matlab functions *eval* and *sprintf*. The first one evaluates the string given as input argument as if it was entered via the command-line of Matlab and executed. The second one simply formats the string it contains. This construct enables the dynamical assignment of variable name. In this example, a variable with the name of the data array is created and assigned the values of all absolute displacements calculated from the data. Now, returning to the switch case, there is a second case introduced by the keyword otherwise (meaning all cases in which the *idName* is not 'displacement' will be handled). It contains another switch translating numbered 'stress_' strings of the *idName* into strings named after the stress component they represent. This is done adapt the naming of the stress components of the ELMER version tested on the linux based servers to the naming convention encountered in the windows based version. On windows systems this switch will have no effect. Like before, variables with the data arrays names are created and given the values of the data they each contain. On the last line (line 41) all variables expected to be created by the function are saved into the cell array that will be returned. The second switch could have been omitted, instead appending the value of each passage of the for-loop to the cell array, but this implementation is slightly more comprehensible.

Code A.24: Code to import the ElmerSolver results from a *.vtu file

```
1  function a=importResults(filename)
2   fid = fileread(['mesh-' filename '/' filename '0001.
       vtu']);
3   pointDataS = strfind(fid,'<PointData>');
4   pointDataE = strfind(fid,'</PointData>');
5   fid=fid(pointDataS:pointDataE);
6   dataStart = strfind(fid,'<DataArray');
7   dataEnd = strfind(fid,'</DataArray>');
8   for i=1:length(dataStart)
9    dataPart=fid(dataStart(i):dataEnd(i));
10   idStart=strfind(dataPart,'Name="')+6;
11   idEnd=find(dataPart(idStart:end)=='"',1,'first')+
       idStart-2;
12   idName=dataPart(idStart:idEnd)';
13   dataProper=dataPart(find(dataPart=='>',1,'first'):
       end);
14   val=strsplit(dataProper',' ');
15   val=str2double(val);
16   val(isnan(val(:)))=[];
17   switch(idName)
18    case 'displacement'
19     points=length(val)/3;
```

```
20      val=reshape(val,3,points)';
21      displacement_abs=sqrt(val(:,1).^2+val(:,2).^2+val
            (:,3).^2);
22      eval(sprintf([idName '=max(displacement_abs);']));
23     otherwise
24      switch(idName)
25       case 'stress_1'
26        idName='stress_xx';
27       case 'stress_2'
28        idName='stress_yy';
29       case 'stress_3'
30        idName='stress_zz';
31       case 'stress_4'
32        idName='stress_xy';
33       case 'stress_5'
34        idName='stress_yz';
35       case 'stress_6'
36        idName='stress_xz';
37      end
38      eval(sprintf([idName '=max(val);']));
39    end
40  end
41  a={stress_xx,stress_yy,stress_zz,stress_xy,stress_yz,
        stress_xz,vonmises,displacement};
```

A.3 Parallelization of the fitness calculation

Understanding all the key functions needed for the parallelized calculation of the fitness values in the advanced genetic algorithm, the connecting functions can be analysed. As the genetic algorithm itself has a negligible runtime compared to the generation of the meshes and the FES (even at low complexity and low populations of only 10 individuals it makes up less then 0.5% of the total runtime of a generation) it was decided to limit the parallelization to the fitness calculations. This however required to source out the simple function call to *fitness* out of *setupGA* and *evolveGA* to a separate function solely dedicated to managing the function calls made to different servers on a cluster.

Code A.25: Code of the *getFitness* function, enabling the parallel calculation of fitness values on server clusters

```
1  function fit=getFitness(gen,genome,filename)
2   global profile_type
3   if nargin<3
```

```
4    filename='defaultfile';
5   end
6   individuals=size(genome,1);
7   nodeList=getNode(1);
8   nodeCount=length(nodeList);
9   nodeWork=zeros(nodeCount,1);
10  for i=1:individuals
11   while all(nodeWork~=0)
12    pause(0.5);
13    for j=1:nodeCount
14     if exist(['fit_check_' sprintf('%d.mat',nodeWork(j
           ))],'file')==2
15      nodeWork(j)=0;
16     end
17    end
18   end
19   nodeIndex=find(nodeWork==0,1,'first');
20   save(sprintf('%d.mat',i),'open','genome','gen','
         filename');
21   system(['ssh mmt@' nodeList{nodeIndex} ' octave --
         eval "fitness\(' sprintf('%d',i) '\)" &']);
22   nodeWork(nodeIndex)=i;
23  end
24  while any(nodeWork~=0)
25   for j=1:nodeCount
26    if nodeWork(j)~=0
27     if exist(['fit_check_' sprintf('%d.mat',nodeWork(j
           ))],'file')==2
28      nodeWork(j)=0;
29     end
30    end
31   end
32  end
33  fit=zeros(1,size(genome,1));
34  for i=1:individuals
35   load(['fit_' sprintf('%d.mat',i)]);
36   fit(i)=fitValue;
37  end
38  cleaning(individuals);
```

The function *getFitness* is essentially such a queue-management system. It requires at least two input arguments: *gen* (the number of the current generation) and *genome* (a matrix containing the genomes of all individuals in a generation). The third variable *filename* is optional and will be set

to 'defaultfile' if not given. Before the calls to the *fitness* function can be made, four more variables have to be declared. The *individuals* variable holds the number of individuals in the *genome*. The cell array *nodeList* contains strings with the names of the servers connected to the main server that can be used for fitness calculations. The list is generated by the function *getNode*. *nodeCount* stores the total number of available nodes in *nodeList*. Lastly, *nodeWork* is a one dimensional zero matrix of the same length as the *nodeList*. This is used to store the state of each available server node. The value zero is used to flag a node as free. If a node is busy, the index of the individual currently being calculated is written to the position in *nodeWork* associated with the respective server node.
A for-loop goes through all individuals, making a *fitness* call for each. If a server node is free, its index is determined in *nodeIndex* (line 19). A *.mat file containing the global scope variable *profile_type*, *genome*, *gen*, and the *filename* is saved with the current individual count i as name. The code on line 21 makes the *fitness* function call as a system call. Since Matlab (as well as octave) is a asynchronous software, meaning it will wait for a command or called function to finish before executing the next line of code, it can not normally perform multiple functions at once, or synchronously. To circumvent this problem, the *fitness* are not made within the same program instance running the core algorithm. Instead, via a system call, a new program instance is started on one of the connected nodes and given a function as starting parameter that can load the previously saved file to get access to all the data required to perform the fitness calculation. The system call is performed via the ssh protocol and send to the user 'mmt' (a common user account spanning all servers of the cluster). After the @-sign the server name, taken from the *nodeList* at the *nodeIndex* position, is appended. The rest of the string gives the command that is executed on the targeted server. Octave is started with the parameter '–eval'. This causes a following statement in between quotation marks to be executed as a command by the program once it has started. The function *fitness* is called with a single parameter: the current individual count (while the name is identical to a function presented for the core algorithm, this is a modified version which will be discussed in a later subsection). To this system call the sign & is added. In linux terminal shells, a trailing & is interpreted as a control operator to perform a command in the background. Since the shell immediately returns the status 0 (signifying a successful command execution), the *getFitness* will also continue its execution right away, thus allowing synchronous work over multiple nodes. The current individual count is added to the *nodeIndex* position of *nodeWork*, thereby indicating that the server node is busy calculating the fitness of the specified individual. The code inside the loop described is however preceded by a while loop whose body is executed whenever none of the *nodeWork* is 0, meaning available. The body first pauses the entire algorithm for 0.5 seconds. Then a for-loop it-

erates through *nodeWork* and checks if the result file, generated when the fitness calculation is done, associated with the individual exists. This can be done with the aptly named Matlab function *exist*. If it does, the position in *nodeWork* is set to 0 and so the server node is marked as being available once again.

Once all individuals have been assigned to a server node for their fitness calculation, a second while-loop is executed, that is almost identical to the one inside the previous for-loop. The differences being, that there is no pause and that this loop is not executed as long as no server node is available but until all are available. This prevents the rest of the function from being executed as long as not all results are available, thus preventing errors due to accessing unavailable files. Once all results are in, a one dimensional zero matrix *fit* of length equal to the number of individuals is created. A for-loop loads each result file from the fitness calculation and saves the content of a *fitValue* to the matrix. As this variable is contained in the result files and always has the same name, it is overwritten each time a new file is loaded. Before the *fit* matrix is returned and so the function terminated, a small helping function *cleaning* is executed.

A.3.1 Cleaning up

The function *cleaning* (code A.26)is quite simple. It takes a number as its single input argument. Iterating for that specific number of times, it deletes files based on three patterns. This is done as the existence of those files is being used by other functions as a checking mechanism before performing actions that are depending on their existence. However, since the naming patterns only include one number which is referring to an individual in a generation, they cannot be discerned in between generations. Deleting them after they have served their purpose creates a clean slate so to speak for the next generation.

Code A.26: Code of the *cleaning* function, deleting some files

```
1 function cleaning(a)
2  for i=1:a
3   delete([num2str(i) '.mat']);
4   delete(['fit_' num2str(i) '.mat']);
5   delete(['fit_check_' num2str(i) '.mat']);
6  end
```

A.3.2 Listing available server nodes

The names of all available server nodes are stored in a text file with a predefined layout to make automated parsing possible. It starts with a first line describing to potential users the pattern according to which servers must

be entered to be readable. This line starts with a single percent sign and ends with a sequence of three percent signs, thus marking it as a comment to be ignored by the function itself. The function *getNode* reads the content of the file into a variable *fid*. Only the content from the triple percent signs onward is kept. Next all occurrences of three signs are recorded. *starts* contains all indices of the sign '>' (greater than: marking the start of a new server definition), *mids* stores all instances of the sign '/' (slash: separating the server names from the number of CPU cores available on it) and *ends* saves the positions of ';' (semi-colon: ending a server definition). The return variable *s* is created as an empty cell array (as strings of potentially varying length have to be saved). An variable *cores* is created as all-ones matrix of the same length as *starts* (used as indicators as to the number of servers). A first for-loop iterates through the length of *starts*, parses the number of cores on each server and saves them, in order, to *cores* (after converting them to numbers). A second for-loop iterates through the server names. A nested for-loop iterates through the respective servers core amount. The name of the server is added once for each CPU core available on it. The resulting cell arrays length is equivalent to the sum of all cores available over all servers. This is then returned.

Code A.27: Code of the *getNode* function, parsing the nodelist.txt file for server names and CPU core counts

```
1  function s=getNode(g)
2   fid=fileread('nodelist.txt');
3   fid=fid(strfind(fid,'%%%'):end);
4   starts=strfind(fid,'>');
5   mids=strfind(fid,'/');
6   ends=strfind(fid,';');
7   s={};
8   cores=ones(length(starts),1);
9   for i=1:length(starts)
10    cores(i)=str2double(fid(mids(i)+1:(ends(i)-1))');
11  end
12  for i=1:length(cores)
13    for j=1:cores(i)
14      s{end+1}=fid(starts(i)+1:(mids(i)-1))';
15    end
16  end
```

A.3.3 The *fitness*-function per se

The advanced genetic algorithm, as it no longer bases the fitness on the second moment of area but on finite element simulations, requires a new *fitness* function. This new function is not only adapted to the new method

of acquiring values needed for the fitness calculation, but also accounts for the new function call by *getFitness* and the fact, that it is run in a separate instance of Octave.

Code A.28: Code of the *fitness* function of the advanced genetic algorithm

```
1  function fitness(g)
2   global profile_type
3   source=sprintf('%d.mat',g);
4   load(source);
5   ind=shiftdim(genome(g,:,:));
6   long=3;
7   force=94000
8   filename=[filename '_' num2str(gen) '_' num2str(g)];
9   try
10   [~,l]=schwerpunkt(ind);
11  catch
12   [dismiss,l]=schwerpunkt(ind);
13  end
14  tolerance=max(1.15,2-(gen/500));
15  sXX_co=135300*tolerance;
16  sYY_co=204100*tolerance;
17  sZZ_co=296000*tolerance;
18  sXY_co=72410*tolerance;
19  sYZ_co=105000*tolerance;
20  sXZ_co=211700*tolerance;
21  v_co=403400*tolerance;
22  d_co=0.002;
23  a=solveStructure(ind,long,force,filename);
24  fitValue=(1/l)*sigmoid(sXX_co-max(a{1}))*sigmoid(
        sYY_co-max(a{2}))*sigmoid(sZZ_co-max(a{3}))*
        sigmoid(sXY_co-max(a{4}))*sigmoid(sYZ_co-max(a{5})
        )*sigmoid(sXZ_co-max(a{6}))*sigmoid(d_co-max(a{8})
        );
25  save(['fit_' sprintf('%d.mat',g)],'fitValue');
26  save(['fit_check_' sprintf('%d.mat',g)]);
```

This new fitness function (see code A.28) only receives a single input argument *g*, a single number that represents the individual from the population for which the fitness value is to be calculated. Based on this input, a Matlab file (*.mat) is loaded (line 4) containing all the variables needed to perform the functions roll in an independent instance. Since the variable *profile_type* is set to the global scope on line 2, the variable of the same name loaded from the file will be in the global scope in this Octave instance. Other loaded variables are *genome* (a matrix containing the genome of all

individuals of the population), *gen* (a scalar indicating the current generation of the genetic algorithm) and *filename* (a string with the base name for several of the files that are created over the course of the fitness calculation). The file-name is modified before its first use (line 8) to include a reference to the current generation and the individual count, thus allowing to identify and analyse related files, even long after the algorithm terminates. Next, two key variables are defined. *long* determines the length of the beam in the model in metres. The value 3 was arbitrarily chosen during the development and testing phase. *force* determines the force that is loaded onto the beam in newton per square meter. This is a crucial information. Even though the variable in ELMER (even in the GUI) is simply labelled as force, it actually is a distributed force (force per surface) that is expected (note that none of the manuals of ELMER mention the format expected for the force and the information given here is based on a post on the official ELMER forum by a site administrator going by the pseudonym of 'raback'). Input values should account for this and for the limited contact surface (as explained while discussing the second boundary condition section in chapter A.2.1). The value 94000 was set as per request from a colleague from Linz. The circumference is calculated in *l* by the *schwerpunkt* function. The center of gravity is not necessary and simply saved to the default answer variable *ans* (usually the sign would be used, but this led to issues with older Octave versions not having implemented it). The variables *sXX_co*, *sYY_co*, *sZZ_co*, *sXY_co*, *sYZ_co*, *sXZ_co*, *v_co*, and *d_co* are all maximum values of the various stress components and the displacement calculated for a reference closed profile of a square with side length of twenty centimetres (length of the middle line profile) loaded with the arbitrary distributed force. Only the displacement limit *d_co* was set as per request from a colleague from Linz.
To get the stress components and displacement values for the current individual the function *solveStructure* is called and the results are returned to the cell array *a*. The fitness itself is the inverse of the circumference of the profile (technically of its middle line) multiplied by the results of the *sigmoid* functions (the function was covered in chapter 4.6.3 and its purpose in chapter 4.6) of the difference between the limit values of the individual stress components and their maximum value, as well as between the displacement limit and the maximum displacement. The results are saved to a variable *fitValue*, which in turn is saved to a Matlab file named in a pattern 'fit_' followed by the number of the individual calculated by the function. A second empty Matlab file is created afterwards with the naming pattern 'fit_check_' followed by the individual number. Since the first file is created and subsequently returns a positive result when checked by the Matlab function *exist* before the variable is stored in it, errors can occur if its existence is the sole condition checked before attempting to read it. Taking advantage of the asynchronous nature of Matlab, a second file is created right after the first one is saved, thereby guaranteeing, that by the time the second one can

be considered as existent, the first one is definitely accessible.

A.3.4 Solving the structure for the *fitness*-function

Various functions involved in the generation the stress components and the displacement have already been covered (e.g. *beam3msh*, *writeSif* or *importResults*). But all of these have to be called in the proper order. This is the purpose of *solveStructure* (see code A.29).

Code A.29: Code of the *solveStructure* function of the advanced genetic algorithm

```
1  function a=solveStructure(c,long,force,filename)
2   [o,i]=beamPoints(c);
3   if ~isIntersecting(o,i)&&~isIntersecting(o)&&~
       isIntersecting(i)
4    beam3msh(i,o,long,filename);
5    counter=0;
6    while counter<10
7     if length(fileread([filename '.msh']))<10^6
8      beam3msh(i,o,long,filename);
9     else
10     break;
11    end
12    counter=counter+1;
13   end
14   if length(fileread([filename '.msh']))<10^3
15    a={Inf,Inf,Inf,Inf,Inf,Inf,Inf,Inf};
16   else
17    writeSif(filename,o,long,force);
18    system(['ElmerGrid 14 2 ' filename ' -out ' pwd() '
         /mesh-' filename]);
19    system(['ElmerSolver ' filename '.sif' ]);
20    a=importResults(filename);
21   end
22   else
23    a={Inf,Inf,Inf,Inf,Inf,Inf,Inf,Inf};
24   end
```

This function has four mandatory input arguments: *c* (the genome of an individual), *long* (the length of the beam in metres), *force* (the distributed force in newton per square metres), and *filename* (a string with the file name). First the coordinates of the inner and outer wall of the profile are calculated by *beamPoints*. To prevent errors during the mesh creation, the rest of code is only executed if neither of the wall coordinates result in

intersections. Otherwise the return variable *a* is created as cell array of the expected size filled with zeros. If there is no intersection, the mesh is created by *beam3msh*. Next the command file for ELMER is created by *writeSif*. Before the ElmerSolver modul can be started however, the mesh has to be translated. While ELMER can load *.msh files (the file type created by gmsh), it cannot perform simulations with them. The model must first be converted from this single file to four files respectively named: 'mesh.boundary', 'mesh.elements', 'mesh.header', and 'mesh.nodes'. This is done by calling the modul ElmerGrid with a few parameters via system call. The first number defines the format of the input file (14 stands for *.msh files format as used by gmsh). The second number defines the output format (2 being the ElmerSolver format). These are followed by the name of the input file. By adding '-out' an output folder can be defined. If it is not given, the files are created in a default folder (on Windows operating systems for example, they are created directly on 'c:/'). This output folder is set to be a sub-folder of the current directory. The Matlab function *pwd* returns the current folder. To this the string '/mesh-' is added along with the file name. Thus creating a new folder, named in the pattern 'mesh-' plus the file name. The *.msh files are still created with gmsh since *.geo files are not supported by ElmerGrid or ElmerSolver. After all necessary files are created, ElmerSolver can finally be started with a system call only adding the name of the command file as a parameter. Finally the results a imported into Octave with *importResults* and directly saved to the return variable *a*.

A.3.5 Modifications to accommodate the new *fitness* function calls

Since for the advanced genetic algorithm the *getFitness* function handles the fitness calculations of all individuals, some adaptations are needed to a few other functions. The main change to be made is to the function call of *fitness* in *setupGA* and *evolveGA*. As this is now performed by the function *getFitness*, the *fitness* function iterating for-loops have to be replaced. A single call to the *getFitness*, with the current generation count (0 during *setupGA*) and all genomes as input arguments, suffices to replace the loops. This, however, is only efficient if all fitness calculations are performed at the same time. In the basic algorithm for example the *crossover* function compares the fitness values of the offspring before returning the fittest individual. This had to be changed. In the advanced genetic algorithm, the *crossover* function returns both offspring without calculating and comparing their fitness. The matrix receiving the results of this function in *evolveGA* have an increased size to account for this change.

The *setupGA* function contains an additional change. As the advanced genetic algorithm is more expensive in terms of computational resources

(requiring a server cluster and more computing time). To avoid runs without any useful results due to bad fitness throughout the initial population, the creation of individuals is handled by two loops. The first while-loop is only executed as long as there are no individuals with a fitness above 0 (individuals with a fitness greater than 0 are called fit) forming a base population. The body of this loop largely resembles the base form of *setupGA* with some counters added to determine whether there are fit individuals in the population. The second while-loop, which executes until the number of fit individuals in the population meets the number requested by the input argument *individuals*, additionally uses the mutation operator with the fit individuals created so far in an attempt to speed up the creation process. If the fitness conditions are very stringent the creation of fit individuals by the standard method of calling the *genotype* function might yield a low success rate. The mutation operator on the other hand is can be much more successful. While the fifth rule tries to set the success-rate of mutation to 0.2, it defines as success only those mutations where the offspring has a higher fitness value than the parent. During the creation of individuals however any positive fitness can be considered a success. And by those standards, the *mutation* function is quite successful.

This concludes the modifications necessary to accommodate the new handling of fitness calculations for the advanced genetic algorithm.

Bibliography

[1] The mutagenic effect of benzene, toluene and xylene studied by the sce technique. *Mutation Research/Genetic Toxicology*, 58(2-3):313 – 316, 1978.

[2] BN Ames. Dietary carcinogens and anticarcinogens: Oxygen radicals and degenerative diseases. *Science*, 221(4617):1256–1264, 1983.

[3] Ruth Bollongino, Joachim Burger, Adam Powell, Marjan Mashkour, Jean-Denis Vigne, and Mark G. Thomas. Modern taurine cattle descended from small number of near-eastern founders. *Molecular Biology and Evolution*, 29(9):2101–2104, 2012.

[4] S. Chiraphadhanakul, P. Dangprasert, and V. Avatchanakorn. Genetic algorithms in forecasting commercial banks deposit. In *Intelligent Processing Systems, 1997. ICIPS '97. 1997 IEEE International Conference on*, volume 1, pages 116–121 vol.1, 1997.

[5] Guy Cowlishaw and Robin IM Dunbar. Dominance rank and mating success in male primates. *Animal Behaviour*, 41(6):1045–1056, 1991.

[6] Kenneth Alan De Jong. Analysis of the behavior of a class of genetic adaptive systems. 1975.

[7] John W Eaton, David Bateman, and Soren Hauberg. *GNU Octave Manual Version 3*. Network Theory Ltd., 2008.

[8] Reza Ebrahimzadeh and Mahdi Jampour. Chaotic genetic algorithm based on lorenz chaotic system for optimization problems. *International Journal of Intelligent Systems and Applications (IJISA)*, 5(5):19, 2013.

[9] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*. springer, 2003.

[10] C Ericson and I Ordonez-Reinoso. Dialogue on uniform crossover, 1991.

[11] Larry J Eshelman and J David Schaffer. Gas and very fast simulated re-annealing. *GA-Digest (December, 199 1) v5n37*, 1991.

[12] Christophe Geuzaine and Jean-François Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

[13] David E Goldberg. E.(1989). genetic algorithms in search, optimization and machine learning. *Reading: Addison-Wesley*, 1990.

[14] D.E. Goldberg. Alleles, loci, and the tsp. In *Proceedings of the First International Conference on Genetic Algorithms*, pages 154–159. Lawrence Erlbaum Associates, 1985.

[15] Erick Greene, Bruce E Lyon, Vincent R Muehter, Laurene Ratcliffe, Steven J Oliver, and Peter T Boag. Disruptive sexual selection for plumage coloration in a passerine bird. *Nature*, 407(6807):1000–1003, 2000.

[16] D. Gross, W. Hauger, and W. Schnell. *Technische Mechanik: Band 2: Elastostatik*. Springer, 2009.

[17] Daniel L Hartl et al. *A primer of population genetics.* Sinauer Associates, Inc., 1988.

[18] JH Holland. Adaption in natural and artificial systems. *Ann Arbor MI: The University of Michigan Press*, 1975.

[19] J.H. Holland. Genetic algorithms and classifier systems: foundations and future directions. In J.J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pages 82–89. Lawrence Erlbaum Associates, 1987.

[20] Robert R Jackson. Courtship versatility in the jumping spider, phidippus johnsoni (araneae: Salticidae). *Animal behaviour*, 25:953–957, 1977.

[21] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.

[22] R. Caruna L.J. Eshelman and J.D. Schaffer. Biases in the crossover landscape. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 10–19. Morgan Kaufmann, 1989.

[23] S.J. Louis and G.J.E. Rawlins. Designer genetic algorithms: Genetic algorithms in structure design. In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 53–60. Morgan Kaufmann, 1991.

[24] MATLAB. *version 8.0.0.783 64-bit (R2012b).* The MathWorks Inc., Natick, Massachusetts, 2012.

[25] Markus Metz, Nicole Geberzahn, Lars H Hansen, Georg M Klump, and Thomas WP Friedl. Effects of behavioural time budgets and nest-building efficiency on male reproductive performance in red bishops (euplectes orix). *Journal of Ornithology*, 148(2):145–155, 2007.

[26] Zbigniew Michalewicz. *Genetic algorithms+ data structures= evolution programs*. springer, 1996.

[27] Jeffrey H. Miller. Mutagenic specificity of ultraviolet light. *Journal of Molecular Biology*, 182(1):45 – 65, 1985.

[28] Vincent Mirabet, Pradeep Das, Arezki Boudaoud, and Olivier Hamant. The role of mechanical forces in plant morphogenesis. *Annual Review of Plant Biology*, 62:365–85, 2011.

[29] Melanie Mitchell. An introduction to genetic algorithms, 1996. *PHI Pvt. Ltd., New Delhi*, 1996.

[30] Camilo Mora, Derek P Tittensor, Sina Adl, Alastair GB Simpson, and Boris Worm. How many species are there on earth and in the ocean? *PLoS biology*, 9(8):e1001127, 2011.

[31] H Allen Orr. The genetic theory of adaptation: a brief history. *Nature Reviews Genetics*, 6(2):119–127, 2005.

[32] Taejin Park, Ri Choe, and Kwang Ryel Ryu. Adjusting population distance for the dual-population genetic algorithm. In *AI 2007: Advances in Artificial Intelligence*, pages 171–180. Springer, 2007.

[33] Antti Pursula, Mikko Lyly, Esko Järvinen, Thomas Zwinger, and Juha Ruokolainen. Multiphysical simulations with elmer finite element software in combination with gid.

[34] Gregory JE Rawlins. *Foundations of genetic algorithms*. Morgan Kaufmann Publishers Inc., 1992.

[35] Ingo Rechenberg. Evolutionsstrategie–optimierung technisher systeme nach prinzipien der biologischen evolution. 1973.

[36] FJ Richards. A flexible growth function for empirical use. *Journal of experimental Botany*, 10(2):290–301, 1959.

[37] Marc H. J. Romanycia and Francis Jeffry Pelletier. What is a heuristic? *Computational Intelligence*, 1(1):47–58, 1985.

[38] W.L. Russell. *RECENT STUDIES ON THE GENETIC EFFECTS OF RADIATION IN MICE*. Oct 1968.

[39] J.D. Schaffer and A. Morishma. An adaptive crossover distribution mechanism for genetic algorithms. In J.J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pages 36–40. Lawrence Erlbaum Associates, 1987.

[40] Gilbert Syswerda. Uniform crossover in genetic algorithms. 1989.

[41] K.S. Tang, K.F. Man, S. Kwong, and Q. He. Genetic algorithms and their applications. *Signal Processing Magazine, IEEE*, 13(6):22–37, 1996.

[42] Thomas Zwinger. Elmersolver command file. Presentation from Elmer/Ice course 14 and 15 February, 2008, LGGE, Grenoble, France, 2008.