HORVÁTH ANDRÁS LEVENTE
SZAKDOLGOZAT

Budapesti Műszaki és Gazdaságtudományi Egyetem
Gépészmérnöki Kar
Műszaki Mechanikai Tanszék

Budapesti Műszaki és Gazdaságtudományi Egyetem
Gépészmérnöki Kar

Műszaki Mechanikai Tanszék

Manual and automated optimization of a crashbox geometry
SZAKDOLGOZAT

Készítette: Horváth András Levente

Konzulens:
Tóth Péter
Fejlesztőmérnök

Témavezető:
Dr. Kossa Attila
Egyetemi docens

Budapest, 2017

Budapesti Műszaki és Gazdaságtudományi Egyetem
Gépészmérnöki Kar
Műszaki Mechanikai Tanszék
1111 Budapest, Műegyetem rkp. 5. MM épület
www.mm.bme.hu

SZAKDOLGOZAT FELADATKIÍRÁS (BSc)

AZONOSÍTÁS

Név: **Horváth András Levente**			Azonosító: BHI0A4
Képzéskód:	**2N-AM0**	Specializáció kódja:	Feladatkiírás azonosítója:
Képzésnév:	**Mechatronikai mérnök alapszak (BSc)**	**2N-AM0-GM-2010**	**SDGM-MM-17 / BHI0A4**
Szakdolgozatot kiadó tanszék:	**Műszaki Mechanikai Tanszék**	Záróvizsgát szervező tanszék:	**Műszaki Mechanikai Tanszék**
Témavezető:	**Dr. Kossa Attila, egyetemi docens (kossa@mm.bme.hu, Tel.: +36 1 463-1368)**		

FELADAT

Cím:

SZEMÉLYGÉPJÁRMŰ ÜTKÖZÉSI ENERGIA ELNYELŐ ELEM (CRASHBOX) GEOMETRIA KÉZI ÉS AUTOMATIZÁLT OPTIMALIZÁLÁSA

Manual and automated optimisation of Crashbox geometry

Részletes feladatok:

1. Irodalomkutatás.
2. CAD geometria hálózása, peremfeltételek definiálása. A szimulációhoz szükséges fájlok elkészítése.
3. Kézi optimalizálás a háló manuális módosításával.
4. Automatizál optimalizálás előkészítése, Python script írása. Összekötés az optimalizáló programmal.
5. Próbafuttatások, legfontosabb geometriai méretek megkeresése.
6. Futtatások a kiválasztott méretek módosításával.
7. Összehasonlító értékelés több szempont alapján: kézi és automatizált módszer.

Hely:

A szakdolgozat készítés helye:
Vállalat/üzem megnevezése: EDAG Hungary Kft.
Vállalat/üzem címe: 9024, Győr, Zrínyi u. 11.
Konzulens: Tóth Péter, fejlesztőmérnök, (peter.andras.toth@edag.hu)

ZÁRÓVIZSGA

1. záróvizsga tárgycsoport	2. záróvizsga tárgycsoport	3. záróvizsga tárgycsoport
Mechatronika ZVGEMIAM00	**Analóg és digitális technika** ZVEVIAUAM00	**Robotok mechanikája és áramlások numerikus modellezése** ZVEGETOAM02

HITELESÍTÉS

Feladat kiadása: 2017. szeptember 4.		Beadási határidő: 2017. december 8.
Összeállította:	Ellenőrizte:	Jóváhagyta:
	PH.	PH.
……	……	……
témavezető	tanszékvezető/tanszékvezető-h.	dékán/dékánhelyettes

Alulírott, a feladatkiírás átvételével egyúttal kijelentem, hogy a szakdolgozat előkövetelményeit maradéktalanul teljesítettem. Ellenkező esetben tudomásul veszem, hogy a jelen feladatkiírás és a tárgy felvétele érvényét veszti.

Budapest, 2017. szeptember 4.

……
hallgató

Nyilatkozatok

Beadhatósági nyilatkozat

A jelen szakdolgozat az üzem/intézmény által elvárt szakmai színvonalnak mind tartalmilag, mind formailag megfelel, beadható.

Győr, 2017

Az üzem részéről:

üzemi konzulens

Elfogadási nyilatkozat

Ezen szakdolgozat a Budapesti Műszaki és Gazdaságtudományi Egyetem Gépészmérnöki Kara által a Diplomatervezési és Szakdolgozat feladatokra előírt valamennyi tartalmi és formai követelménynek, továbbá a feladatkiírásban előírtaknak maradéktalanul eleget tesz. E szakdolgozatot a nyilvános bírálatra és nyilvános előadásra alkalmasnak tartom.

A beadás időpontja:

témavezető

Nyilatkozat az önálló munkáról

Alulírott Horváth András Levente (BHI0A4) a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatója, büntetőjogi és fegyelmi felelősségem tudatában kijelentem és sajátkezű aláírásommal igazolom, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, és dolgozatomban csak a megadott forrásokat használtam fel. Minden olyan részt, melyet szó szerint vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a hatályos előírásoknak megfelelően, a forrás megadásával megjelöltem.

Budapest, 2017

szigorló hallgató

Összefoglalás

A "crashbox" egy gyakori alkatrész a mai és régebbi személyautókban, valamilyen formában minden modellben megtalálható. Elterjedtsége miatt a crashbox geometriájának optimálása minden új autómodell tervezésének része, így ez egy gyakori feladat az iparban. A crashbox a Crash Management System (CMS) része, ami a jármű több tulajdonságára is kihat, az utasbiztonságtól kezdve a biztosítási költségeken át egy ütközés utáni javítás költségeiig.

A crashbox egy ütközési energia elnyelő elem, párokban alkalmazzák a személyautók első és hátsó felén is. Ez az alkatrész ütközés esetén a jármű mozgási energiáját képlékeny alakváltozással nyeli el. A crashbox elsődleges célja, hogy alacsony sebesség ütközések (max. 15 km/h) során megóvja az autó fő vázszerkezetét a károsodástól, ezáltal jelentősen csökkentve a javítás költségeit. Emiatt a vázszerkezetre átadható erőre egy határértéket írunk elő. További cél a deformációs hossz minimalizálása az egyéb alkatrészek védelmében. Így adódik az ideális viselkedés, melyben az átadott erő az ütközés során a határértéken fut végig. A geometria optimálásának célja minél jobban közelíteni ezt a viselkedést. A dolgozatomban én az első crashbox optimálásával foglalkoztam.

A geometria optimálása történhet manuálisan, ez az elterjedt módszer erre a feladatra. Ennek végeredménye általában egy jó geometria, de a feladatban lévő erős nemlinearitások miatt nem könnyű létrehozni. Továbbá ez egy időigényes és gyakran monoton részekből álló feladat. Mivel a megoldás sok próbálhatásból áll, egy jó megoldás megtalálása sok munkaórát vesz igénybe. Ezt én is elvégeztem egy crashboxon, kb. 80 munkaóra alatt.

A feladatom célja ennek a folyamatnak az automatizálása volt, hogy ezáltal csökkenthető legyen a szükséges emberi időráfordítás. Ezt a módszert egy genetikus optimalizáló programmal és a CAE (Computer Aided Engineering) szoftverek Python scriptből való irányításával valósítottam meg. A program automatikusan elkészíti az új formát az alapforma alapján, lefuttatja az ütközés szimulációját az új geometriával, majd kiértékeli a kapott eredményeket. Ezután kiszámolja, hogy a viselkedése mennyire tér el az ideálistól. Az optimalizáló célja ennek az eltérésnek a csökkentése.

Az automata módszerrel sikerül elérni a kitűzött célt. Az ezáltal készített formák jóval kevesebb, mint 24 óra alatt mutattak olyan jó viselkedést, mint a legjobb kézzel készített geometria. Ha a program elég futásidőt kapott, akkor az általa készített változatok lényegesen pontosabban követték az ideális viselkedést, mint az általam létrehozott crashboxok bármelyike. A módszer hátránya, hogy több szimuláció futtatására van szükség, de mindez automatikusan történik. Az általam készített automatizált módszerrel a szükséges emberi időráfordítás több, mint 95%-al csökkenthető.

Summary

The "crashbox" is a standard part of modern and older passenger cars; it is to be found in every model. As they are so widespread, optimizing the crashbox geometry is part of the design process of any new car model. Thus, this is a common task in the vehicle industry. The crashbox is part of the Crash Management System (CMS), which affect multiple aspects of a car, including passenger safety, insurance cost and the repair cost after a collision.

The crashbox is a part designed to absorb kinetic energy via plastic deformation; it is used in pairs on the front an rear end of the vehicle. The primary goal of the crashbox is to protect the main frame of the car during low speed (maximum 15 km/h) crashes, thus significantly reducing repair costs. For this reason, there is a limit for the force transmitted from the crashbox to the main frame. Additionally, the deformation length is to be minimized to protect additional parts. The ideal crashbox behavior can be defined as the transmitter force staying on the force limit during the crash. The goal of geometry optimization is to follow this behavior as closely as possible. In my thesis, I write about the optimization of the front crashbox.

The optimization can be done manually; this is the most common method used for this task. The result is usually a good geometry. However, they are not easy to produce due to the strong nonlinear nature of the problem. This is also a time-consuming and repetitive process. As this method involves a lot of 'trial and error', finding a good geometry takes many workhours. I did this process on a crashbox; I needed about 80 workhours.

The goal of my work was to automate this process, thus reducing the human workhours required. I achieved this by a genetic optimizer and controlling the CAE (Computer Aided Engineering) software form a Python script. The program automatically makes a new shape from the base shape, runs the crash simulation with the new geometry, and evaluates the results. Then the difference between the ideal and actual behavior is calculated. The goal of the optimization is to minimize this difference.

The automated method achieved its goal. The geometries made by it shown as good behavior as the manually made best geometry in less than 24 hours. If it was given enough time, the variants made by it followed the ideal behavior significantly better, than any of the hand-made crashboxes. The drawback of this method is that it requires more simulations than the manual process, but all of this is done automatically. With my automated method the human time required for an optimization can be reduced by more than 95%.

Keywords: crashbox, frontal crash, optimization, automated, automated optimization, genetic algorithm

Contents

Köszönetnyilvánítás

A dolgozatom elején szeretném megköszönni a témavezetőmnek, Dr. Kossa Attilának az íráshoz kapott iránymutatást és a tőle kapott anyagok általi mélyebb betekintést a végeselem módszerbe. Köszönöm konzulensemnek, Tóth Péternek a használt szofverekkel kapcsolatos, illetve témában nyújtott segítségét és a dolgozat ellenőrzését. Nélkülük ez a dolgozat csak gyengébb minőségben készülhetett volna el.

Külön köszönet illeti az EDAG Hungary Kft.-t, mind a szakmai gyakorlat, mind a szakdolgozat írása közbeni barátságos, rugalmas hozzáállásáért. Ebbe többek közt beletartozik a szoftverek használatának megmutatása, a téma elkezdéséhez nyújtott szakmai segítség; valamint a nyitottság az új, korábban nem alkalmazott módszer tesztelésére, mindezt egy értékes ipari szoftvercsomag használatával. Enélkül ez a dolgozat nem jöhetett volna létre.

Szintén külön köszönöm Dr. Horváth Andrásnak a szülői és szakmai támogatását. Ebbe beleértem többek között az optimalizálási módszerekbe nyújtott, szakdolgozaton túlmutató betekintést, illetve az általa rendelkezésemre bocsátott genetikus optimalizáló programot. Ez szintén elengedhetetlen volt a dolgozat létrejöttéhez.

Budapest, 2017. december

Horváth A. Levente

Jelölések / Symbols

Minden jelölés a megfelelő indexszel a használat helyén is definiálva van. A táblázatban nem szereplő, egyedi jelölések szintén magyarázattal együtt találhatóak a használatuk helyén.

All symbols defined here are defined with the corresponding indexes at the equations where they are used. Some unique symbols are not included in the table, their definitions and explanations are found near the formulas, where they are used.

Jelölés/Symbol	Megnevezés/Name	Mértékegység/Unit
E	energyia/energy	J
E	Rugalmassági modulus/Elastic modulus	Pa
F	erő/force	N
l	hossz/length	m
m	tömeg/mass	kg
t	idő/time	s
v	sebesség/velocity	$\frac{m}{s}$
ϱ	sűrűség/density	$\frac{kg}{m^3}$

Chapter 1

Goal and structure of the thesis

1.1 Goal

The crashbox (sometimes referred to as crash box) is a very common part of passenger cars; they can be found in some form in any model from the last decade. As they are so widespread, optimizing them is a necessary task during the development of any new car model. The crashbox is part of the Crash Management System (CMS), so it's performance affects many aspects of the vehicle. These range from passenger safety to insurance cost, including repair costs after a crash.

The crashbox geometry can be optimized manually, which is the common method for this task. However, this process is time-consuming and repetitive. The optimized shapes are good, but as the problem is strongly non-linear, they are hard to create. The process involves a lot of 'trial and error', many human workhours are needed to find a decent solution.

The goal of my work was to automate the optimization process. If the automation is able to do this task, it can provide good solutions with significantly less human workhours. This method was made using a genetic optimizer and a short program (a script) controlling the commercial CAE (Computer Aided Engineering) software.

The automated method successfully achieved its primary goal. The geometries made by it became as good as the best manually optimized shape in less than 24 hours. If left running for enough time, it created significantly better geometries than I was able to create by hand. This method also requires way less human workhours, than manual optimization.

1.2 Structure

In the second chapter of my thesis, I am going to introduce the company, where I wrote the thesis, EDAG. Then I will briefly write about the software I used during my work.

In the third chapter I write about the vehicle safety systems in general, then I will describe the crashbox's role in this system. I am going to define what I will call the ideal

crashbox behavior for the rest of the thesis. After this, I present one of the widely used crash tests, where the role of the crashbox is important and its behavior can be evaluated.

In the fourth chapter, I am going to write about the created FEM model in detail. This description includes some thoughts about meshing and the used elements and material models as well.

The fifth chapter is about the manual optimization process. It describes the design principles I followed during the manual optimization as well as some observations about the problem in general. At the end of this chapter, I will present the result of manual optimization compared to the base shape.

The sixth chapter gives a detailed description of the automated optimization. First, I am going to write about the two main steps of the process in general: parametrization and the generic "optimization loop". Then I will introduce possible optimization methods briefly. As I chose genetic algorithms as my method for optimization, I am going to write about them in detail. After this, I write about the genetic optimizer I used, "GA_procq". I am going to present the simplified model created for testing the method. Finally, I am to describe the created "optimization loop" in detail, including the creation of new geometries, automated behavior evaluation and error handling in the loop. I am also going to present a short list of all runs I made during my work.

The seventh chapter includes the result of the automated optimization in three select runs. I will compare these results to each other and with the final geometry of manual optimization. I also present some selected snapshots from the evolution of the design and some local optima in one of the runs.

Finally, in chapter eight, I am going to compare the manual and automated method using several aspects, including the number of simulations and human workhours required. The comparison will be followed by some ideas, which would further improve the automated method. The thesis closes with a summary of the comparison.

The bibliography can be found in chapter nine. It is followed by the appendix, which includes some additional information about geometries only briefly presented in the main part of the thesis.

Chapter 2

Introduction

2.1 About EDAG

EDAG Engineering GmbH is an engineering company based in Germany, founded by Horst Eckard in 1969. Currently, it is one of the largest independent development partners of the automotive industry. The EDAG Group has over 8000 employees and 80 offices spread around the world. Most of these are located in Europe, especially in Germany. However, some sites are located in Asia (for example Malaysia, China, Japan) or North and South America. EDAG also has an office in Hungary, located in Győr.

The head office was initially located in Fulda. The primary focus at that time was vehicle and production plant development, which remain important even today. In the 80s and 90s, the company continued to grow in Germany and expanded to other European countries as well. These include the office in Győr, which was founded in 1999. In the early 2000s, the growth continued, reaching Asia in 2004 with the opening of the Chinese office.

In 2006 ATON Group became the sole shareholder of EDAG. This lead to strategy changes in the company, including the start of a restructuring process disposing of non-automotive related businesses. After acquiring other Engineering Service Provider (ESP) companies, EDAG became one of the leading ESPs for the automotive industry. In the upcoming years, multiple concept cars were made for the Geneva Motor Show and the International Motor Show. These include the EDAG "Light Car - Open Source" in 2009, "EDAG GENESIS" showcasing the benefits of additive manufacturing in 2014. The latest concept vehicle is "EDAG Light Cocoon", which was presented in 2015 at the Geneva Motor Show. (Source: [7], [8])

2.2 Used software

2.2.1 Ansa

Ansa is the product of an engineering software company, BETA CAE Systems. BETA CAE Systems develop software systems, including the Ansa preprocessor, the EPILYSIS solver, and the META post-processor. The name "Ansa" comes from the original purpose: Automatic Net generation for Structural Analysis.

Ansa is an advanced preprocessing software, which includes the tools to build full models from CAD data. These models can be built for several CAE software, including Abaqus, LS-DYNA, Pam-Crash, etc.

One of the more powerful features of Ansa is the ability to create and automatically execute scripts. These can use commands from both Ansa itself and generic commands for programming. (Source: [9])

2.2.2 Pam-Crash

Pam-Crash is a FEM solver and product of ESI Group. Its predecessor originated from a research aimed at simulating aerospace and nuclear applications. At a meeting organized by VDI (Verein Deutscher Ingenieure) in 1978, a simulation of an accidental crash of a military fighter plane into a nuclear power plant was presented using this software. German automobile manufacturers saw potential in the software to simulate destructive car tests.

After years of development in cooperation with large automotive companies, Pam-Crash was created. As it was based on FEM, it enabled modeling complex geometries by using different structural elements. In 1986, the software successfully ran the simulation of a frontal impact of a full car structure. This model used only 5500 elements, but the results were accurate enough to help the design process.

Today, Pam-Crash is capable of several other types of simulations, including crash, occupant safety, statics and dynamics, noise, vibration and harshness simulations. (Source: [10], [11])

2.2.3 Animator 4

Animator 4 (or GNS Animator 4) is a FEA post-processor and the product of GNS and CDH AG. It is designed to handle large finite element models from several solvers, like Abaqus, Pam-crash, Nastran, etc.

The development of the predecessor of Animator 4 began in the early 90s. The first commercial version was released in 1996 under the name Animator 3. In 1998 CDH AG acquired rights from GNS, the original developer of the software. The successor of Animator 3, Animator 4 was released in 2009, and it is a commonly used post-processor in the automotive, aerospace and chemical industry.

Animator 4 allows the user to create so-called "session" files, which can be used to execute commands written in Animator 4's command language for batch processing. (Source: [12])

2.2.4 Python programming language

Python is a programming language created by Guido van Rossum in 1989. Since 2001, it is developed by Python Software Foundation. It is a high level language for general programming. It provides a fast and easy program development, but it's run performance can be considerably slower than code written in C++ for example. (Source: [14], [13])

Chapter 3

Description of the crashbox

3.1 Automobile safety systems

Automobile safety systems can be assigned to two main groups: active and passive systems. The term "Active safety system" is used to refer to technology assisting in the prevention of a crash. These include Driver Assistance like Automatic Braking, Backup Cameras, Anti-Block Systems (ABS) and several other technologies. These systems, however, do not protect occupants or other people once the crash happens.

On the other hand "Passive safety systems" refer to components of the vehicle that are passive before the crash. They become active once the accident happens and help to protect occupants during a collision. These systems are also responsible for pedestrian safety in case of a runover accident. The most important technologies protecting the occupants are the airbags and seatbelts. The structure of the car plays a role in both occupant and pedestrian safety. Some systems only protect the pedestrians, for example, external under the hood airbags aiming to prevent severe head injuries. (Source: [15], [16])

The main priority of these systems differs based on the speed of the crash.

1. At 15 km/h or lower (low-speed crashes) the main goal is to minimize the repair cost of the vehicle.

2. Between 15 and 40 km/h, the main concern is pedestrian safety.

3. Above 40 km/h occupant protection takes priority.

3.2 The crashbox

The crashbox is one of the passive structural elements, which absorbs kinetic energy by plastic deformation. Its role is most significant during low-speed crashes. A pair of crashboxes is located both on the front and back of the car frame structure. Most crashboxes are made of steel or aluminum sheets, but extruded profiles are available for

aluminum designs. Cost efficiency is an important factor, as the purpose of this part is to reduce the repair cost by protecting the more expensive ones. (Source: [16])

The most widespread crashbox geometry is one with a closed, rectangular cross-section, which may consist of multiple parts. It usually has push-ins on at least two sides to control the deformation process during the crash. Many designs have holes on some walls for mounting other parts.

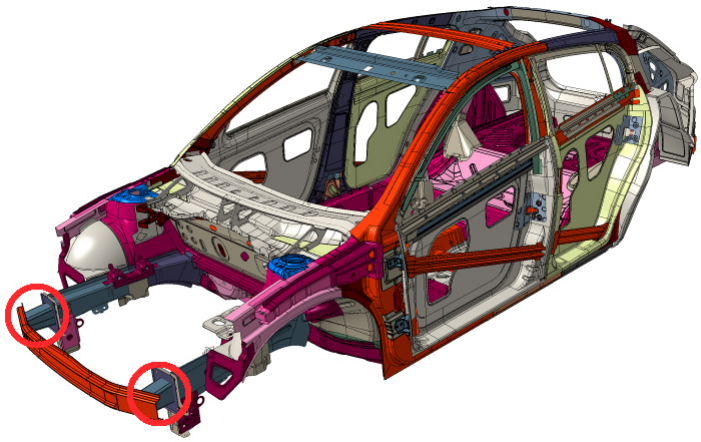

Figure 3.1. The location of the frontal crashboxes (Source: [17])

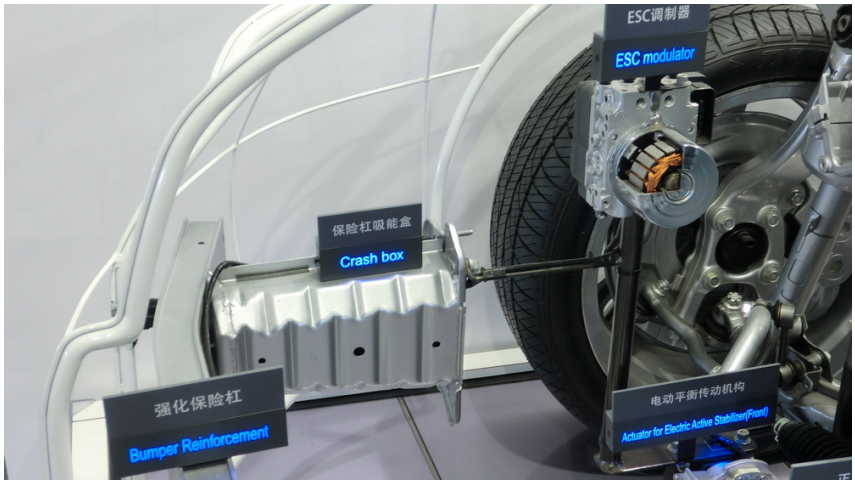

Figure 3.2. A real frontal crashbox geometry (Source: [18])

3.3 Ideal behavior

Crashboxes have to satisfy conflicting criteria. One of the most important parts to protect is the car's main beam frame as this piece is expensive to repair. Car frames can only bear a limited amount of longitudinal load before taking damage. For this reason, one of the design criteria what the crashbox has to fulfill is that the transmitted force from the

crashbox to the main frame should not exceed a defined value. This limit is typically 120 kN, so this is the value I have used during my work.

The second criterion is that the deformation length should be minimized. This aims to protect more fragile parts in the zone of deformation, for example, the front cooler ventilation system. These parts are not as expensive as the main beam frame, but their protection should not be ignored.

The kinetic energy of the car is determined by its speed and mass, so the kinetic energy is defined by the crash scenario. Analytically calculating the energy absorbed by the deformation of the crashbox would require knowledge of continuum mechanics. Instead of trying to get the exact integral equation, we can approximate the energy absorbed by the crashbox. We know that conservation of energy is true in a closed system. Assuming that crashbox is the only part absorbing the energy, we can estimate the absorbed energy with equation (3.1). $E_{absorbed}$ is the energy absorbed by the crashbox, F_{crash} is the force crushing the crashbox and $l_{deformation}$ is the deformation length.

$$E_{absorbed} = F_{crash} \cdot l_{deformation} \tag{3.1}$$

The crushing force is almost parallel to the direction of the deformation. Thus the energy can be calculated as a simple product. The force crushing the crashbox equals to the force transmitted to the main car frame, as the center of gravity remains still during the crash. As the energy is defined by the crash scenario, the minimal deformation length can be achieved by maximizing the transmitted force. Increasing the force can be done until it reaches the limit mentioned previously. This concludes the ideal crashbox behavior: the transmitted force should reach its upper limit as soon as possible, and stay on that value with no or minimal fluctuation. When the kinetic energy has been absorbed, the displacement no longer increases, and the transmitted force falls to zero.

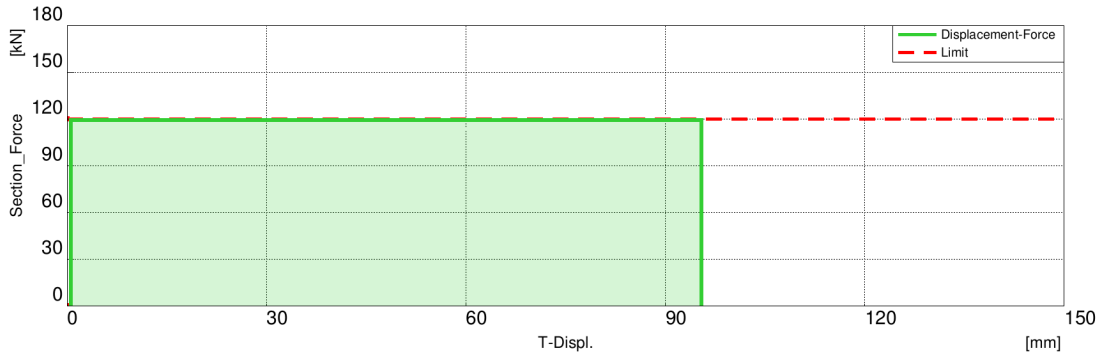

Figure 3.3. The ideal crashbox behavior

As the most important values of interest are the displacement and force, the primary tool for evaluating the behavior is the "Displacement-Force" curve. In Figure 3.3 we can see this curve for the described ideal behavior Unfortunately it is impossible to achieve

in practice, it can only be approximated by real designs. The goal of the geometry optimization is to get as close to this ideal behavior as possible.

If the deformation of the crashbox is the only process absorbing kinetic energy then the minimal crashbox length can be estimated using the following formula:

$$l_{min} = \frac{\frac{1}{2} \cdot m \cdot v^2}{F} \tag{3.2}$$

3.4 Low-speed frontal crash test

As mentioned before, the primary goal of the Crash Management System during a low-speed crash is to minimize economic damage. Several different protocols exist to evaluate the potential damage during such a collision, made by various organizations. I have used RCAR's "Low-speed structural crash test protocol - Issue 2.2" as a base for my simulations [19]. The protocol contains details for both front and rear crash tests. I did my optimization for the front crashbox, so I'll only write about the frontal crash test specifications.

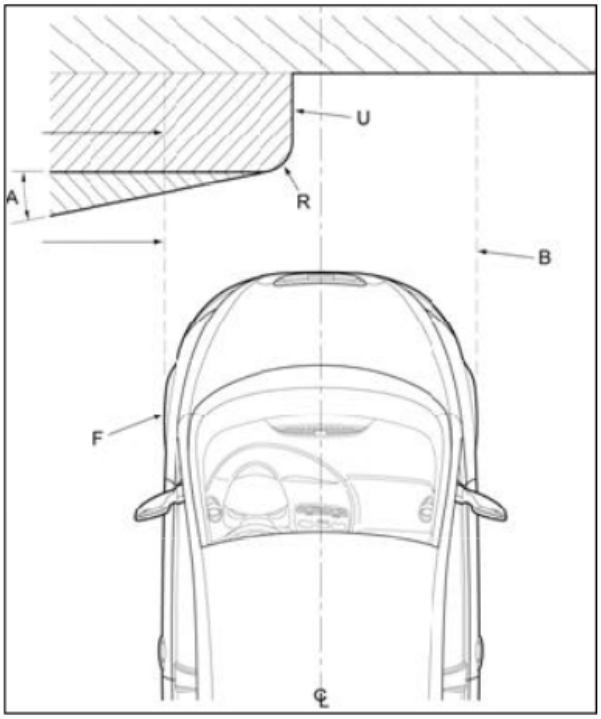

Figure 3.4. Frontal crash test setup (Source: [19])

In this test the vehicle is propelled towards a rigid barrier at 15 ± 1 km/h. The barrier's front face is oriented 10° relative to the perpendicular of the longitudinal axis of the vehicle. The impact happens on the driver side with 40% overlap. The radius of the barrier's corner is also specified as 150 mm. The setup can be seen in Figure 3.4.

I have slightly changed some of these values in my simulations. My vehicle impacts at 16 km/h. The angle of the barrier can be either 0° or 10°. I will specify this value for each

result shown and write about why I have made this adjustment for certain simulations. I did not simulate a complete car model because that would have drastically increased the simulation time. Instead, I mounted the crashbox and belonging parts on a rigid, simplified car model. This model had the size of an average car and weighed 1440 kg.

During these tests, the vehicle typically impacts the barrier, but its speed is not reduced to zero. Instead, the vehicle "bounces back" and slowly rotates counter-clockwise, if we use the directions shown in Figure 3.4.

Chapter 4

Details of the created model

The frontal crash test simulation was done in Pam-Crash. For this I had to use the explicit solver, meaning that the state of the model at a specified time step was calculated from the state of the previous time step alone. A detailed descripoin of explicit methods can be found in: T. Belytschko: Nonlinear Finite Elements for Continua and Structures [1].

4.1 Finite element model

Each Pam-Crash simulation has a main '.pc' file. It contains the settings for the simulation, and it may include element, boundary condition or other definitions. However, in most cases the '.pc' file refers to '.inc' include files and the definition are made in these '.inc' files. This method allows for a flexible system, where modifying a certain part of the simulation is done by changing its include file. These files may have references to each other, for example, one of the include files might have all material models defined. In this case, all other includes only need references to the material include file.

My simulation consisted of four include files. All of these are standard files for frontal crash simulations at EDAG Győr, which I had to modify if needed for my crash scenario.

1. cmsvar01.inc — This file contains the mesh made for the crashbox geometry. If a new version is made, only this file needs to be updated.

2. barriere.inc — This is the file containing the rigid model of the car frame and road surface.

3. phal2.inc — This file contains the impact barrier, which the vehicle collides with.

4. mater.inc — This file contains the data for all material models used by the simulation.

The crashbox is connected to the main frame of the car via a node set with a specific name. These nodes have their location fixed in the local coordinate system of the car frame.

Following the industry standard for car modeling, the X axis points from the front of the car to the back; the Y axis points from left to right and the Z axis from bottom to up. The symmetry plane of the car frame is the XZ plane.

The transmitted force from the crashbox to the rest of the car frame is measure using "Secfo". The name stands for "Section Force", and it is a defined by selecting a cross section on the geometry. The Secfo is located near the base of the crashbox, and it measures the X, Y and Z components of the total force applied to the cross-section by the front side of the crashbox.

I used Node 500 as the reference node for the displacement. It was located on the horizontal bar connecting the left and right side crashbox, roughly in the middle of the connecting area.

The location of the Secfo cross-section and Node 500 can be seen in Figure 4.1. The crashbox shown is a simplified model, which is discussed in detail later.

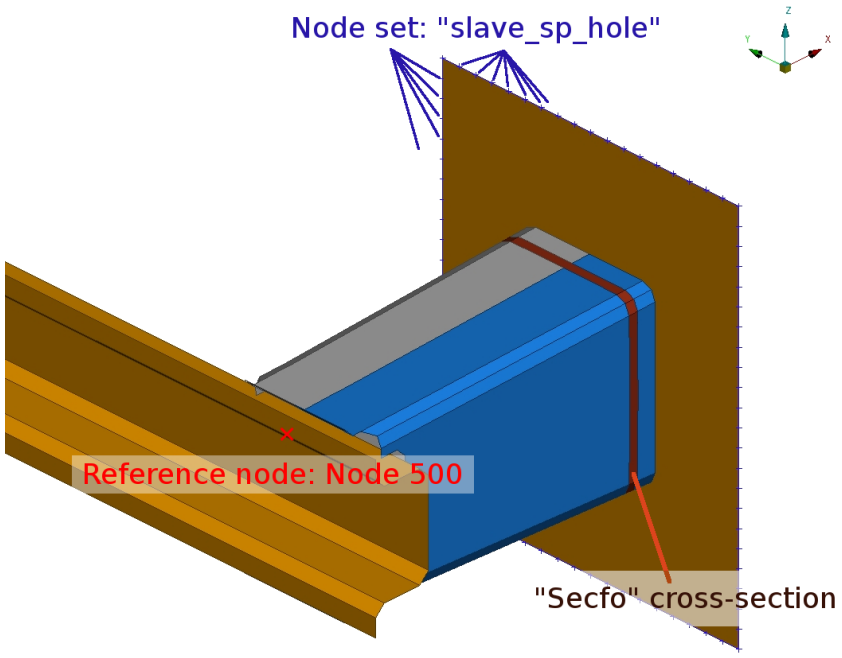

Figure 4.1. The location of the reference node, "section force", and the connection node set

4.1.1 Meshing

I got the CAD geometry of an existing crashbox with all connecting parts to optimize. These parts were the left and right side crashbox, the horizontal bar-like piece connecting them and another horizontal bar under the previous one, which is connected to the crashboxes via two vertical parts. Unfortunately, I'm not allowed to include an image of the exact geometry due to the company policy for intellectual property protection, as this is a geometry taken from an existing car. I'm only allowed to show the crashbox itself.

The geometry was meshed with quad and tria shell elements to the middle surface of the parts. The target edge length was 5mm, which is a standard size for metal sheet parts in crash simulations. I also used a mesh quality file made for these simulations to ensure that the mesh quality is good enough to get the best results. The completed mesh had about 25000 shell elements.

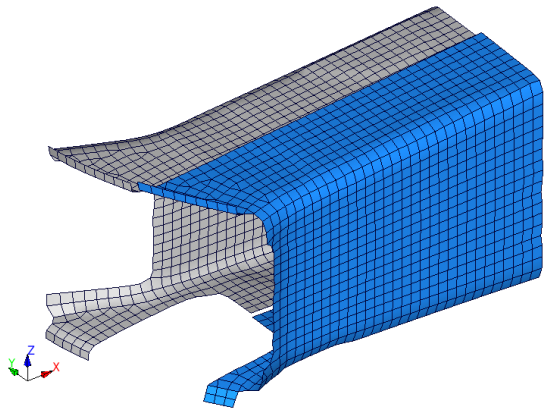

Figure 4.2. The meshed left side crashbox

The two colors in Figure 4.2 represent separate properties in Ansa. This allows modifying the material type, thickness, etc. independently on the two halves of the crashbox. I have always kept the same material for both, but I have modified the thickness individually. These two halves are welded together.

4.1.2 Used elements and models

My model was built entirely from shell elements. The crashbox was made out of H340 LAD steel. This material is used fro crashboxes for its high yield strength of 340MPa, good properties for part manufacturability and relatively low cost. This material is produced in sheets with 0.3 to 3mm thickness. (Source: [20])

The rest of the structure was made out of BTR 165 steel. This is an ultra-high strength steel alloy commonly used for safety beams in vehicles. (Source: [21])

The element of choice was the standard "thin shell". These finite elements discretize shell surfaces with thickness t. To calculate the nonlinear stress distribution resulting from plastification in the element Pam-Crash uses numerical integration in the thickness direction. The user can specify the number of integration points. Using only one integration point causes membrane-like behavior. Tree points are sufficient for calculating bending and membrane effects in most cases. More points can be applied to simulate springback effects. I have used five integration points for all shell elements.

The material used for the deformable parts was "Material 103", which is described as an Elastic-Plastic material for Shell Elements. 103 uses a plasticity algorithm that includes transverse shear effects. It exactly satisfies Hill's criterion and precisely updates

the element thickness during plastic deformation. Unfortunately, the exact equations and theory used by the program are not available for the user.

The exact material properties, including the "true stress-true strain" curve, are defined in 'mater.inc' for modeling plastic hardening.

For the rigid bodies "Material 100", which is sometimes referred to as the "Null Shell Material". It is used for performance-friendly modeling of rigid bodies. (Source: [22], [23])

4.2 Boundary and initial conditions

The road surface and impact barrier are rigid bodies with all of their degrees of freedom fixed. The car frame is also a rigid body; it's free degrees of freedom are movement along the X and Y axis, rotation around the Z axis. The elements of the car frame do not have enough weight to achieve the desired 1440 kg, so the required mass is added to the frame as nodal mass distributed equally on the nodes.

As mentioned before, the crashbox is connected to the frame via a node set named "slave_sp_hole". The location of this set is marked in Figure 4.1. It includes all nodes at the edge of the connection plate. These nodes are not part of the crashbox directly; they are part of a sheet behind it. The crashbox is connected to the sheet via "TIED" contacts, representing the welding. TIED contacts use the existing nodes and elements of the mesh. The welding between the two halves is represented by "PLINKS". These can connect parallel elements, as long as they have some overlap when looked at from their surface normal's direction. PLINKS are treated as separate elements during the simulation and they are independent from the mesh.

Gravity is applied as an acceleration field with -9.81 $\frac{m}{s^2}$ magnitude to the Z direction. The initial speed of $4.444444\frac{m}{s}(\approx 16\frac{km}{h})$ is applied to the frame and crashbox parts as well.

All of the elements are part of a group, named "Selfcontact". This group has a "Type 33" contact applied to it, which means all elements collide with each other. This prevents elements of the parts penetrating each other.

Chapter 5

Manual optimization

5.1 Method

After preparing the simulation, my task was to optimize the crashbox geometry and sheet thickness to follow the ideal behavior as closely as possible. The manual optimization was done with a 10° barrier. I have tried to follow some guidelines during the process. I did not want to make any major changes in the shape of the crashbox. For this reason, I slightly adjusted the angle of the crashbox walls and tried to control the deformation process via push-ins and push-outs. I changed the material thickness in 0.1mm increments but never used more than 3.0mm, because this is the thickest plate made out of this material I was able to find.

Push-ins were useful as deformation initiators. Even a slight push-in was able to manipulate the starting point of the collapse effectively.

I have tried to use push-outs to give the structure some extra strength against bending in the selected cross-sections. However, they were not too sufficient for this purpose, so they served as folding edges for the geometry. It was more effective to strengthen the design by increasing the material thickness, then creating push-ins to make it weaker at critical cross-sections.

The "displacement-force" curve was the main tool I have used for evaluating the results. I used Animator 4 to create it. To get the displacement values I had to fix the viewpoint location in the local coordinate system of the car frame; then I could plot the absolute value of the displacement of the reference node in this coordinate system. The force per coordinate direction was read from the Pam-Crash result file, and the components were summed by Animator. These two curves were then combined to get the "displacement-force" curve.

The evaluation of the curves did not have any particular tools. I optimized the geometry by modifying the mesh by moving some of it's nodes, paying attention to mesh quality. Then I ran the simulation and used to the curve to determine where I needed to make the crashbox stronger or weaker, based on the force values. For example, if the transmitted force significantly increased before a new fold was made, I tried to apply

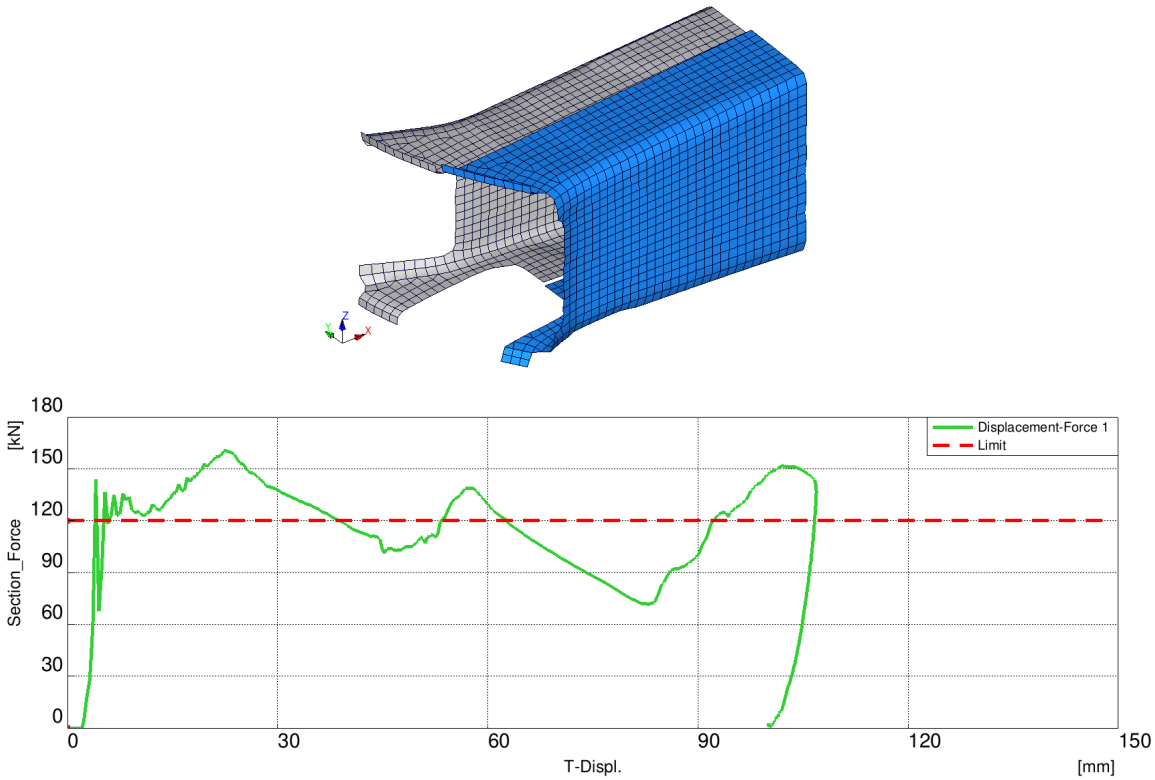

Figure 5.1. The base geometry and it's Displacement-Force curve

push-ins to reduce the force required. The typical height of the applied push-ins was 2mm.

In figure 5.1 we can see how the base geometry performed. The material thickness on both sides was 2.7 mm. The average of the force is roughly at the desired 120 kN line, but it is oscillating heavily around it. Appendix A contains additional images of the deformation process.

5.2 Observations

On figures like 5.2 the green line always belongs to the variant shown on the left and the blue line to the right side. The white-gray color scheme of the crashbox will be used for presenting the manually created variants.

I have found that this problem is very strongly non-linear. A rather small change - especially on the rear side of the crashbox - can have a dramatic impact on the way the process behaves. In figure 5.2 we can see two very similar variants. The only difference being is a slight push-out on the back side. However, the difference in behavior is major.

As we can see variant 48 is significantly better: up to 60mm deformation, it follows the limit more closely. It also never goes over the limit, while variant 47 goes over 150kN at the end.

One of the hardest part of the curve to optimize was the very last one, from approxi-

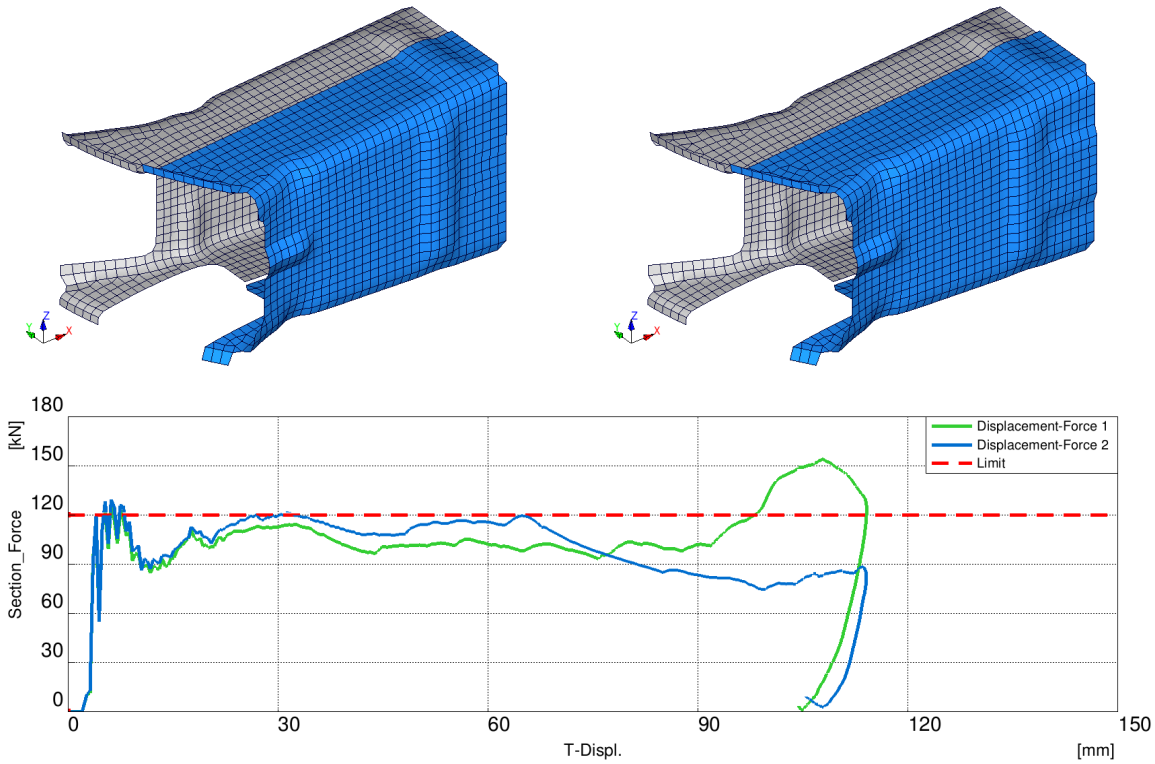

Figure 5.2. Variants 47 and 48

mately 90mm deformation length. The main issue was "self-collision": when the deformed layers of the collapsed crashbox touched. As the "flow" of the force was able to skip some geometry when this happened, it caused a major increase in the transmitted force.

This phenomenon is demonstrated in figure 5.3, where the red square on the green curve (at roughly 85mm deformation length) shows the time step of the figures shown above. We can also see an orange part, which is the top part of a vertical bar connected to the crashbox. This bar amplifies the problem further: not only is it not crushed like the rest of the geometry, thus 'sweeping' it's way across the lower wall of the crashbox, it also provides a direct path for the force to 'flow' through. Best seen in the right-hand image, contact was made between the deformed layers on the lower and upper side as well. After this, the transmitted force continues to grow until the vehicle slowly bounces back from the barrier.

As I can not change the connected part, the only way to mitigate this problem is absorbing more energy before self-collision happens, thus reducing the negative effects.

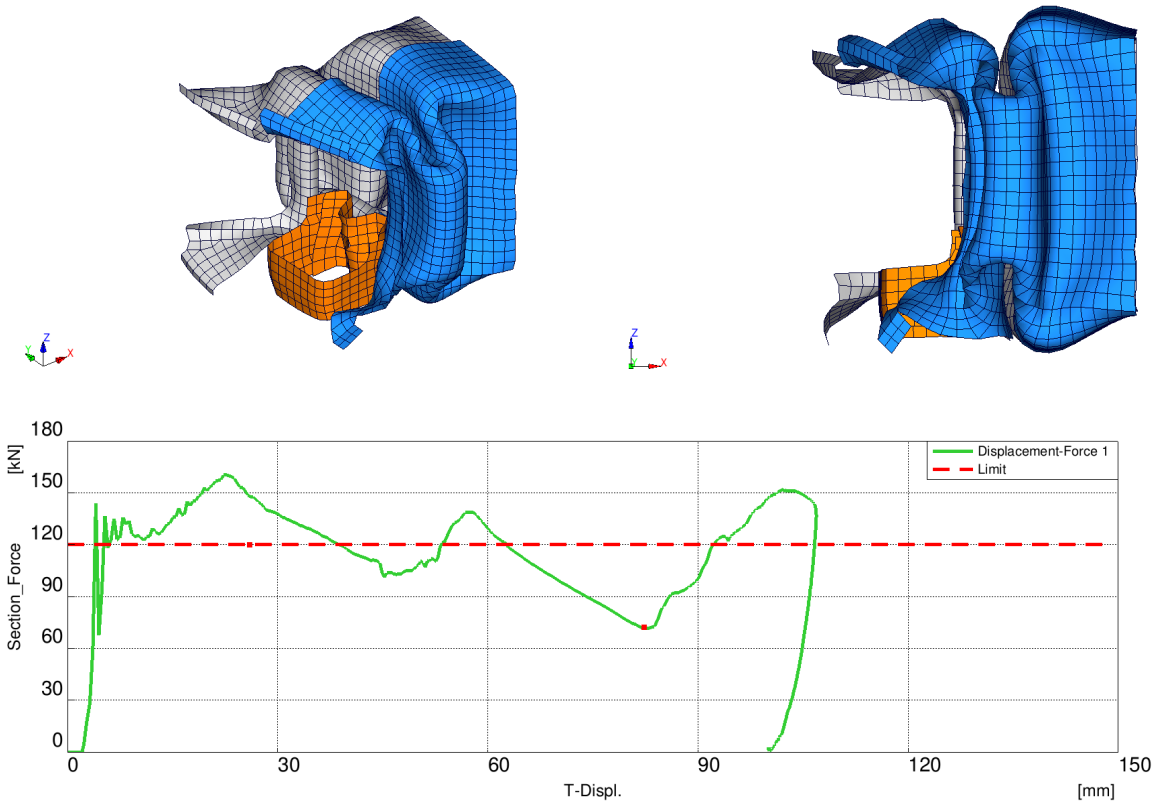

Figure 5.3. The result of self collision

5.3 Results

I was working on the manual optimization for about 80 workhours. I ended up with more than 100 variants in total. I did not have an objective way of determining which is the "best" variant. I was able to make a decision based multiple viewpoints, for example, the maximum force values, deformation length, etc.

In Figure 5.4 we can see the final result compared to the base variant. The material thickness was changed to 2.6mm on the white and 2.8mm on the blue half. The optimized shape follows the ideal behavior quite closely. The maximum of the force is also reduced considerably to approximately 130 kN.

Deformation initiator push-ins are located on the front, on both the upper and lower edges, both left and right. The push-out on the front part of the blue half provides some extra help to initiate the collapse of that side as the wall rotates. Push-outs are located on the left, right and top walls to help the folding of the walls. Additional push-outs are located on the top and bottom edge of the blue half. Most of these changes are easy to manufacture, except the push-outs along the edges.

It is important to note that during a real crash there are other car parts which absorb energy like the engine hood or bumpers. In my setup the crashbox absorbs 70-80% of the kinetic energy, about 15% is absorbed by the horizontal bar, and some energy is used up by the car bouncing back after the impact. (Source: [2]) If all parts were included, the

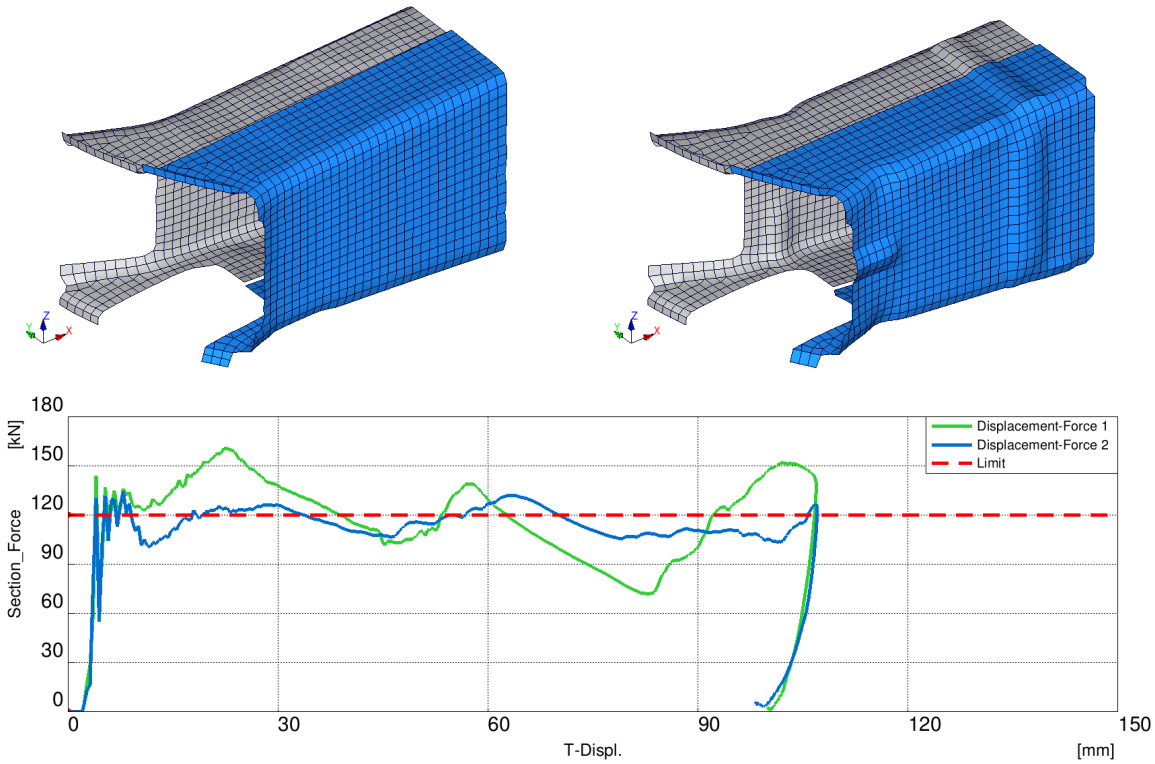

Figure 5.4. The base shape and manually optimized shape

curve would be 'pushed' towards lower force values. It could also mean that self-collision is no longer an issue, due to the vertical bar being slowed down sooner.

Chapter 6

Automated optimization

The idea of automation grabbed my attention because many steps were repetitive during the manual optimization process. It is made possible by the ability to use scripts in Ansa, written in either Python and Beta's script language. I used Python because I was already familiar with its fundamentals and syntax. The vast majority of commands available on the user interface are also available from the script in some form. Calling other programs is also possible with Python.

Two main steps are essential for automated optimization. First of all, a set of parameters must be chosen. We can also call them the "design variables". I will call the number of parameters N from now on. These will be the values, what the program can modify later to create a new variant. Choosing the right parameters is quite important because these define the possible modifications for the optimizer. Increasing the number of parameters gives more freedom to the program, thus achieving better results is possible. However, the optimization process will be slower if the parameter count is higher, as the number of possible combinations is increased. For this reason, only the most influential parameters should be used. The chosen N parameters define an N-dimensional space named the "parameter space".

The second step is to create a system, which I call the "optimization loop". The purpose of this loop is to create better and better variants until stopped. When started, the loop will generate new parameter values and create a new variant based on these. Then it runs the simulation with this new geometry and evaluates the results. The results are returned to the optimizer, which tries to create a better variant based on the information gathered from previous attempts. This loop keeps running until a stop sign is given.

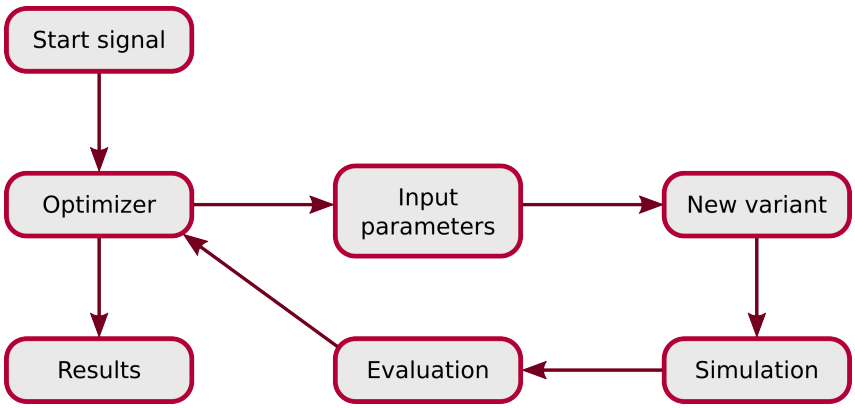

Figure 6.1. Flowchart of a generic optimization loop

6.1 Possible optimization methods

There are several methods available for the optimizer program mentioned above. Each has its own benefits on drawbacks, which must be considered when choosing the right method to use in the loop. These include the speed and applicability of the method, as some optimizers are only usable on certain (e.g., continuous) functions. The function, which the optimization is performed on is often called the "objective function". The optimization goal can be either minimizing or maximizing the objective function. All of the mentioned automated methods require a high number of iterations to get good results, but the precise number differs from method to method.

An easy, but flawed approach is "per-parameter" optimization. The idea of this method is to change the parameters one-by-one to their optimum, so the single N dimensional optimization task is split up to N one-dimensional problems. Each parameter has a fixed value, but the one we are optimizing. Once the optimum has been found for the first parameter, then its value is going to be fixed. Then the algorithm changes the second parameter to find its optimum etc. Once the last parameter is optimized, the algorithm return to the first parameter and starts over the process.

Unfortunately, this is a fundamentally flawed method. The minimums or maximums of multi-dimensional function cannot be determined by the first partial derivates only. The Hessian matrix must be used for this purpose, and it cannot be approximated by this method like the first partial derivates. In Figure 6.2 we can see a "false optimum" found by this method. Both partial derivates are zero, the second partial derivates are negative, but this is not a real optimum.

"Random search" is a method, where the program tries random combinations. Its main benefit is that it is applicable for even the most extreme of functions. The function can be non-continuous or non-differentiable, may have any number of local optima, etc. The main drawback is that this algorithm is very slow to find a good solution and it can not refine them quickly. Some strategies exist for these methods, for example, to search in a certain radius of the known best variant. However, random search should not be used

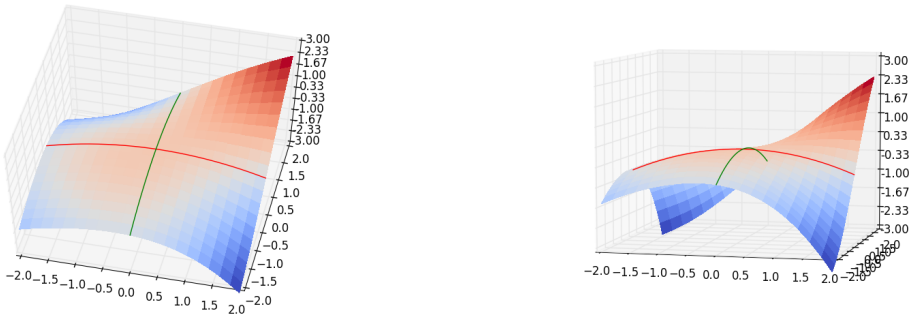

Figure 6.2. False optimum found by the "per-parameter" method

when other algorithms are available because it is very slow to make improvements.

"Grid search" is a method, where the program tries every parameter combination with a defined resolution. Let M be the number of values we check each parameter in. This means that we perform the investigation on an N dimensional grid, and check M values per dimension. The total number of combinations is M^N. This algorithm is also simple, and it spreads the design points evenly in the parameter space. The most important disadvantage is that as either N or M increases, the number of required simulations goes up drastically. For example, if $N = 5$ and $M = 5$ then $5^5 \approx 3000$ simulations are needed, which is a lot, but it is still a manageable amount. However, as we are going to see later, I am using 18 parameters, with $M = 5$ this would require over $5^{18} \approx 3.8 \cdot 10^{12}$ simulations. This is an impossibly high number to handle for current computers. Even if we could run 100 simulations per second (in reality, one simulation needs about 250 seconds, so this is an over 10000 times speed increase), it would take over 1200 years to finish the calculations. Additionally, this method would not find the optimum, it would only give a very rough estimation of its position in the parameter space.

Gradient-based methods are commonly used algorithms, which do optimization based on the objective function's gradient. Two typical versions are the method of "steepest ascend" for maximization, and the "steepest descend" method for minimization. A common strategy is to take steps proportional to the gradient vector with every iteration. These methods can provide good results very quickly, if some conditions are met. The most important thing they require, is it object function's gradient. Unfortunately, it is only possible to obtain if the object is differentiable. In many optimization problems - including mine as well - the analytical formula of the objective function is unknown, as the function is calculated numerically in specific points of the parameter space only. This makes gradient-based methods impossible to use. The other issue with these algorithms is that they tend to converge to local optima. As my objective function is expected to have many local optima, this tendency would most likely cause the optimizer to miss the global optimum completely. Of course, this would result in significantly worse results than ideal. Overall these methods are best suited for differentiable object functions with a single op-

timum, where they provide good results quickly. However, they are not applicable to my problem.

Another potential issue with all of the methods mentioned above is that they cannot treat flawed object function evaluations properly. I am going to write about this in detail later. However, I must mention in advance that it is a possibility what the program has to be prepared for, as it can happen as a result of factors which I cannot control.

Of course, there are many other methods which I didn't write about in this section, but these are some of the most widely used methods. Additional methods and more detailed descriptions can be found in: [3], [4].

6.2 Genetic algorithms

After considering the possibilities, I have chosen genetic algorithms as my optimization method. These algorithms are inspired by biological evolution. As a result, this topic has its unique terminology, which significantly differs from the one used for other methods.

In this field, the parameters are often referred to as "genes". The variants created by giving the values to the genes are called "individuals". The full parameter set which determines an individual is named "genome". Each gene has a specified range of values to use or a pre-defined set of potential values.

Individuals are rated by the so-called "fitness function". This is the indicator of how good the individual performs compared to the others. The primary component for calculating the fitness is the value of the objective function value of the individual. Other factors can be considered as well, for example, to help to maintain the diversity of the different genomes. This can be achieved by lowering the fitness of those individuals, which have a better one very close to them in the parameter space. This ensures that a group of individuals very close together does not suppress every other individual and gene combination. Like in real population, genome variety is important to maintain. If the population only contains individuals with similar genomes, the optimizer has a high chance of getting stuck on a local optima located near them.

Most - but not all - genetic algorithms are "generation based". A generation is a set of defined number individuals created in the same iteration. They typically include 10-100 individuals. The initial generation is filled with individuals using random genomes. After the generation is created, the fitness value is calculated for each individual. Then the next generation is created using the results from the previous generation. The algorithm can use different ways to do this, for example, it may pick one of the best individuals and slightly change some (or all) of its genes. This step is called "mutation". Alternatively, the algorithm may combine the genomes of two of the best individuals. There are several ways to define the combination method, for example: use a pre-defined or random-generated mask, where the mask determines which gene is used in the new individual from the previous genes. Using linear (or some other type of) combination is also possible. This

step is called "cross-over". These were of course just some examples of the possible methods to create the new individuals, many other ways exist. After the new generation is defined, calculating the fitness for this iteration may begin. When it is done, a new generation is created for the next iteration. After evaluating some populations the algorithm will start to find better and better genomes. The bad individuals are either deleted or get forgotten as the algorithm progresses.

One of the most important benefits of genetic algorithms is that they are applicable to almost any problem. Unless the object function is strongly non-continuous, this method is usable and will find good results. As randomness plays a vital role in the search, these algorithms will not get stuck on local optima, which was one of the major issues with gradient-based methods. When the algorithm is set up correctly, it will find the proximity of the global optimum in the vast majority of the time. It is important to note, that due to the random factors it is impossible to perfectly replicate an optimization run. The algorithm will use different starting genomes, combine them differently, etc.

One drawback is that 'purely' genetic algorithms - the ones which do not contain steps inspired by other optimization methods - are very slow to refine the individuals near the optimum. For this reason, they often use steps inspired by gradient-based methods. One of these steps can be defined as taking the current best genome, and another good genome from its proximity in the parameter space. Then linear extrapolation of their genes is used to create the new individual. This step is called "hill climbing".

The other disadvantage of these methods compared to the gradient-based ones is that genetic algorithms are quite slow. If a gradient-based method is applicable, it usually finds the optimum significantly faster. In extreme scenarios - where the objective function suits gradient-based methods - genetic algorithms can be 100-1000 times slower. With the parameter count and objective function complexity I have, genetic algorithms need to evaluate 2000-3000 individuals to get good results.

However, genetic algorithms are very robust. This means that they can handle failed object function evaluations. As the algorithm forgets about bad individuals over time, if the failed evaluation is given a bad fitness value it will barely affect the results overall. It is also possible that the same genome will be rechecked after it got forgotten.

It is important to note, that giving a good individual a bad fitness value as the result of an error will not have a major impact on the overall results. However, if a bad individual is given a good fitness value, the algorithm will not forget about it. As it has a good fitness, it will have a high chance of getting mutated or crossed over with other genomes. Thus it will continue to mislead the optimizer and 'poison' the population. For this reason, giving bad individuals good fitness values is to be avoided at all costs. (Source: [5])

6.2.1 The "GA_procq" optimizer

I have used the program named "GA_procq" for optimization. This software is developed by András Horváth Ph.D. at the Department of Physics and Chemistry and Zoltán

Horváth Ph.D. at Department of Mathematics and Computational Sciences at Széchenyi István University. GA_procq is a software primarily for research, but it has been used for optimization in industrial projects as well. A related publication can be found at: [6].

The name stands for "Genetic Algorithm Process Query" as an indication of the philosophy behind the program. It is designed to evaluate the objective function using external software. This philosophy means that this optimizer can be connected by any software using a straightforward file-based 'interface'. There is no limit to the complexity or type of the program structure used to get the objective function value. This evaluation structure is often based on a script, which starts the required external programs in order, then waits for their finish.

GA_procq uses floating point numbers as the numerical value of the genes. The only requirement to fulfill when connecting a program with GA_procq is to follow the defined input and output file format. When a new individual is created, GA_procq writes it's genome in the following format into a text file named 'in.fg':

```
N | gene-1 gene-2 gene-3 ... gene-N | 0 | 1 |.
```

This format is what the objective function calculating program has to recognize. The file extension '.fg' stands for "float gene". N is the total number of genes, 'gene-N' is the numerical value of the Nth gene. The 0 after the gene values indicates that this genome has not been evaluated yet. The last number - in this case: 1 - indicates the dimension of the objective function result. GA_procq is capable of multi-objective optimization using an objective function returning its result in up to 4 dimensions, but I have not used this feature. Thus I will not write about multi-objective optimization in detail.

GA_procq waits until the objective function calculation is finished and expects the output to use the following format in the file named 'out.fg':

```
N | gene-1 gene-2 gene-3 ... gene-N | 1 | 1 | result | ZZ.
```

This is quite similar to the input format. The 0 is changed to 1, indicating, that this is a genome is evaluated. Additionally, 'result' is the object function value written after the genome. 'ZZ' is a required string at the end of these lines.

The available interval of each gene is defined a file named 'minmax.dat'. This file consist of lines using the following format:

```
n min1 max1
m min2 max2
k min3 max3
etc.
```

This format means, that from 'gene-1' to 'gene-n' genes get their value from the specified (min1 max1) interval. From 'gene-(n+1)' to 'gene-(n+k)' they get values from the (min2 max2) interval, etc.

When it is running, communication with GA_procq is done by creating files with specific names in the folder where GA_procq is running. The name of these files always

starts with 'comm.', short for communication. The content of the file does not matter; it can be empty. The program always checks for these files after finishing the ongoing evaluation, so it reacts after that. A file named 'comm.stop' will stop GA_procq after the current evaluation. 'Comm.lock' will pause the program, so it will not start further evaluations until this file is deleted. After a command is recognized, the 'comm.' file is deleted, except 'comm.lock', which the user has to delete manually to continue the iterations.

While running GA_procq will create two '.fgs' files, where all the results are stored. 'Fgs' means 'float genes', and these files can be viewed at any time by the user, making it easy to tack the progress of the optimizer. 'Allresults.fgs' contains every single individual's data in a similar format to the 'out.fg' file. As soon as the evaluation of an individual is complete, its data is put in this file. 'Best.fgs' contains data of those individuals, which were the best in the population for some time. This file makes it quite easy to track the improvements, as it is only updated when the best individual is improved.

Opposed to most other genetic algorithms, GA_procq is not generation-based and does not have a fixed population size. Instead, it creates a configurable number of individuals to start with, then creates new ones one-by-one instead of in generations. When the population gets more than a certain amount of individuals, it sorts the population based on fitness and only keeps the best individuals. This strategy has multiple consequences.

First of all, this allows GA_procq to start refining a good genome immediately after it has been discovered. Generation-based methods would only recognize this after evaluating the entire generation. This more 'aggressive' approach makes improvements on the optimum considerably faster. However, it yields slightly worse results than the generation-based methods in the long run, because some time is wasted trying to refine local optima. This drawback is only noticeable on runs with a very high number of iterations. In most cases, the optimizer will be shut down before reaching this stage, because the results are 'good enough' already. Overall the non-generation-based approach is worth it because it starts to refine the genomes more aggressively and its drawbacks are not noticeable on the number of runs most problems in the industry require.

Another feature this makes possible is reacting to user input immediately after the current evaluation finishes. This is true for the 'comm.' files mentioned previously, but those are not the only option for user input. The user can create a file named 'readme.fgs' which may contain both 'in.fg' or 'out.fg' lines. Once the file is read, GA_procq will evaluate all lines using the input format and integrate them into its own population. As output lines already have their objective function values, they are integrated without the need to re-evaluate them.

This is a powerful feature, as the output lines can be collected from a different PCs or from the result file of a previous run. This behavior means that creating a network of several computers running GA_procq and sharing results is easy to do when needed. Unfortunately, I did not have the option to this during my work. However, using results

from previous runs had proven to be very useful, especially when I had to restart the optimization loop. As I was able to reread some of the best results, the overall progress loss was minimal. Writing input lines into the 'readme.fgs' is also useful. If the user thinks that a specific genome would perform well in the simulation, but the algorithm has not tested it yet, the user can force the program to evaluate the individual. Alternatively, if an individual performed well during a slightly different simulation, it can be tested in the new scenario as well.

Overall GA_procq is a powerful tool, mainly because of its flexibility to be used with any other program. The user can track the progress easily and has ways to interact with the running optimizer. The ability to take and evaluate user-created genomes and reread previous results is also really beneficial in practice.

6.3 Simplified model

As mentioned earlier, my optimization task requires about 2000-3000 evaluation with this method. This amount ensures that the algorithm has time to find and refine the optimum. As I only had a single PC to do these evaluations, I had to reduce the evaluation time as much as possible. For this reason, I have created a simplified geometry to optimize.

This geometry is based on the one I was optimizing by hand. The height and width are similar, the angle of the walls was changed to make the geometry symmetrical. The length was calculated with Formula (3.2) described earlier in the thesis.

$$l_{min} = \frac{\frac{1}{2} \cdot m \cdot v^2}{F} = \frac{0.5 \cdot 1450kg \cdot 4.4444\frac{m}{s}}{120000N} = 0.119m. \tag{6.1}$$

This formula assumes that the crashbox is the only part absorbing the energy. This is not entirely true in reality, but neglecting the other parts promotes a safer design. As some room must be given for the deforming layers, I have rounded up the calculated length to 150mm. The two crashbox halves were connected with shared nodes. This guaranteed that the weldings simulated by PLINKs would not disconnect when the geometry is modified. The material thickness on both sides was 2.0mm. When the rectangular base shape was done, I added the front side of the geometry connecting the crashboxes to the horizontal bar.

I have removed every other part other than the crashboxes themselves, except the horizontal bar between the two. This bar had a simple profile and straight shape. As the exact deformation of this part is not important for me, I have increased the element size on it to further reduce the time required for a simulation.

My goal was to keep the simulation time under 3 minutes because that means almost 500 simulations per day. So if the program is left running for 5 days, it can finish almost 2500 simulation, which is enough to provide good results. Overall this goal was achieved, the average simulation time was a bit less than 2.5 minutes.

I have made a model with 10mm edge length as well, to test the algorithm with an

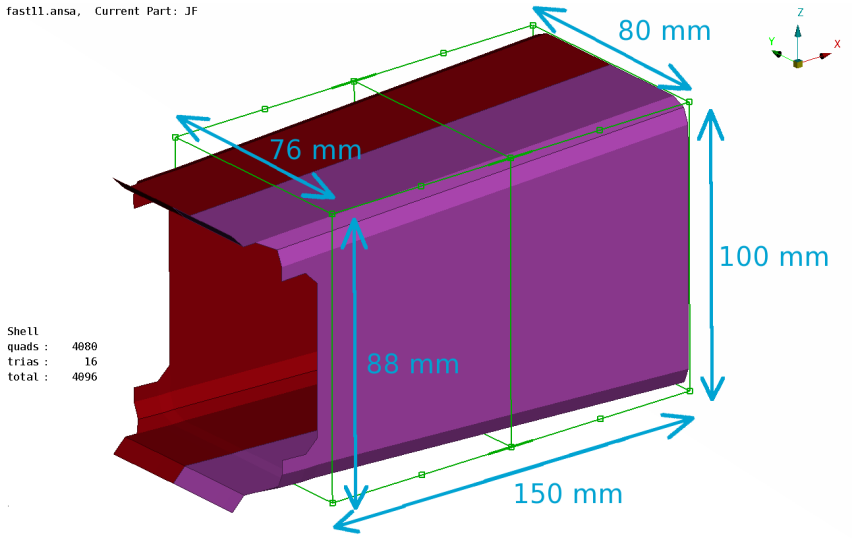

Figure 6.3. Dimensions of the simplified model

even faster simulation. The simulation time was decreased to about 60 seconds. However, the 10mm mesh was not fine enough. When the same geometry was using a 5mm mesh, the results became dramatically different. For this reason, the 10mm mesh was only used during the very early stages of development to prove that the method works. The results obtained from those runs were not usable during any other runs.

The time step length for the explicit solver was increased as much as possible to reduce the amount of calculation needed. To maintain stability, Pam-Crash uses the formula (6.2), where $\Delta t_{elemental}$ is the maximum stable time step, l_c is the characteristic length of the element, ϱ is the material density, and E is the elastic modulus.

$$\Delta t_{elemental} = l_c \cdot \sqrt{\frac{\varrho}{E}} \tag{6.2}$$

If the user chooses too long time steps, the program has to modify parts of the equation to keep stability. The characteristic length is defined by the mesh, so Pam-Crash has to either decrease the elastic modulus or increase the material density for the unstable elements. Pam-Crash uses the latter method., which is named "mass scaling". If the chosen time step is too big, it will add mass to the unstable elements. As the shortest edge length was 3.5mm instead of 5mm, I set my time step to $\Delta t_{elemental} = 0.0035 \cdot \sqrt{\frac{7850}{210 \cdot 10^9}} \approx 0.65 \cdot 10^{-3} ms$. With this setting, the added mass from mass scaling is 0.01 % of the mass of the crashbox. This added mass is not going to influence the simulation in any measurable amount. The simulated time interval was 85ms long, by this time each variant has finished the deformation process. (Source: [22], [23])

6.3.1 The barrier

The angle of the barrier according to Figure 3.4 is 10°. However, this was changed to 0° for the simplified model. The reason behind this change came from the initial test runs

made with the automated optimizer. The optimizer sometimes 'angled' the left side of the crashbox to match the angle of the barrier. These variants might be well-made for this specific barrier angle, but they would perform poorly when put into different scenarios. For this reason, the barrier angle was changed to 0° to get better overall designs.

6.3.2 Parametrization

It would be really hard to replicate the method of the manual optimization by a fixed number of parameters. For example, a singe push-in would require at least 2 parameters for positioning, 2 for sizing and 1 for height. Intotal, this would be minimum of 5 parameters per push-in, which would result in too many parameters.

For this reason, I have chosen another approach. Instead of making relatively small changes in the geometry, I picked parameters to change the shape more drastically. I have decided not to allow modifications on the very front section of the crashbox. Changing that part of the geometry could result in penetrating the elements of the horizontal bar, which would crash the simulation. Penetration would be hard to fix for my program, so I have decided to avoid this issue entirely by not modifying the front side of the geometry.

After considering the options to modify the geometry - which I will write about in the next section - I have chosen the method for creating the parameter set. I defined cross-sections in the geometry and used 4 parameters per defined cross-section. One for moving the top of the cross-section up, one for moving the bottom of the cross-section down, and two more following this method on the left and right side. I can choose how many of these cross-sections I want to include. The rest of the geometry between these cross-sections is moved according to these control cross-sections to create a smooth surface.

Additionally, I chose two more to modify the material thickness per crashbox half. Overall these principles allowed me to create a parameter set with a flexible number of parameters.

These parameters are also some of the most influential ones considering their effect on the behavior. During the manual optimization changing the material thickness was one of the most effective ways of influencing the overall behavior. The dimensions of the rear cross-section also have a great effect on the deformation process, as they control the angle of the crashbox's walls. Resizing the middle cross-section, especially along the Z axis also had a major impact on the behavior. The additional cross-sections were mostly used to fine-tune the design.

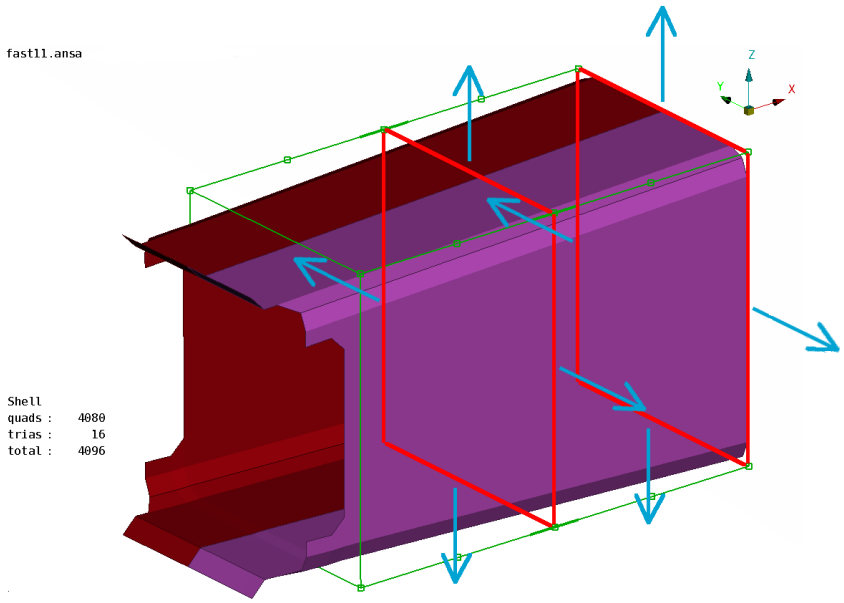

Figure 6.4. A possible set of 8 genes marked by blue arrows

6.4 Creating the new individuals

To make the parameters behave the way I described them previously I have used Ansa's morphing feature. Modifying the CAD geometry is possible with morphing, but re-meshing it automatically would most likely create a distorted mesh. For this reason I created a mesh manually on the base geometry, then modified the mesh directly with morphing.

Morphing allows the user to define bounding boxes called "Morphboxes" around the parts he wants to modify. When the shape of the Morphbox is modified, everything included is deformed accordingly. The user can move either faces, edges or nodes of the Morphbox. These nodes are called "Morphpoints". A Morphbox can be split into two smaller Morphboxes which have 4 shared Morphpoints. When one of these shared nodes is moved, the included elements will form a continuous smooth surface.

I have created a large Morphbox, with includes the entire crashbox, except the very front part. Then I have split the Morphbox in half perpendicular to the X axis. I have picked the rear and middle Morphpoints to be two of the control cross-sections. Then I have defined 4-4 additional Morphpoints on the two smaller Morphboxes to create two more control cross-sections. In total 4 cross-sections were defined with Morphpoints for 16 parameters to modify the geometry with. During most optimization runs, I had these 16 parameters and 2 for materian thickness, so 18 in total. In some of the early runs I have fixed the value of some parameters to zero, thus practically reducing the parameter count.

Morphing is also beneficial because it is easy to use from a script running in Ansa. As Ansa is restarted with every evaluation cycle, I created a file named 'ids.txt' containing

the IDs of the Morphpoints which the script has to move. This method ensures that during any evaluation the same Morphpoints are moved in the same order. I wrote an auxiliary Python script named 'picker.py' which I ran in Ansa to help me creating the 'ids.txt' file with the correct syntax. It is not required to use this script, but it makes the process considerably faster.

As I wrote earlier, GA_procq creates a file named 'in.fg' with the numerical values of the parameters describing the new individual. The first two parameters were controlling the material thickness; then the geometry parameters were listed in a specified order. When Ansa has started to create the new individual, it ran a script named 'morph.py'. This script moved the Morphpoints listed in 'ids.txt' by the magnitudes specified in 'in.fg'. Then the material thickness was modified, and the new variant was exported into the Pam-Crash include file named 'cmsvar01.inc'. After this Ansa was no longer needed, so it was shut down.

6.5 Automated evaluation

When the include file of the new variant is ready, the simulation must be run in Pam-Crash. This creates a result file with '.erfh5' extension containing both the deformation of the geometry and the "Transmitted force-Time" curve. Unfortunately, these cannot be exported directly by my script. For this reason, I must use Animator 4 to read the result file and export the curves into a file format which my script can use. I combine the "Displacement-Time" and "Transmitted force-Time" curve into the required "Displacement-Force" curve in Animator 4. Then this curve is exported into 'dipsforc.csv'. This file format can be handled by my script directly. After this Animator 4 is shut down. I have created a 'session file' for Animator 4 to do these steps automatically.

Once the "Displacement-Force" curve is exported, my script can calculate the objective function. For this purpose, I am using the difference between the ideal and actual behavior and give "penalty points" proportional to the difference. More specifically I am summing up the quadratic difference per curve point between the ideal and actual behavior. The penalty points p_n given for the nth point of the curve can be described by the following formula:

$$p_n = (F_n - F_{limit})^2 \cdot (x_{n+1} - x_n) \tag{6.3}$$

where F_n is the force value of the nth curve point, x_n and x_{n+1} are the displacement value of the nth and $(n+1)$th curve points respectively. Additionally, if $F_n > F_{limit}$ then p_n is multiplied by 2, meaning that values exceeding the force limit are penalized more heavily. The object function value or total score is given by the following sum:

$$score = \sum_{n=0}^{k} p_n \tag{6.4}$$

where k is the first curve point, when $x_k > x_{k+1}$. This condition is met once the crashbox's

load starts to decrease, indicating the end of the deformation and the start of the phase where the car 'bounces back' from the barrier. The calculation of p_n is shown in Figure 6.5, where $dF = F_n - F_{limit}$ and $dx = x_{n+1} - x_n$.

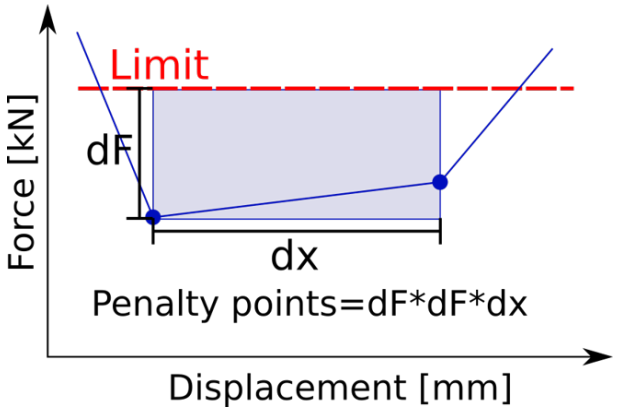

Figure 6.5. Penalty points of a curve point

The more closely the individual's behavior follows the ideal, the fewer penalty points it gets. If a variant were to follow the described ideal behavior perfectly, it would get a total score of 0. Going over the force limit is allowed just like during the manual optimization but penalized harder. As these are penalty points, the objective function's value needs to be minimized. As GA_procq is made to maximize the object function, every score value is multiplied by -1 inside the program. However, I am going to use the positive values when writing about the results because I found them to be more intuitive.

As a point of reference, if this scoring is used on the base geometry of the manual optimization, it gives 78482.9 penalty points. The manually made best variant got 12642.8. Due to the extra penalty points when the force is above the limit it is not possible to tell how much closer the curve runs to the ideal in the second case. A very rough estimation is that on average the optimized behavior is $\sqrt{\frac{78482}{12642}} \approx 2.5$ times closer to the ideal one.

6.6 Optimization loop with Python

As described earlier in detail, the parametrization of the model is done by defining Morphboxes around the mesh. An auxiliary script helps to pick the correct Morphpoint groups. The material thickness of the two halves gives two additional parameters.

I created the optimization loop with the help of Python scripting. When the preparations are done, and GA_procq is started, it generates a new 'in.fg' file then calls for evaluation. The evaluation is controlled by a Python script named 'host.py'. It is to be noted that all of the used commercial programs have special licensing. For me, this meant that using Ansa and Animator 4 costs the company on a time basis, so I had to minimize their runtime. Pam-Crash needs the most time during an evaluation cycle, but fortunately, I was able to use it without extra costs.

As a first step in the evaluation, Ansa is started by 'host.py' loading the mesh immediately and executing 'morph.py' inside Ansa. This script creates the new variant; then Ansa is shut down. After this Pam-Crash is started to run the simulation, which takes about 2-3 minutes. After Pam-Crash is done, the result file is read and processed by Animator 4 as described previously. Both Ansa and Animator 4 are run in batch mode. Batch mode means that no user interface is shown during these runs. This is beneficial for the time needed to execute the commands, as no resources are wasted to create a window, render the interface, etc. Furthermore, the user can work on something else without these windows popping up and closing in front of them.

Once the curve is exported, 'host.py' calculates the total score of the variant and a new 'out.fg' is created. If the evaluated individual is the best one so far, then the result file is stored in a folder, in case the user wants to view it later. The program only saves a configurable amount of result files to reduce the hard-drive space required. If the limit is reached, the oldest file is deleted. I usually kept the result file of the 3 best variants.

After some logging 'host.py' finishes, which is detected by GA_procq. Finally, GA_procq reads the 'out.fg' file, saves the results in 'allresults.fgs' and 'best.fgs' if the current individual is the best one so far. Calculations are done to create the new 'in.fg' file, and the loop goes back to the start.

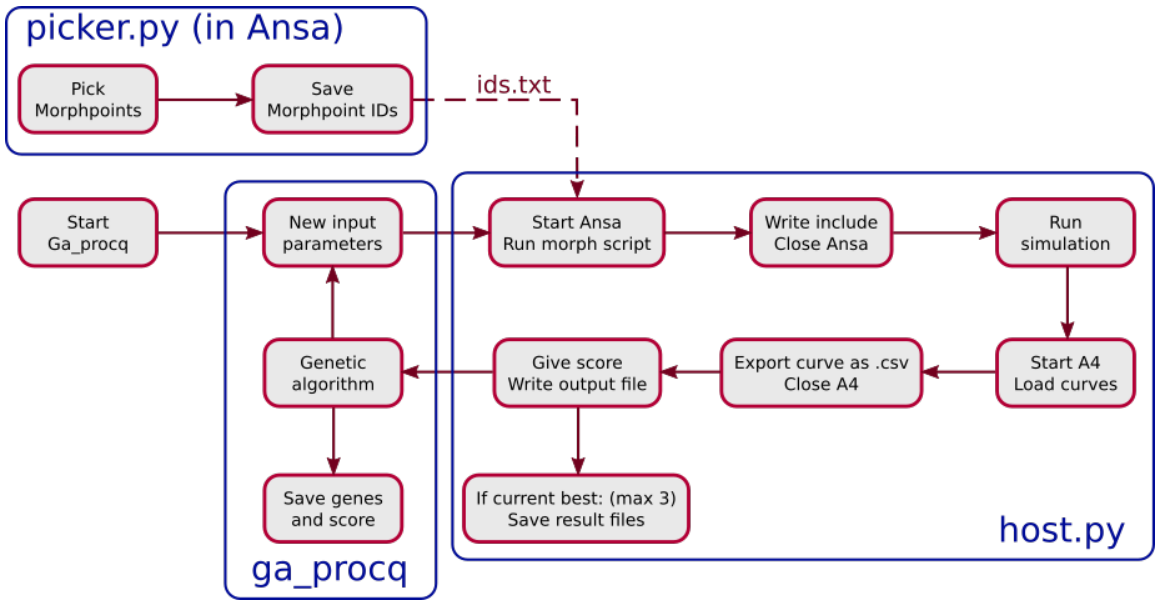

Figure 6.6. Flowchart of the created optimization loop

6.6.1 Handling errors in the loop

Error handling is also crucial in automated loops. It has to be done without any user input, and it should not be too time-consuming. My loop has to be prepared mostly for potential issues with the 3 external programs. For example, all of these require a license server connection, so in case of an Internet failure, none of them start. During my work

with these software, I found out where they might have issues.

Ansa is prone to license connection issues, but this very rarely happens. In case it does, Ansa will try to connect to a license server every 15 seconds. This is achieved by a command line argument when starting Ansa from the Python script. No errors were encountered when running the morphing script.

Pam-Crash did not have any issues when the model created by Ansa was valid. During the early stages of development sometimes the material thickness was set to 0, which caused a Pam-Crash error. In this case, Pam-Crash failed to start the simulation, and the old result file stayed in the folder.

Animator 4 was the most affected by license server issues, even with a stable Internet connection. When it failed to connect, it did not run the curve conversion. This meant that the previously exported curve stayed in the folder. Of course, when this curve was given its score, it was the same score as the last one. I used this to detect this error, and this method also works in case of a Pam-Crash failure. For this reason, I always stored the last score in a separate file for easy access.

If the current calculated score was equal to the last one, the program detected the potential error. This started a cycle, where Animator 4 is called over and over again with a 5-second delay maximum 5 times. The cycle stopped if the recalculated score was different from the last one. If the original issue was due to license issue, it would most likely not last for 5 additional attempts. Usually, it only took maximum 2 additional tries to get a license server connection.

If the issue was not originated from Animator 4, but Pam-Crash, all 5 extra evaluations gave the exact same score. If this happened the run was considered invalid and was given a very bad score. As I wrote earlier, giving a good variant bad score will not have a significant effect on the long run. However, giving a bad variant good score is much more dangerous as it will continue to 'poison' the population. As GA_procq forgets the bad individuals over time, if this genome is suspected to be good it might be rechecked.

It is to be noted that there is some randomness in the Pam-Crash explicit solver. If the same simulation were run multiple times, the resulting score would be very slightly different. The difference in score is $\approx \pm 0.1$, which is practically negligible because it is 4-5 magnitudes lower than the score itself.

6.6.2 Time allocation

Analyzing the time required for each program can help to identify potential bottlenecks. My loop takes about 2-3 minutes to finish. Usually, the elapsed time between the command to start Ansa and Ansa shutting down is 10 seconds but sometimes goes up to 15. Unfortunately, most of this is waiting for the connection with the license server. The lowest elapsed time was 3 seconds for Ansa, so completing the commands themselves probably takes about 2 seconds.

Pam-Crash needs the most time from all the programs. Usually, it takes 150-220

seconds to finish the simulation. If the user is doing something taxing on the CPU, this time can increase significantly.

From the command to start Animator 4 to it shutting down the elapsed time is usually 2 seconds. Any other process, including calculating the score, file operations like logging and creating the new 'out.fg' and generating the new genome is significantly less than 1 second.

This gives us the conclusion that Pam-Crash needs the most time. It could only be reduced significantly by using more powerful hardware. However, the time required to establish the connection with the Ansa license servers does not depend on the user, and it creates a small but not negligible delay in the runs.

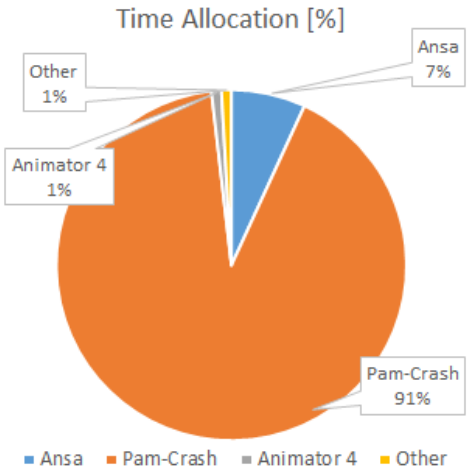

Figure 6.7. Time allocation of a typical iteration

6.6.3 A short overview of the runs

During my period of working on this subject, I made several runs. At first, I made rather short (1-2 days) runs compared to the later ones. These were made to test the optimization loop and the method in general. Later I made longer, 6-day runs. The list of all runs during my work can be seen below.

1. Run 1: First test run, done on the 10mm mesh, 10 parameters. Ran for about 23 hours. Ansa runs in batch mode.

2. Run 2: Second test run, done on the 10mm mesh, 10 parameters Ran for about 48 hours. Animator 4 runs in batch mode.

3. Run 3: Run on the new 5mm mesh, 14 parameters. Ran for about 130 hours. The loop now saves the result files of the best 3 variants.

4. Run 4: Attempt to continue Run 3. Stopped after 23 hours because of a bug in the program.

5. Run 5: Run on the 5mm mesh, 18 parameters. Ran for about 90 hours.

6. Run 6: The bug what occurred during Run 4 is fixed. Run 5 restarted with the new program. Ran for 26 hours.

7. Run 7: Run 6 restarted after a syntax error during user input. Run 7 got some good genomes from the results of Run 3. Ran for about 130 hours.

8. Run 8: Final run, on the same mesh as the manual optimization, providing the base for direct comparison of the two methods. Ran for about 130 hours.

All runs were done on a single computer.
I picked only some of these runs to present in detail.

Chapter 7

Results in detail

The parameter limits were from 1.5 to 3.0 for material thickness. The range of the morphing parameters was typically from -20mm to +20mm.

7.1 Run 3

Run 3 was the first run, where the 5mm mesh was used. 5mm is the standard edge length in the industry to simulate metal parts. This simulation also used the 0° version of the barrier. The total number of individuals created and tested is 3266. The 14 parameters were: 2 material thickness, 4+4 parameters from the middle and rear control cross-section and 2+2 parameters from additional 2 control cross-sections, where the Z direction movements were not used as parameters.

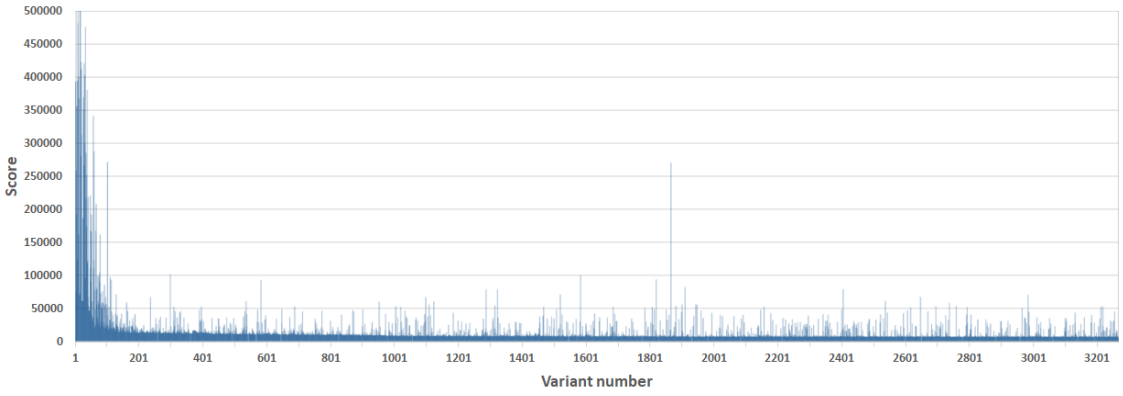

Figure 7.1. Penalty points of the variants in Run 3

In Figure 7.1 we can see the total score (the sum of penalty points) for each variant. At the start of the run, genomes are generated randomly, so there are a lot of bad variants in the first 100. The general trend of a genetic algorithm can be observed. First, sudden improvement after the initial bad genomes. Then population keeps improving over time. The random behavior can be observed on the high 'spikes': as the algorithm mutates, crosses, etc. the genomes, many are created, which are significantly worse than their

predecessors.

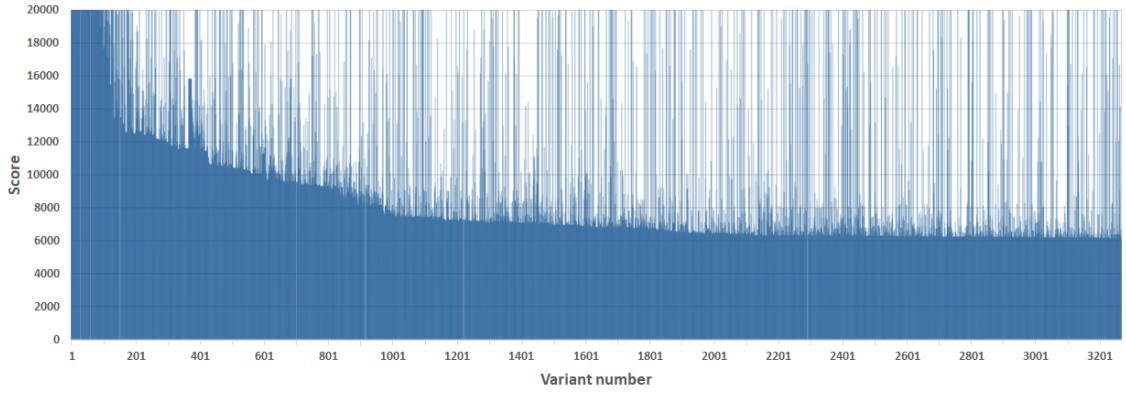

Figure 7.2. Penalty points of the variants in Run 3 - zoomed in

Figure 7.2 shows the same graph, but with zooming in along the Y axis, so the lower values can be seen in greater detail. It can be observed, that almost all of the first 200 variants got more than 20000 penalty points. Then there is a period from 200 to 1000 where the individuals still improve quickly. From 1000 to 2000 the population keeps progressing, but significantly slower. Finally, after 2000 variants the new improvements are minimal.

As mentioned previously, the hand-made best variant got roughly 12500 penalty points. This run uses a different model, but we can safely say, that any variant scoring below 10000 is quite good. As we can see, there are hundreds of variants in this category. The final score achieved by Run 3 is 6129.2. The material thickness of the optimized shape is 1.92mm and 3.0mm.

This result is not directly comparable to the one made by hand, as it is achieved on a different model. For example one of the main issues during the manual optimization was the end of the vertical bar causing self-collision (shown in Figure 5.3). This simplified model did not have such parts. However, this run still demonstrates the potential of the method.

7.1.1 Evolution steps of Run 3

To demonstrate how the algorithm improves the geometry, I picked some "snapshots" of the evolution of the geometry. Every variant shown was once the best in the population. In Figure 7.4 the variants improve from left to right, and each row has better ones than the previous row. The bottom-right geometry is the overall best. As we can see, the first two are significantly different from the rest, the second one being especially different from the third.

These variants were picked based on their "distance" in the parameter space. First, all genes were normalized by their parameter range. So if a parameter was equal to the lower limit, it's normalized values was 0, and if it was equal to the upper limit, the normalized

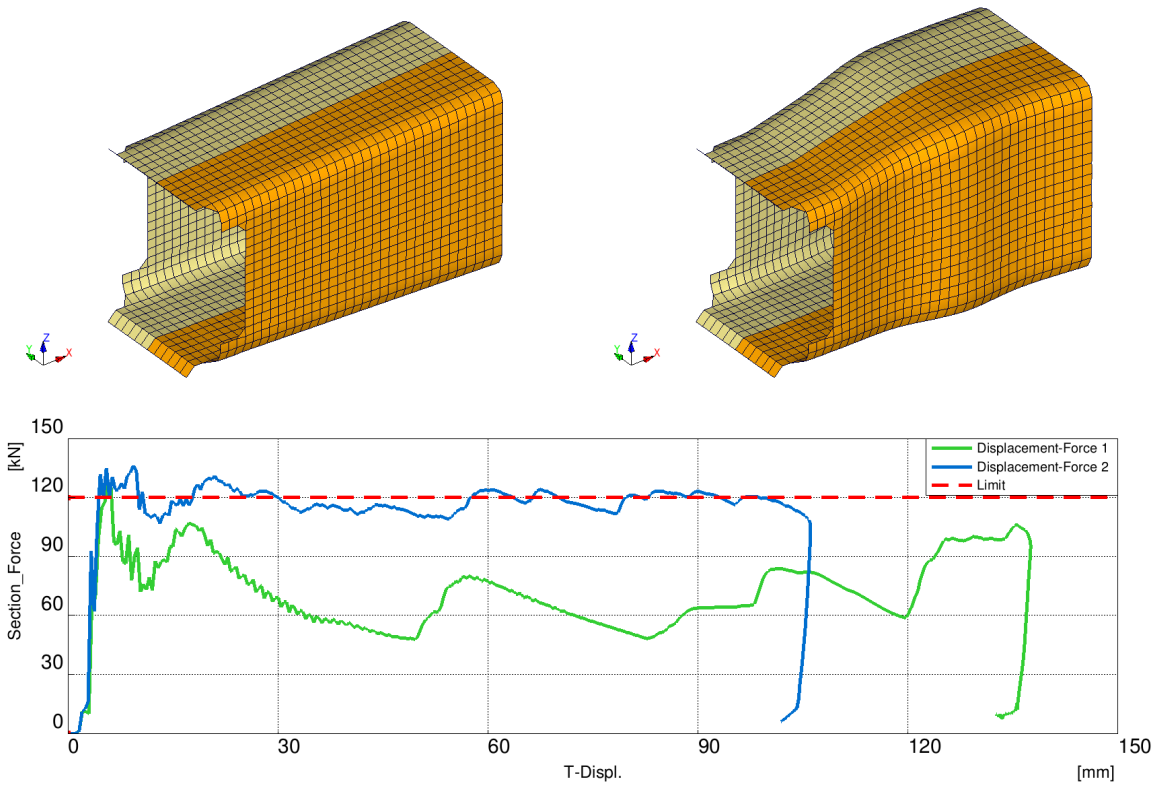

Figure 7.3. The base shape and the result shape of Run 3

value was 1. Then the distance was calculated by the common formula (7.1), where $d_{a,b}$ is the distance between variant a and b. x_k it the difference between the two variants kth parameter's normalized value.

$$d_{a,b} = \sqrt{\sum_{k=0}^{N}(x_k)^2} \tag{7.1}$$

The distance of the nth best and $(n+1)$th best value was calculated for all n values. The highest values indicated the most drastic changes in design to make an improvement. In Figure 7.4 we can see the 5 most drastic changes and the final variant.

The "Displacement-Force" curves of these variants can be found in Appendix B.

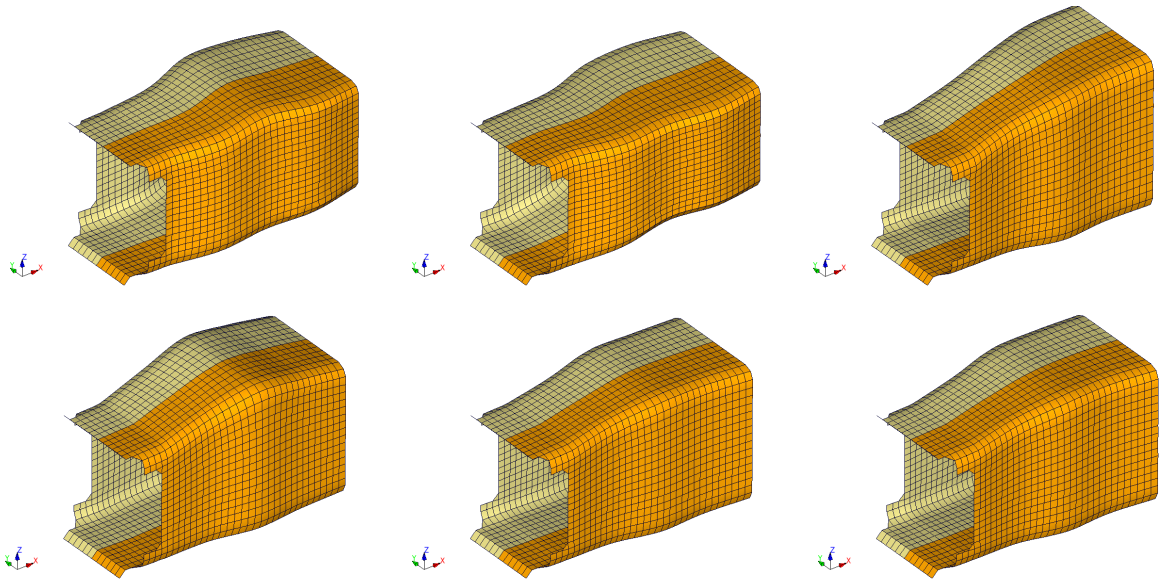

Figure 7.4. Some samples showing the evolution of Run 3

7.2 Run 5-6-7

As Run 6 and 7 are both restarts of the previous one, I am presenting their results together. Every setting was the same as in Run 3, the only difference being the number of parameters. This run used 18 parameters, 2 for material thickness and 4*4 geometry parameters from 4 control-cross sections. If the last 4 parameters had their values fixed to 0, then the program would have the same parameter set as it had in Run 3. The combined time of the runs was about 250 hours, and over 6200 variants were evaluated. The final score achieved by these runs is 4492.8.

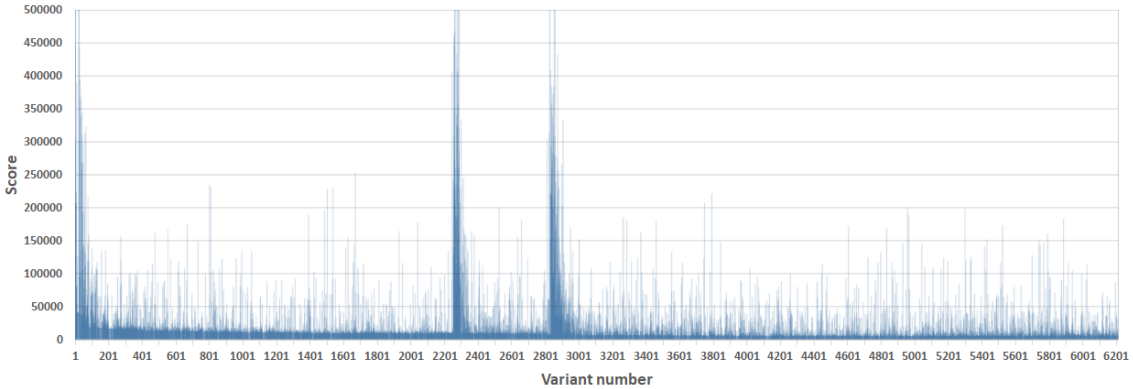

Figure 7.5. Penalty points of the variants in Run 5-6-7

This high number of individuals makes them hard to visualize on a plot (over 6200 vertical lines would be needed...), the lighter colors in Figure 7.5 are a result of that.

The restarts are clearly visible in Figure 7.5 as the tallest spikes, but the scores go back to their previous level soon after the restarts. This occurred because I chose not

to give the optimizer all of the earlier results. Instead, when the restart happened, and GA_procq started creating a new population, I gave about 5 of the best previous results as user input in a 'readme.fgs'. I did not necessarily use the best 5, instead, I used 5 which were reasonably good, but relatively far apart in the parameter space. So the new population started with some already good individuals in it, which helped the improvements drastically, but the variety was maintained by not using too many previous genomes.

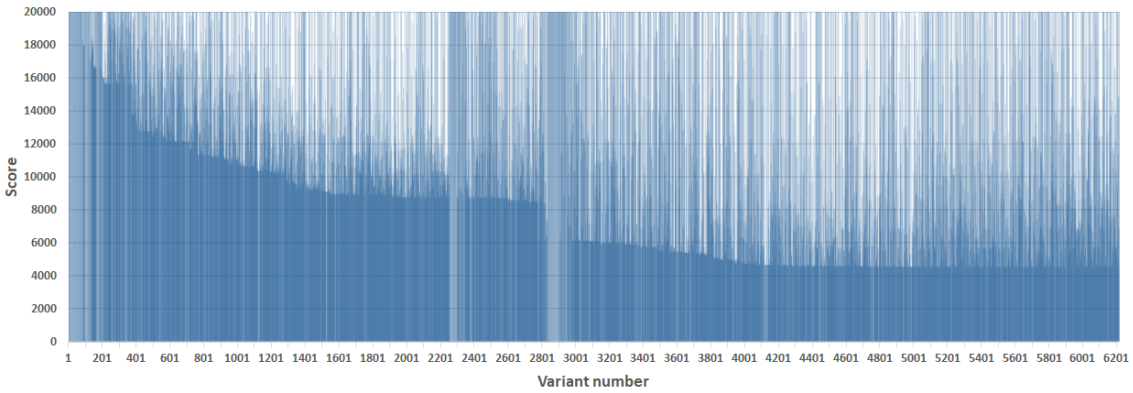

Figure 7.6. Penalty points of the variants in Run 5-6-7 - zoomed in

The trend we can observe in Figure 7.6 is similar to the one observed on Run 3. After a period of fast improvement up to about 1500 variants, progress is slowed down considerably. The next 'jump' is the inclusion of some good variants from Run 3 near the second restart. After that, the score goes down from roughly 6000 to 4500 in 1300 iterations. After about the 5000th iteration no significant improvements are made, and there is an increased number of 'bad' variants. This is the indication that the optimizer has come very close to the optimum, further adjustments are much more likely to have a negative impact.

The fact that inserting some better elements from Run 3 caused such a significant improvement is quite interesting. It can be explained by the parameter space now having 4 additional dimensions. It appears that the algorithm has missed the optimum in the extended parameter space. This is unfortunately always a possibility with genetic algorithms: finding the global optimum is not guaranteed. The chance of missing the optimum can be reduced by using larger populations and better-spread initial individuals in the parameter space. Genetic algorithms always find 'decent' solutions, however. As we are going to see later, the difference between a variant with 6000 or 4500 is not major in behavior.

We can see the final shape compared to the base shape in Figure 7.7.

An other interesting comparison can be made between the optimized shape of Run 3 and 5-6-7.

As we can see in Figure 7.8 the difference in the geometry is hardly noticeable. The material thickness is also unchanged. The curve of the second variant follows the ideal slightly better, but not by much. The deformation length is shorter on the second variant

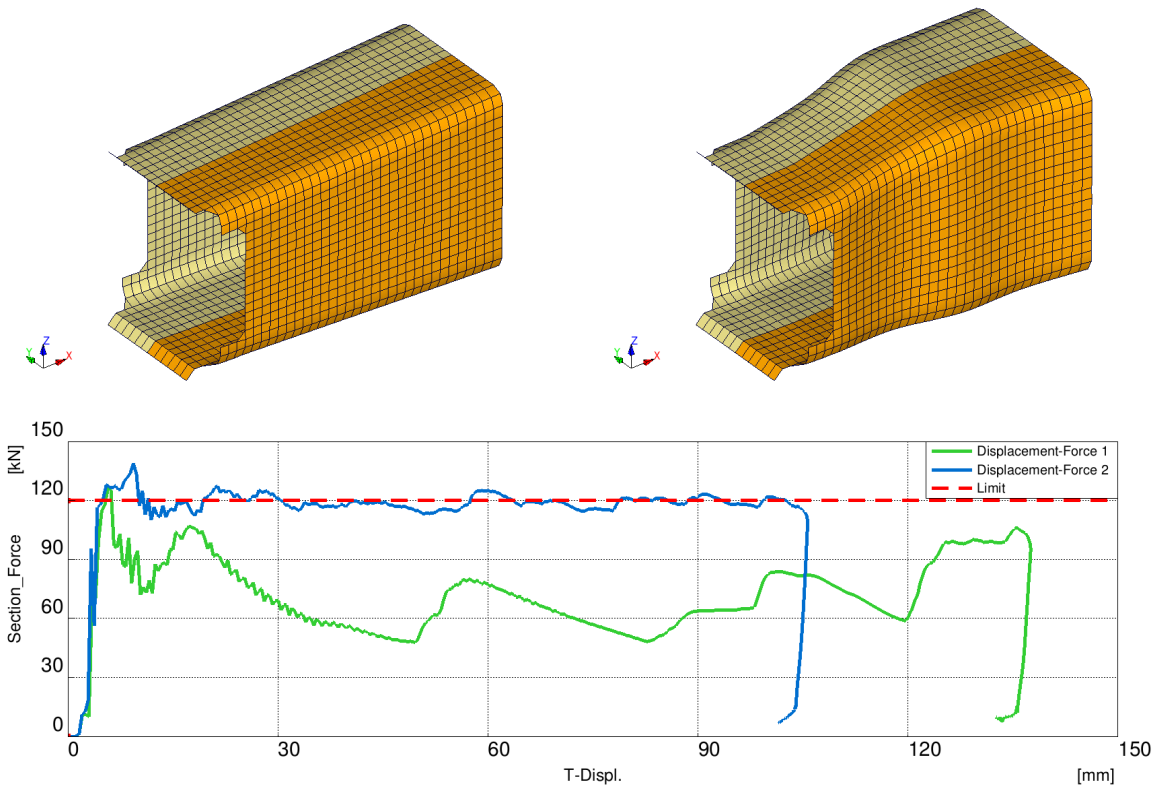

Figure 7.7. The base shape and the result shape of Run 5-6-7

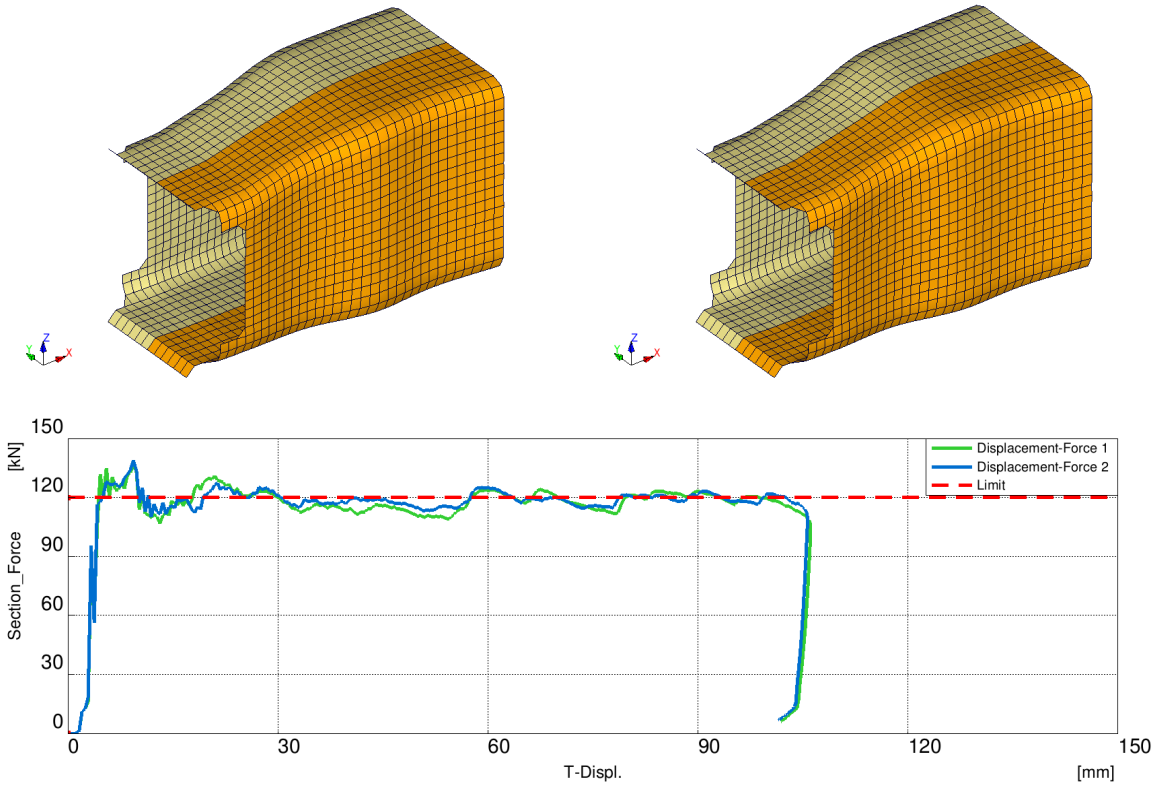

Figure 7.8. The optimized shape of Run 3 and 5-6-7

as well, but the difference is negligible. We can conclude, that the final shape of Run 3 was an already excellent variant, so major improvements were not possible.

7.2.1 Local optima in Run 5-6-7

As I have manually inserted some good genomes from Run 3, the evolution steps became distorted. Instead, I have picked some local maximums from all evaluated variants. To select these individuals, I have calculated the distance between each variant pair, using the method described in subsection 7.1.1. This gave me the distance matrix $\mathbf{D}$, where $d_{j,k}$ is the distance of the jth individual from the kth individual measured in the parameter space.

$$\mathbf{D} = \begin{bmatrix} d_{1,1} & d_{1,2} & d_{1,3} & \dots & d_{1,N} \\ d_{2,1} & d_{2,2} & d_{2,3} & \dots & d_{2,N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ d_{N,1} & d_{N,2} & d_{N,3} & \dots & d_{N,N} \end{bmatrix} \tag{7.2}$$

To determine, whether the jth variant is a local best, we take the jth row of $\mathbf{D}$. Then we select the n variants, which are closes to the jth variant. If the score achieved by the jth variant is better than all of the n closest variants' score, then it is considered to be a local optimum. It is important to note, that these local optima are not as refined as the global optimum, as GA_procq did not spend as much time improving them.

These local optima can have great practical value. If the found global optimum does not satisfy some property which was not programmed into the optimization loop, for example, manufacturability, then these local optima can provide help for the user. These results might contain different, but good enough variants, so further optimization might not be needed. If none of these local optima are good enough, they can be refined by the optimization loop, if the parameter limits are set up appropriately.

The "Displacement-Force" curve of the variants shown in Figure 7.9 can be found in Appendix C. These variants are as good as the final result, but they are still acceptable. They are also not as refined as the final optimum, so it is possible to make them better.

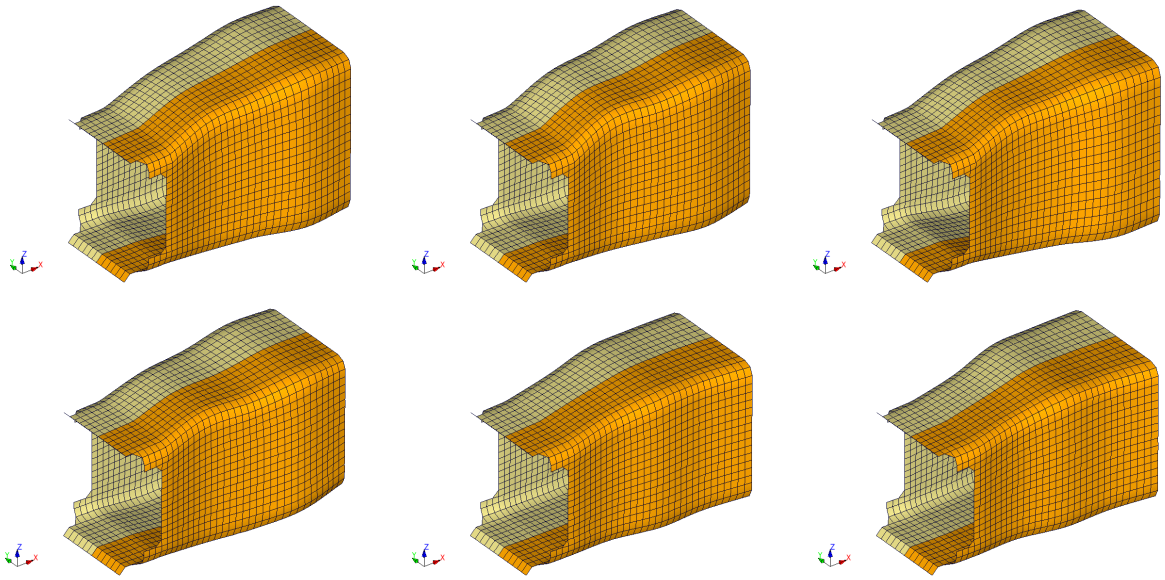

Figure 7.9. Some local optimums from Run 5-6-7

7.3 Run 8

The previous runs were made on a simplified geometry, so their results can not be compared directly with the result of the manual optimization. However, the simulation time during the manual optimization was about 10-14 minutes, which is too long for the optimization loop. (Evaluating 2000 variants with 10 minutes each would take about 330 hours.) For this reason, I made some changes in the model. I deleted some parts of the geometry, which were not playing an important role during the crash. These include the lower horizontal bar, the vertical bars connecting to it and some additional minor parts. The top of the vertical bars was not deleted, as they had an important role, as shown in Figure 5.3.

The element edge length was increased on those parts, where the deformation was not significant, for example, in the right side crashbox and most of the horizontal bar. The edge length was increased in steps: the zone close to the main deforming parts was given a 7mm target length, the zone located further away got a 10mm mesh, and some parts of the horizontal bar got a 15mm mesh. This way the element count was reduced from 26000 to 9900 and the runtime from 10-14 minutes to 3-5 minutes. After these changes I made some test runs, and the difference between the results gotten from the full 26000-element and the 9900-element model were negligible. Thus we can say, that the optimization loop worked with the same model as I did during the manual optimization.

In Run 8 the optimization loop was running for about 130 hours, creating over 2000 variants in total. The barrier angle was 10°, as this was used during the manual optimization as well. The tendencies in Figure 7.10 are similar to the previous runs.

When we zoom in along the Y axis, we can see a slightly different trend than before. After the period of fast improvements up to about 300 variants, the progress suddenly

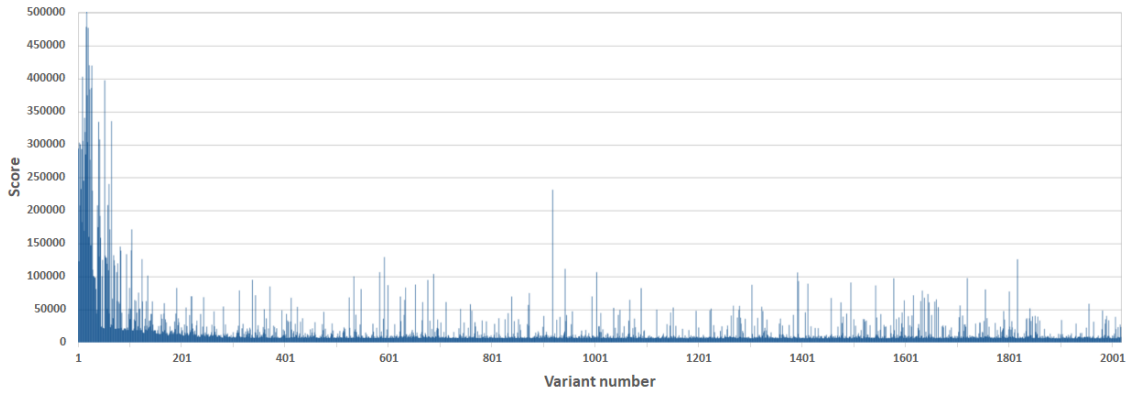

Figure 7.10. Penalty points of the variants in Run 8

slows down to a nearly constant pace. What is more interesting this time, is the scores themselves. The manual optimization's best achieved 12642.8 points. Run 8 has created considerably better variants, reaching the final score of 6264.5.

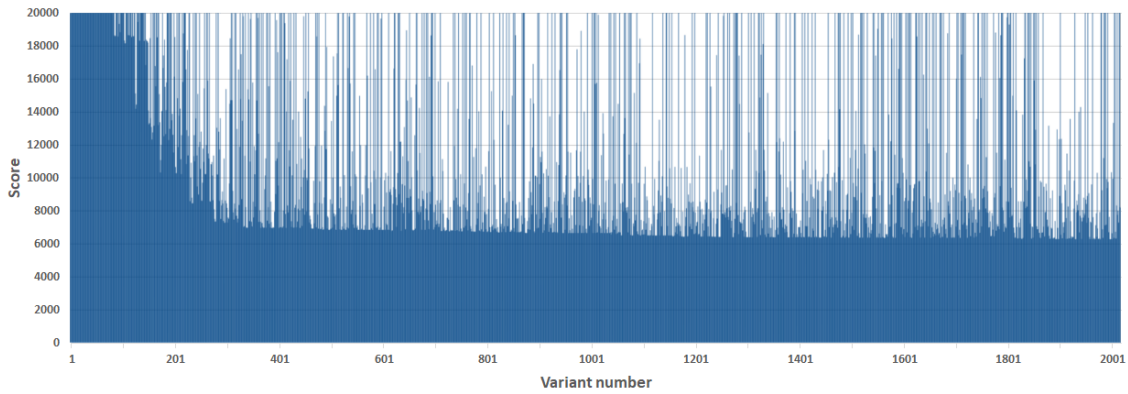

Figure 7.11. Penalty points of the variants in Run 8 - zoomed in

As we can see in Figure 7.11, the program only needed about 150-200 variants and 14 hours to beat the best variant I was able to make by hand. In Figure 7.12 we can see the base shape compared to the optimized shape. Its behavior follows the ideal almost perfectly until 40mm deformation. Between 40mm and 80mm, some fluctuation can be observed, but it is not significant. The force stays between 110kN and 127kN. Finally, after 80mm, it follows the ideal behavior very closely, except the very last part of the process, where some self-collision can be observed. The force never goes above 130kN.

The more important comparison is made between the final result of manual and automated optimization. The behavior of the program-made variant is significantly better. A very rough estimation is that on average this behavior is $\sqrt{\frac{12642}{6264}} \approx 1.42$ times closer to the ideal one. The deformation length is about 5mm shorter on the hand-made variant. The material thickness on the hand-made variant is 2.6mm, 2.8mm, the result of the automated optimization uses 2.82mm, 2.78mm thick sheets.

The final result of the optimization is also quite different in shape, that the hand-made

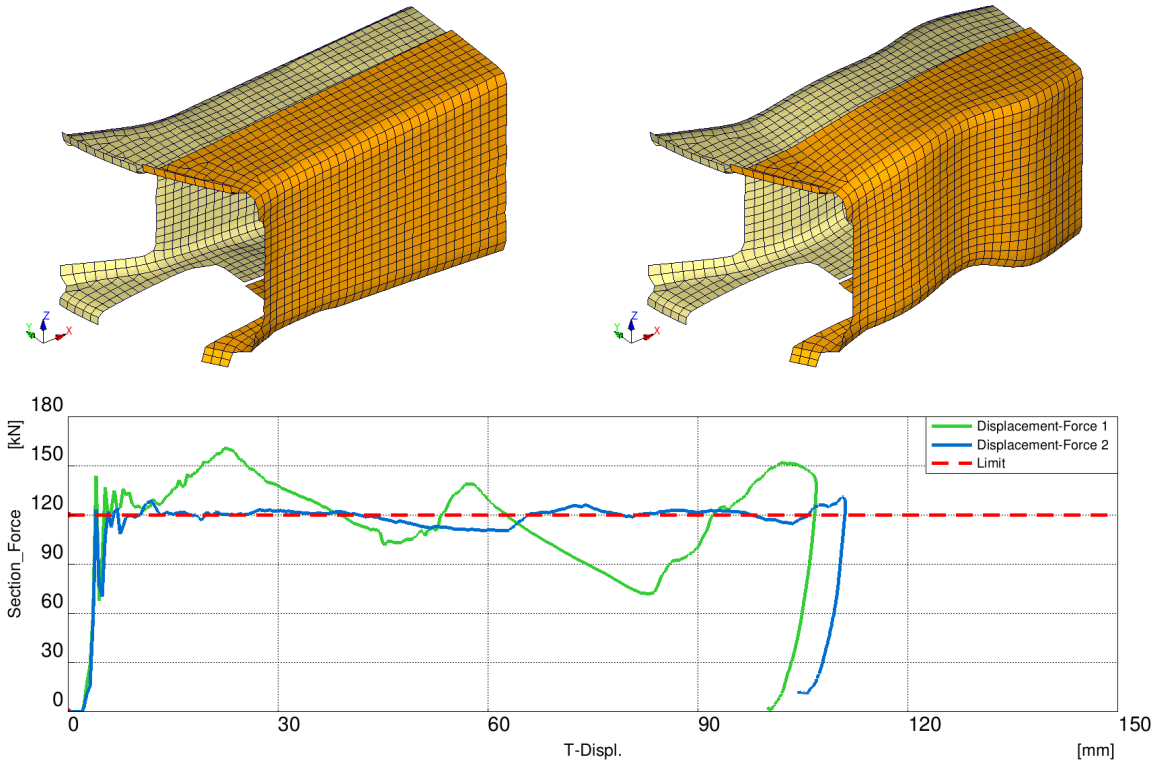

Figure 7.12. The optimized shape of Run 8

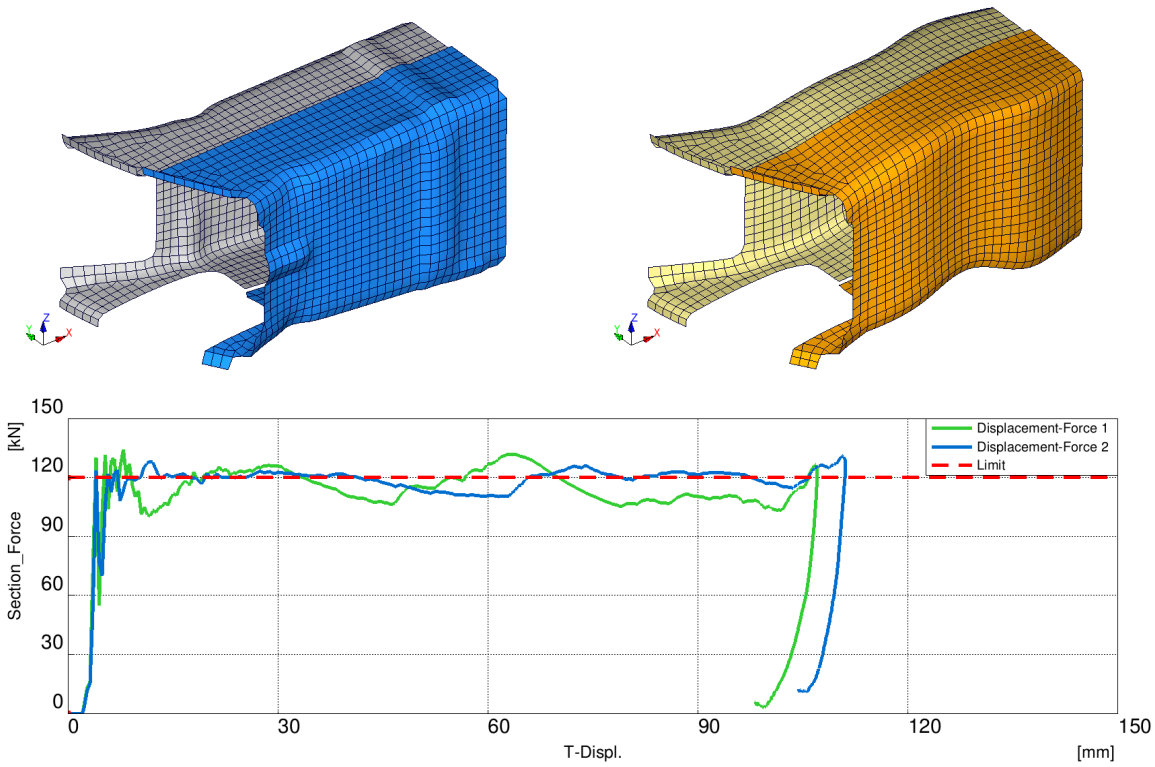

Figure 7.13. The manually optimized shape and the result of Run 8

design. There is a unique, deep push-in located on the bottom and that cross-section is also pushed in on both sides. For this reason, it behaves like a bottleneck during the crash. The rear cross-section is also narrower than most of the geometry. These solutions can be considered unconventional, but the optimization loop was able to fine-tune them to create significantly better variants than the manually optimized version.

7.3.1 Evolution steps of Run 8

Similarly to subsection (7.1.1), I picked some of the more important steps in the evolution of Run 8. In Figure 7.14 we can see, that most of these once best variants used the 'bottleneck' cross-section at the middle.

The "Displacement-Force" curves and penalty points can be found in Appendix D.

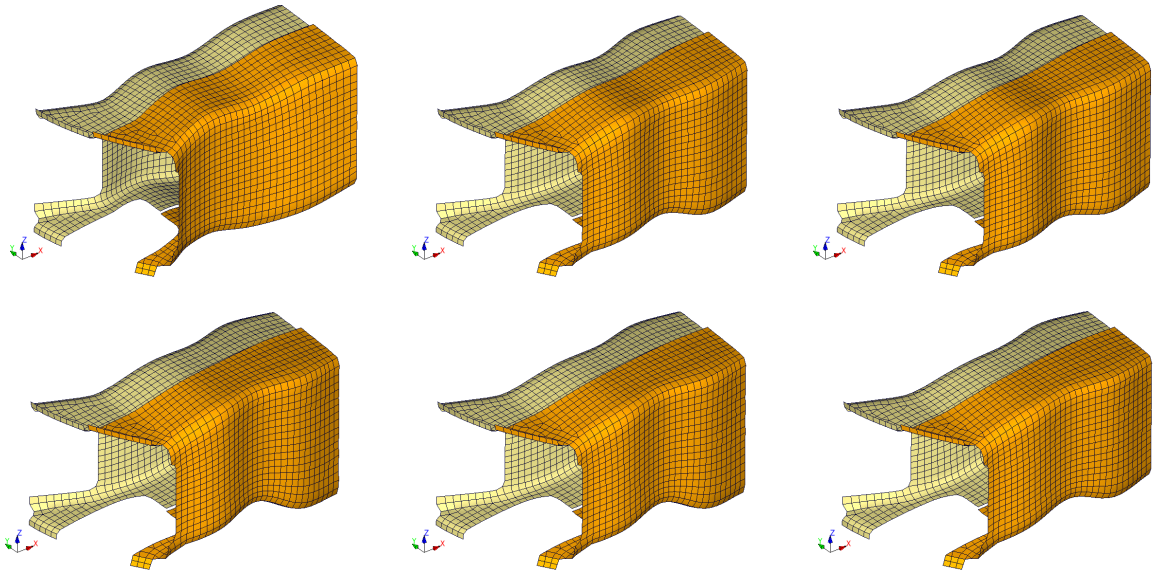

Figure 7.14. Some samples showing the evolution of Run 8

Chapter 8

Conclusion

8.1 Comparing the manual and automatic method

In this section, I will use the data from the manual optimization and Run 8. The comparison can be made considering several aspects. The automated method in general needs more simulations, than the manual process. Run 8 did over 2000 iterations in total, which provided really good results. However, to reach the score of achieved manually, it needed about 200 runs, roughly 2 times more than the manual process used.

The total time needed for the manual process was about 80 hours. The automated method was running for 130 hours. However, it could have been stopped at any time, especially when the progress has slowed down after 400 iterations. It only needed 14 hours to beat the best hand-made version. Due to the random nature of genetic algorithms, this time may vary, but Run 3 and 5-6-7 also went below 12600 score significantly faster, that 24 hours (Run 3 needed 7 hours, Run 5-6-7 needed 17). In practice, this means that if the program is started at the end of a workday and left running overnight, it can provide good results by the next morning.

Run 8 used 130 hours in total, but only 1-2 hours were needed to set it up. Once it was started, it required no human interaction at all. Naturally, all 80 hours used for the manual optimization were workhours. This means a major reduction in the human time needed for one optimization task, essentially reducing the number of workhours by over 95%.

During the manual optimization, I had complete freedom over the geometry, only following some general rules described in chapter (5). The program only had 18 parameters but managed to create a unique, but very fine-tuned solution.

Finally, the variant created by the program follows the ideal behavior significantly better. This is indicated by both the penalty points acquired and the "Displacement-Force" curves.

A less specific benefit of the automated method is in hard-drive usage. With manual optimization, the user has to store the file containing the geometry if they want to use or review it later. The "Displacement-Force" curve also has to be stored. With the automated

Table 8.1. Comparing the two methods

Aspect	Manual	Automated
Number of simulations	100	2000
Total time (hours)	80	130
Human workhours	80	1-2
Number of parameters	N/A	18
Result score	12642.8	6264.5

method, it is enough to store the 'allresults.fgs' file. It contains the parameters used to create the geometry, and the behavior is described by a single number. This method requires less hard-drive space, which can be important when thousands of iterations are made. For example, the 'allresults.fgs' file of Run 5-6-7 contains over 6200 variants, and it is only 1.2 MB. The 'cmsvar01.inc' file includes the mesh of the crashbox and related parts only, and it is almost 1.5 MB in size. (The '.csv' file containing the "Displacement-Force" curve is only 30 KB.) This means that storing the results requires over 5000 times less hard-drive space with the automated method.

8.2 Ways to improve the method

This automated method already provides good results, but it could be improved. It would be useful if the program adjusted the parameter limits when needed. At the moment the limits are read from 'minmax.dat' at the start. During some test runs some of the parameters reached the minimum/maximum value on the best variants. This suggests that allowing those parameters to have smaller/larger values would help to improve the design. When starting a new run it is hard to judge, where the limits should be. If the parameter ranges are too wide, the algorithm might take much longer to find a good solution. If the parameter ranges are too narrow, the optimum might be out these ranges. With flexible bounds, the program could change the limits in small increments as long as these changes help to create better variants. However, some limits must not be crossed due to other design reasons. For example, the H340 LAD steel sheets are manufactured up to 3mm thickness. The program should not create variants with 4mm thickness even if they behave better, as they can not be made out of the selected material. Implementing this feature would require adding a new file containing such limits.

Another improvement could be creating a more user-friendly interface. At the moment many options are only available by modifying the Python code. For example, adding or removing parameters requires the user to edit both the 'morph.py' script used to modify the geometry, 'picker.py' to create the 'ids.txt' file with the appropriate number of Morphpoint groups, etc. The parameters are also not in any logical order, so even if the new user knows Python scripting, the lack of structure makes understanding the code a lot more difficult.

GA_procq can be run on multiple PCs at the same time, but this ability was not used

during my work. If more machines would work on the same problem, they could share their results, making improvements faster. This method would also reduce the chance of missing the global optimum like it did initially in Run 5-6-7. Sharing results could be performed by an agent program, which shares the current best individuals calculated by each PC via a 'readme.fgs file.

The variants created by the optimization loop are currently optimized for one specific crash scenario. The behavior of the final variant is very good for this specific case. In practice, it is more useful if the variant performs well in more scenarios. This would probably cause the variant to behave slightly worse in each scenario individually, but overall it would be worth it for the flexibility. A simple way would be to run multiple simulations per variant, then take the average of their score. This would require more runs, so I did not attempt this during my work on the subject.

One more unused feature of GA_procq is multi-object optimization. A variant could get multiple numbers describing it's behavior instead of one. This way different type of properties could be considered, for example, weight. After the optimization is done, the user could pick the best variant based on either weight, behavior or most likely both.

Finally, physically building and testing one of these optimized shapes could validate the results. FEM software provide reliable results when used properly, but creating a prototype could point out hidden weaknesses of the design. It would also reveal some properties, which are not analyzable with FEM, for example, manufacturability.

8.3 Summary

The automated method quickly created better variants than the result of the manual optimization. However, this might require more simulation time than manual optimization would. It is capable of generating significantly better variants than it is possible by hand.

Naturally, after a certain time, improvements become slower and slower, but this only happened when the loop was creating really good variants. As described at the end of section (7.3), it may create unique solutions, but it is able to fine-tune them very well.

Overall the automated optimization seems much more effective than the manual way, especially if the required amount of human workhours in considered.

Chapter 9

References

9.1 Book, thesis

[1] Ted Belytschko, Wing Kam Liu, Brian Moran, Khalil Elkhodary: *Nonlinear Finite Elements for Continua and Structures*, John Wiley & Sons Ldt, 2001

[2] Marton, Árpád: *Crashbox optimalizálása alacsony sebességű ütközés esetén*, Széchenyi Istán University, Győr, 2014

[3] Stoyan, Gisbert; Takó, Galina: *Numerikus módszerek 1.*, Typotex, 2005

[4] Press, W. H.; Teukolsky, S. A.; Vetterling, W. T.; Flannery, B. P.: *Numerical recipes in C (Vol. 2)*, Cambridge university press, 1996

[5] Álmos, Attila; Győri, Sándor; Dr. Horváth, Gábor; Várkonyiné Kóczy, Annamária: *Genetikus algoritmusok*, Typotex, 2012

9.2 Publication

[6] Dr. Horváth, András; Dr. Horváth, Zoltán: *Optimal shape design of diesel intake ports with evolutionary algorithm*, Proceedings of 5th European conference on numerical mathematics and advanced applications (ENUMATH 2003), edited by Feistauer, M. et al., Springer Verlag, 2004

9.3 Website

[7] Wikipedia: *EDAG*, https://en.wikipedia.org/wiki/EDAG

[8] EDAG website: *EDAG - an Overview*, https://www.edag.de/en/edag/edag-an-overview.html

[9] Beta website: *Ansa pre-processor*, https://www.beta-cae.com/ansa.htm

[10] Wikipedia: *Pam-Crash*, https://en.wikipedia.org/wiki/Pam-Crash

[11] ESI Group website: *Eberhard Haug, Co-founder of ESI Group, speaks about the origins of Pam-Crash*, https://www.esi-group.com/pam-crash

[12] CDH AG website: *GNS Animator 4*, http://www.cdh-ag.com/en/software/gns-software/gns-animator-4.html

[13] Wikipedia: *Python (programming language)*, https://en.wikipedia.org/wiki/Python_(programming_language)

[14] Python Software Foundation website: *Applications for Python*, https://www.python.org/about/apps/

[15] Wikipedia: *Automobile safety*, https://en.wikipedia.org/wiki/Automobile_safety

[16] The Aluminium Automotive Manual: *Crash management systems*, https://www.european-aluminium.eu/media/1548/4_aam_crash-management-systems1.pdf

[17] Rohin Nagrani: *Hot Stamping and Automotive Safety*, https://www.linkedin.com/pulse/hot-stamping-automotive-safety-rohin-nagrani

[18] VETR - Vehicle Engineering Technical Review, http://www.invetr.com/uploads/2/1/8/2/21829402/7933738_orig.jpg

[19] RCAR - Procedures: *RCAR Low-speed structural crash test protocol, Issue 2.2*, http://rcar.org/Papers/Procedures/rcar_LowSpeedCrashTest2_2.pdf, 2011

[20] Shanghai Royal Industry: *H340 LAD*, http://www.sphc-steel.com/Cold-Rolled-Automotive-Steel-Sheets/H340LAD-Z.html

[21] DYNAmore: *Material Failure Approaches for Ultra High strength Steel*, 2006

9.4 Software manual

[22] Virtual Performance Solution: *Solver Notes Manual*, 2010

[23] Virtual Performance Solution: *Explicit Solver Reference Manual*, 2010

Chapter 10

Appendix A

This appendix contains images of the deformation of the base geometry shown in Figure 5.1. In these figures the top two images show the geometry in the same time step. The current time step is marked by a red square on the "Displacement-Force" curve. As these are the time steps from the result file, each time step is 1ms long. Each result file contains the initial time step (time step 0) and 85 after it.

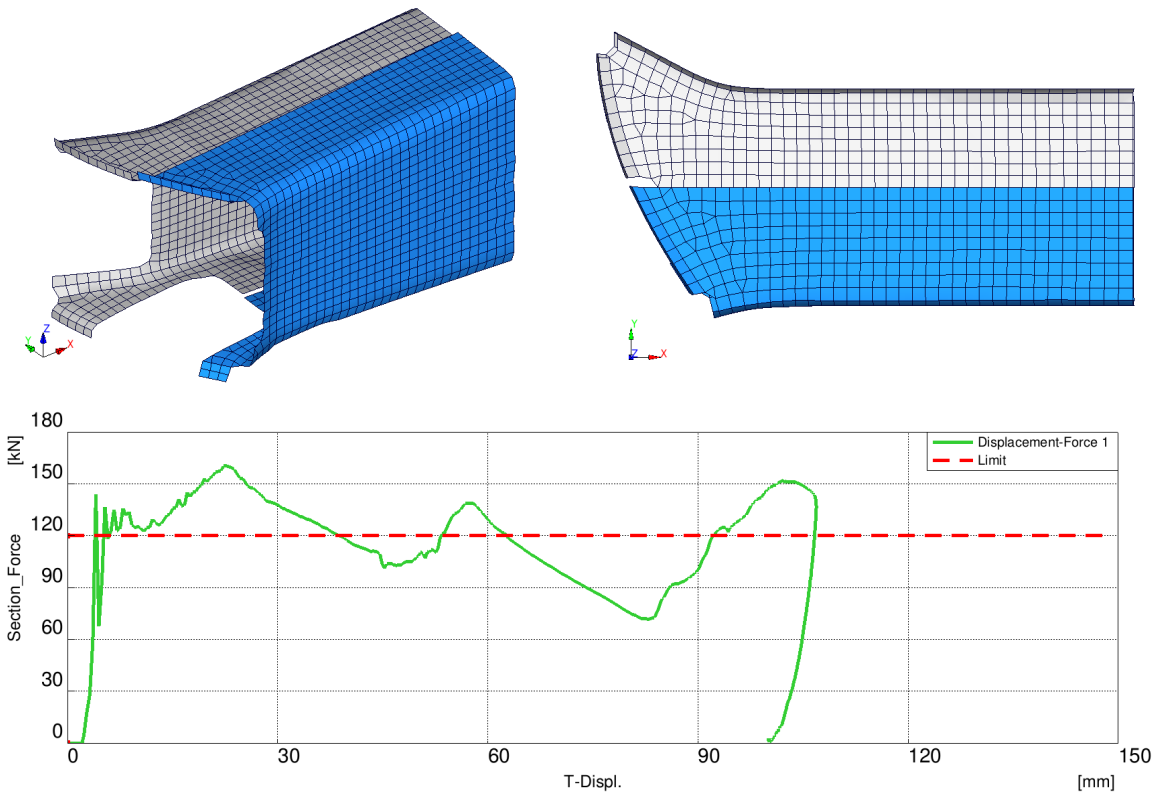

Figure 10.1. Time step 0

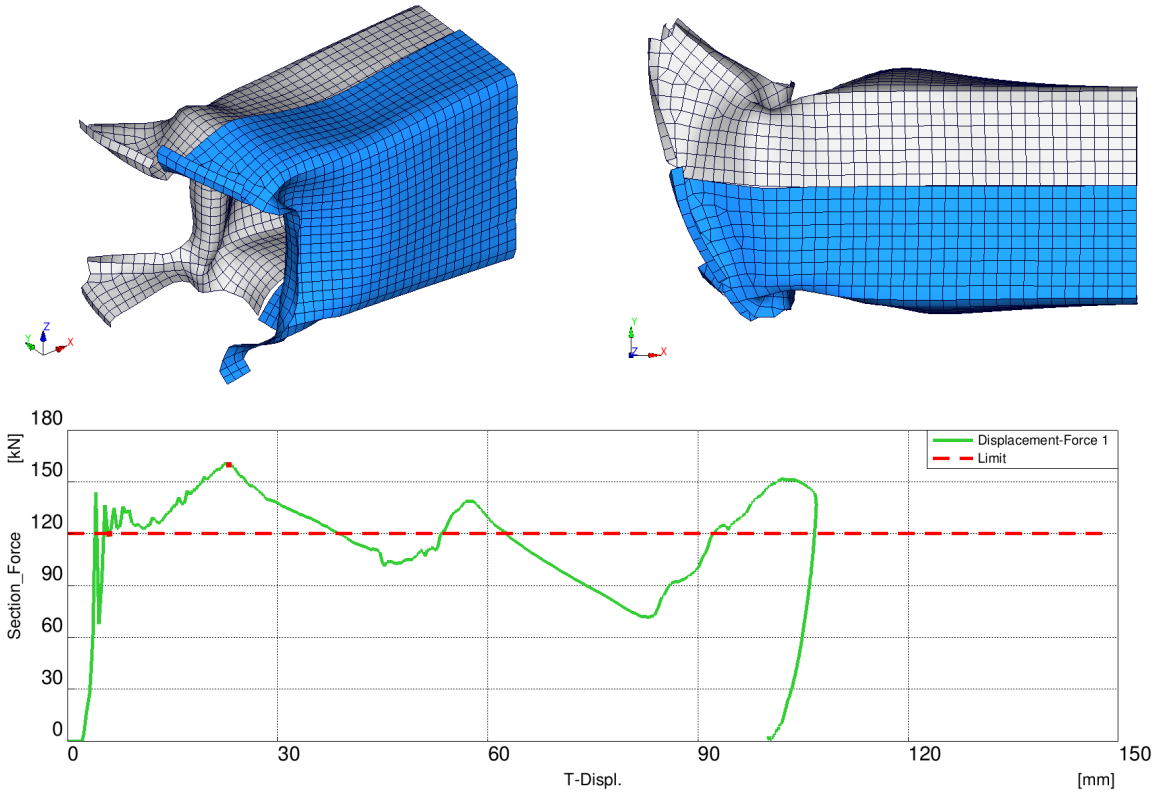

Figure 10.2. Time step 07

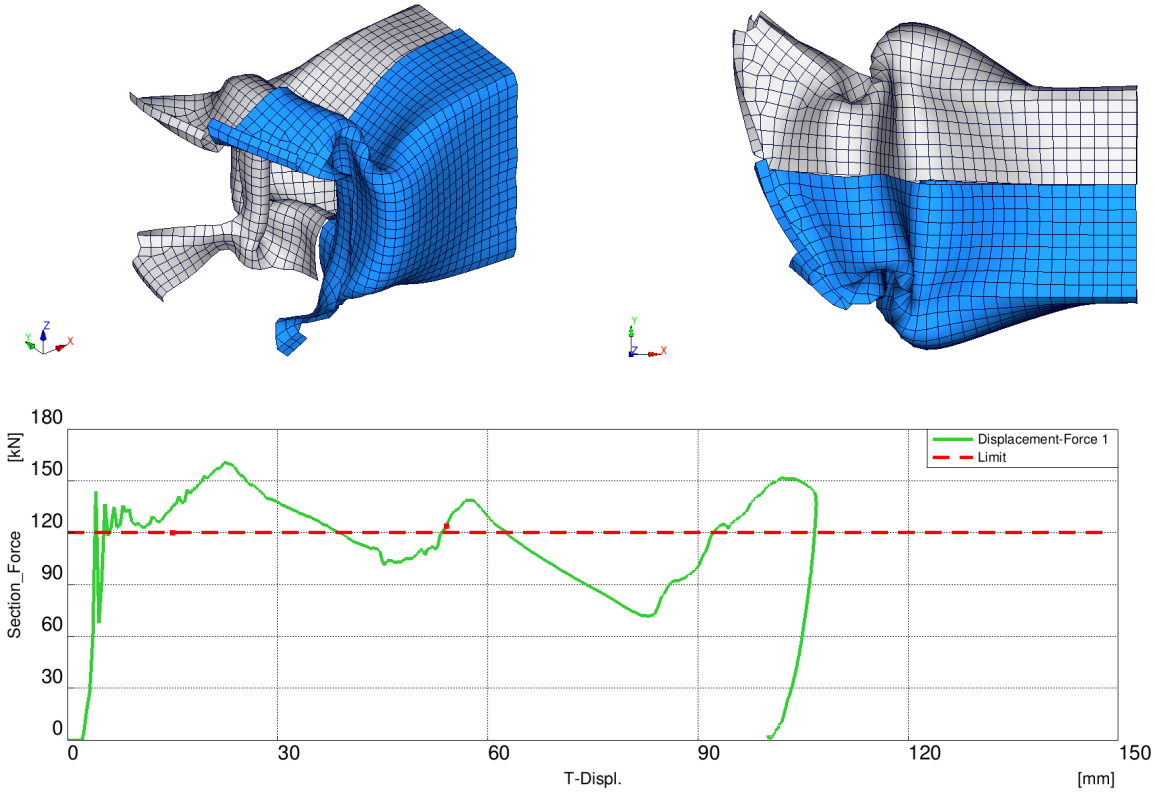

Figure 10.3. Time step 16

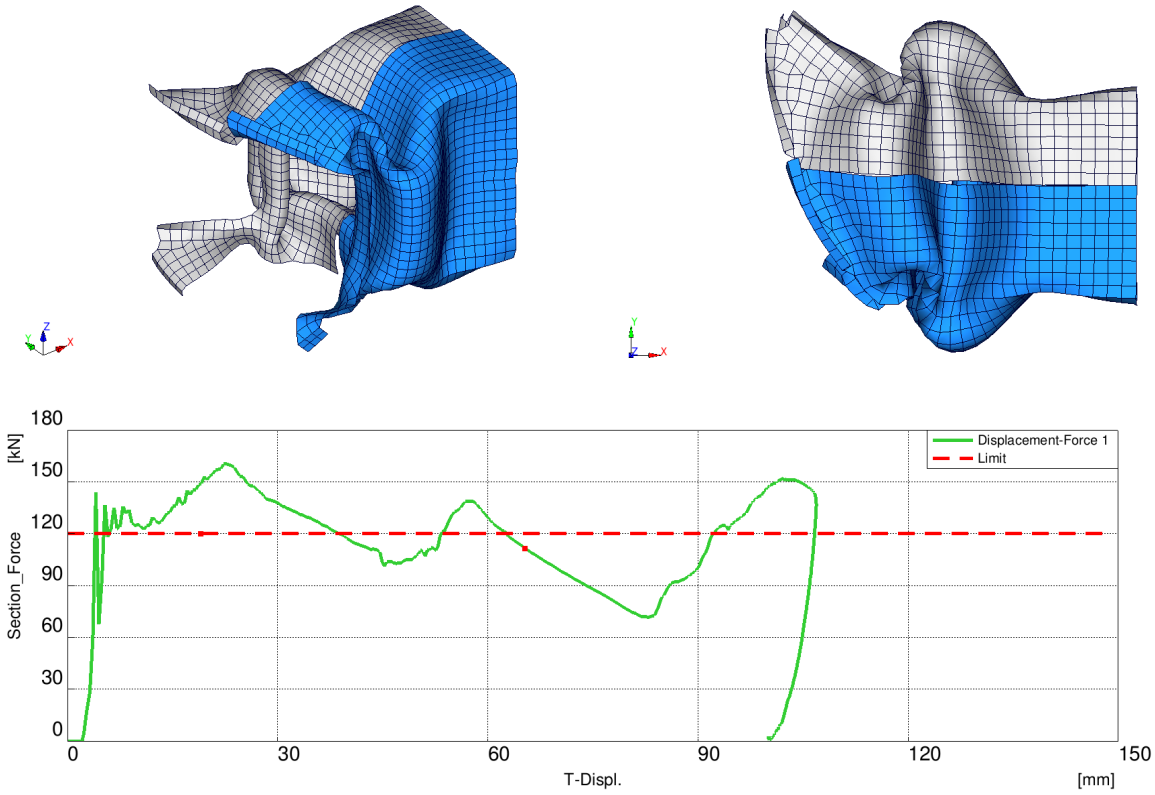

Figure 10.4. Time step 20

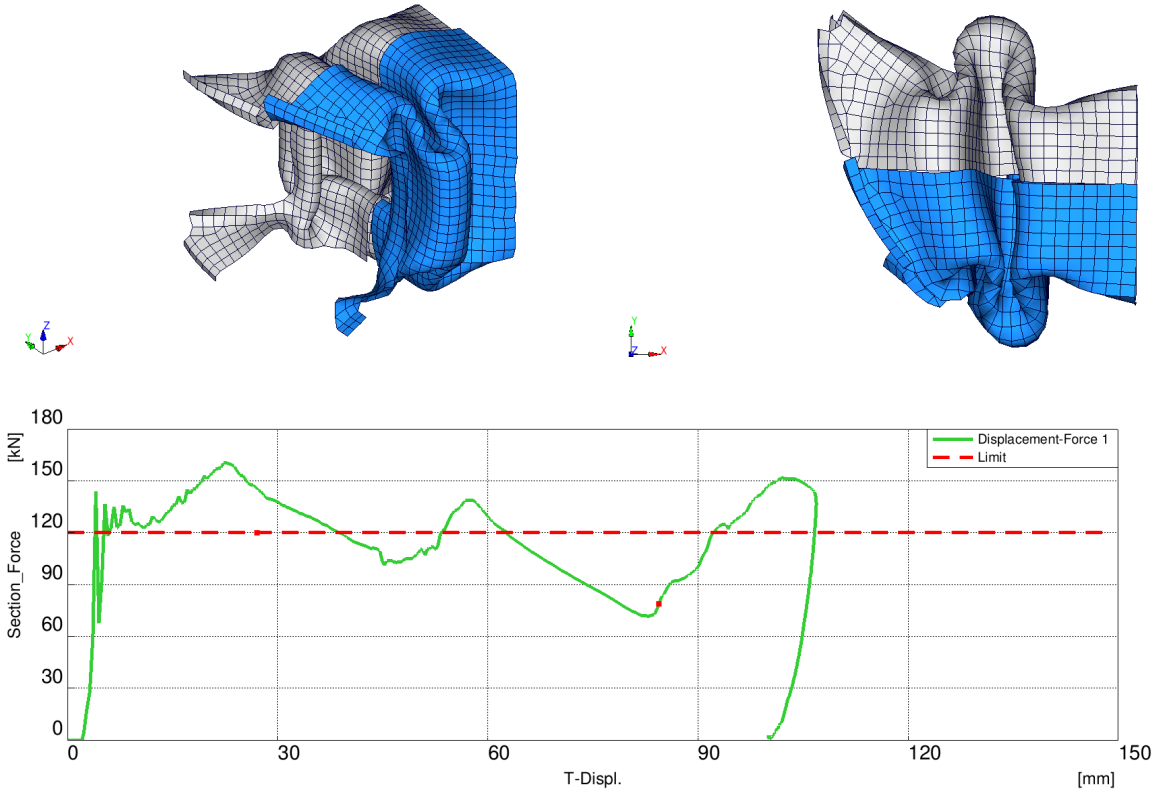

Figure 10.5. Time step 28

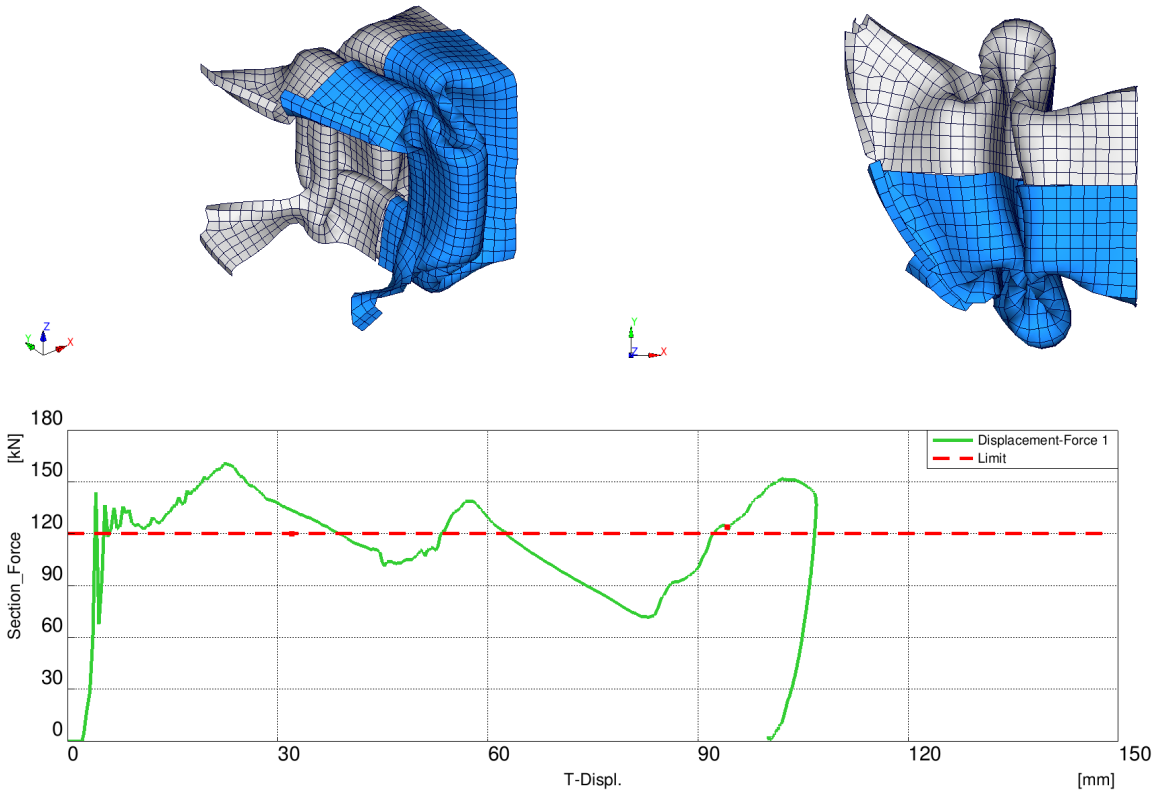

Figure 10.6. Time step 33

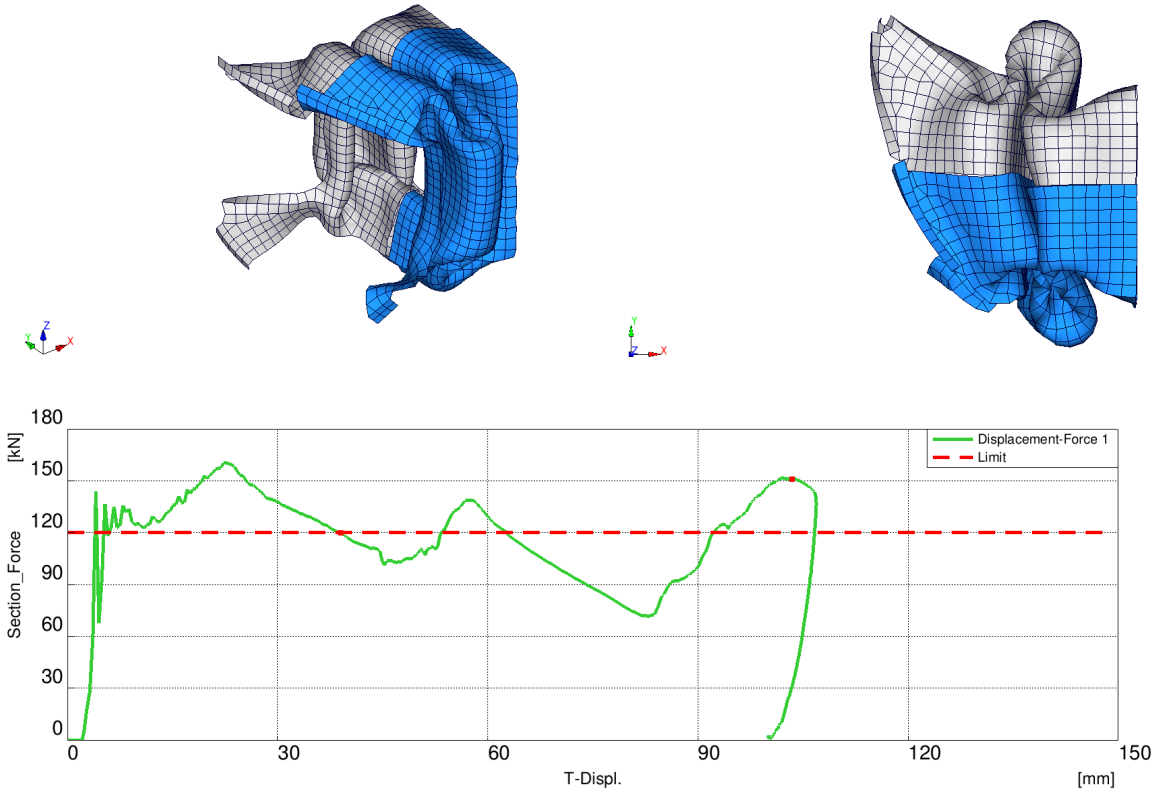

Figure 10.7. Time step 40

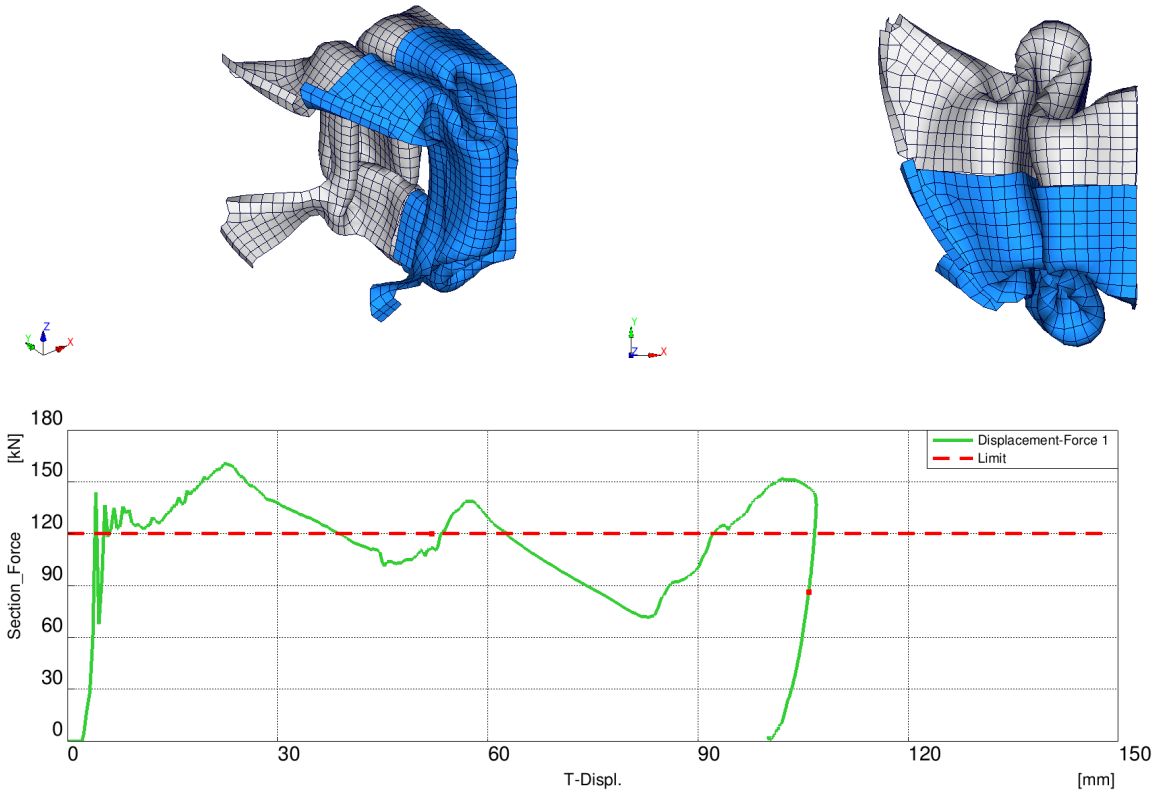

Figure 10.8. Time step 53

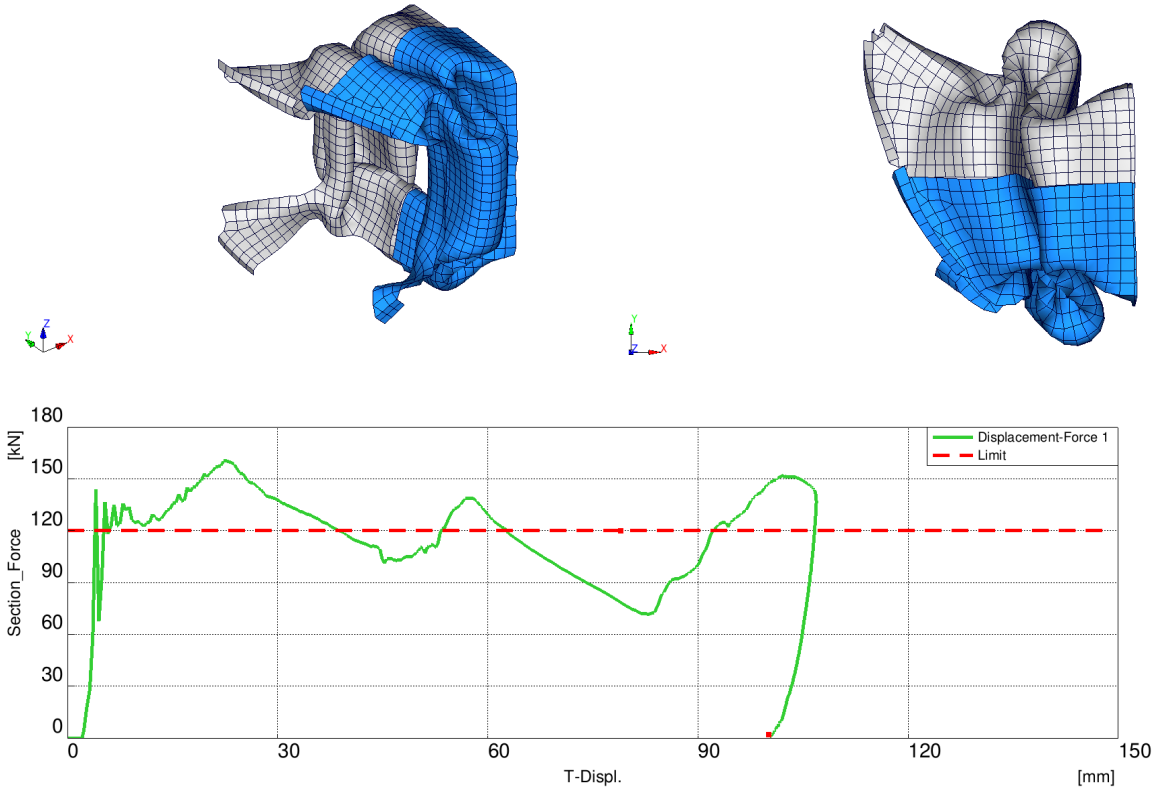

Figure 10.9. Time step 80

Chapter 11

Appendix B

This appendix contains the "Displacement-Force" curves of the Run 3 evolution steps, briefly shown in Figure 7.4. The score of each variant is included in the caption.

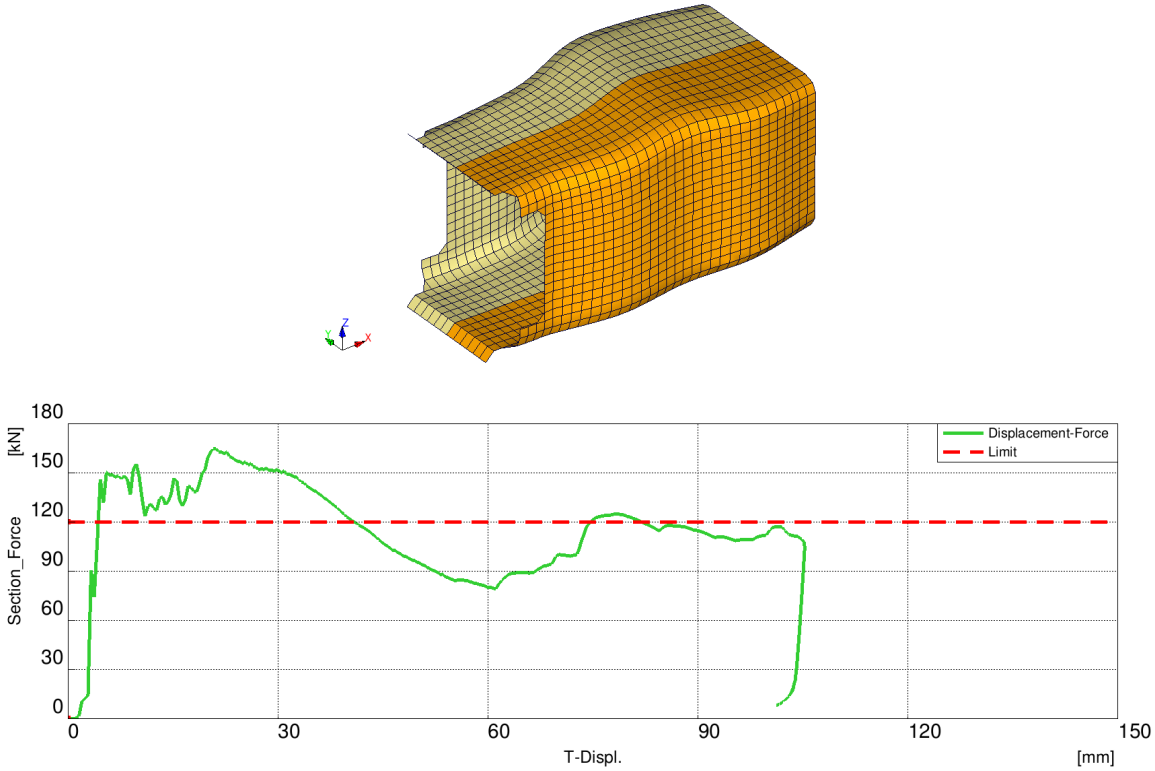

Figure 11.1. The 1st step, score: 79324

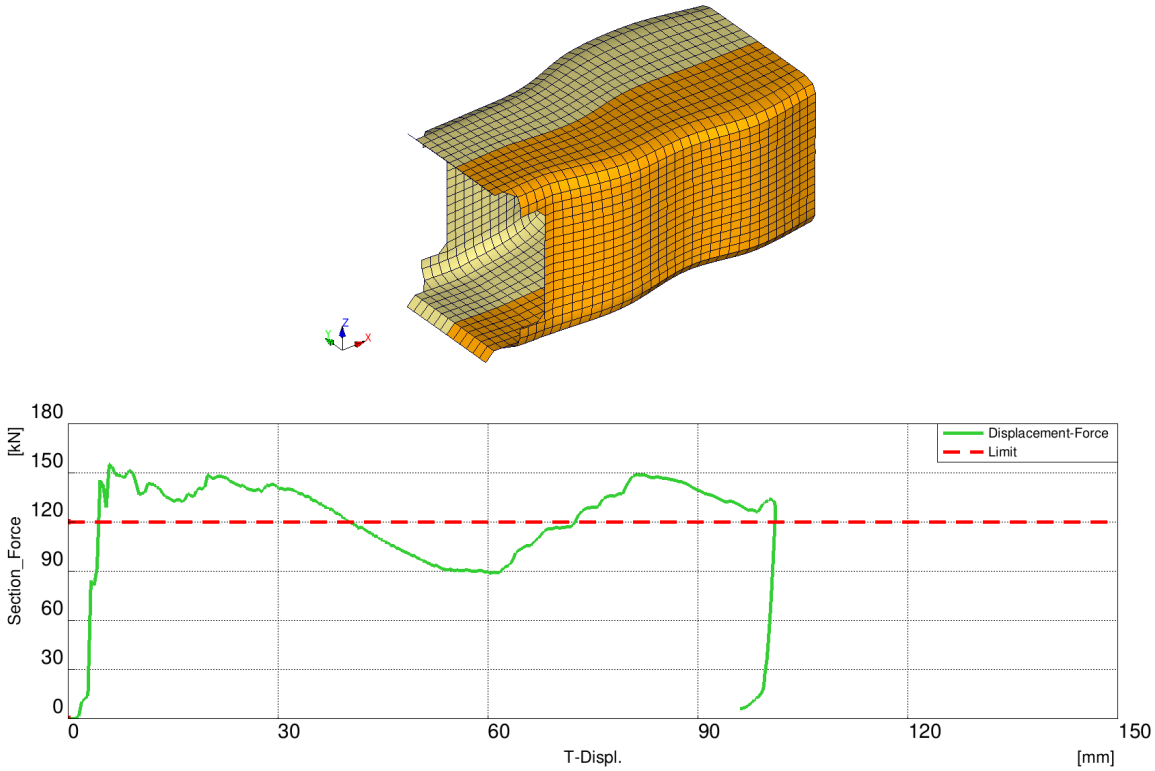

Figure 11.2. The 2nd step, score: 64657

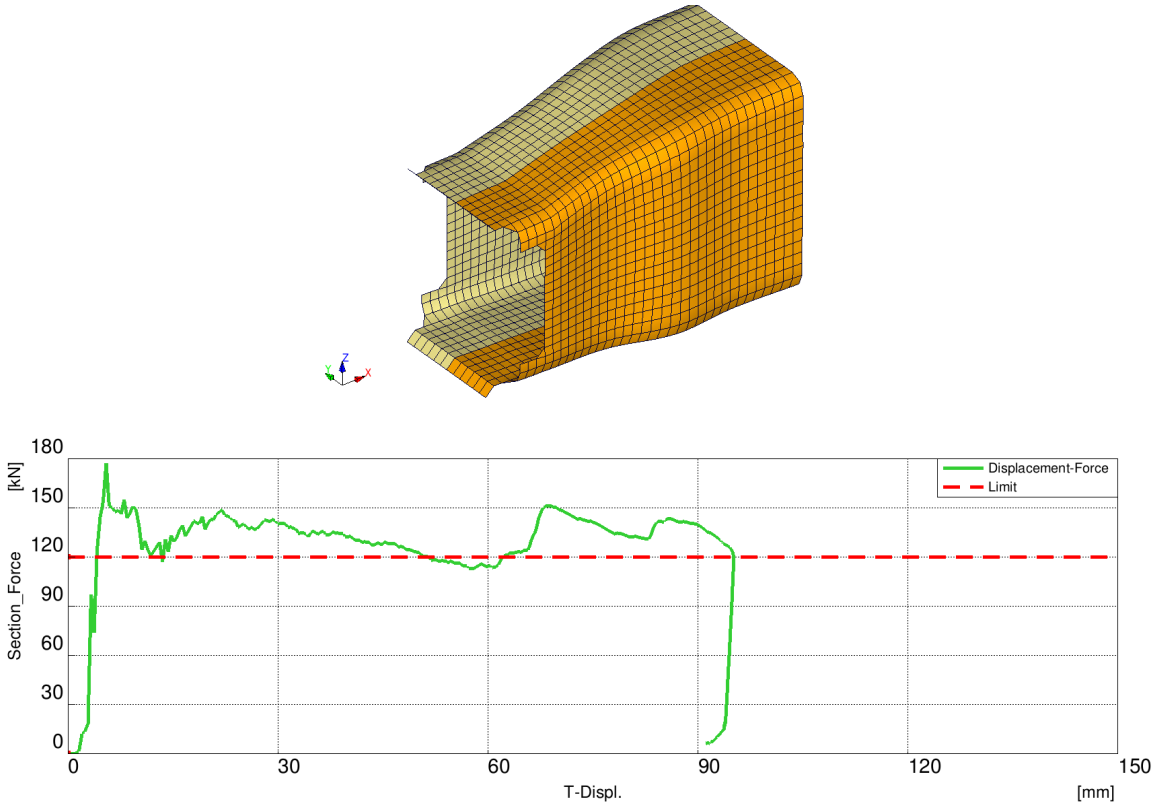

Figure 11.3. The 3rd step, score: 55140

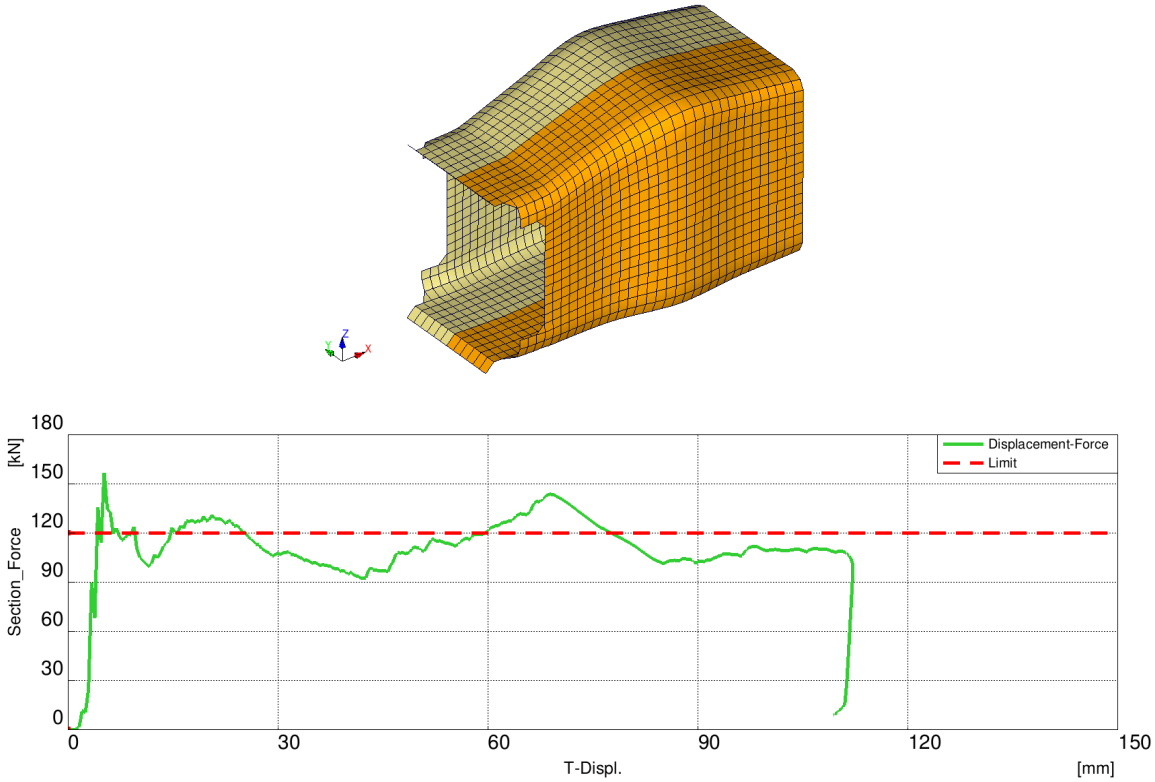

Figure 11.4. The 4th step, score: 25800

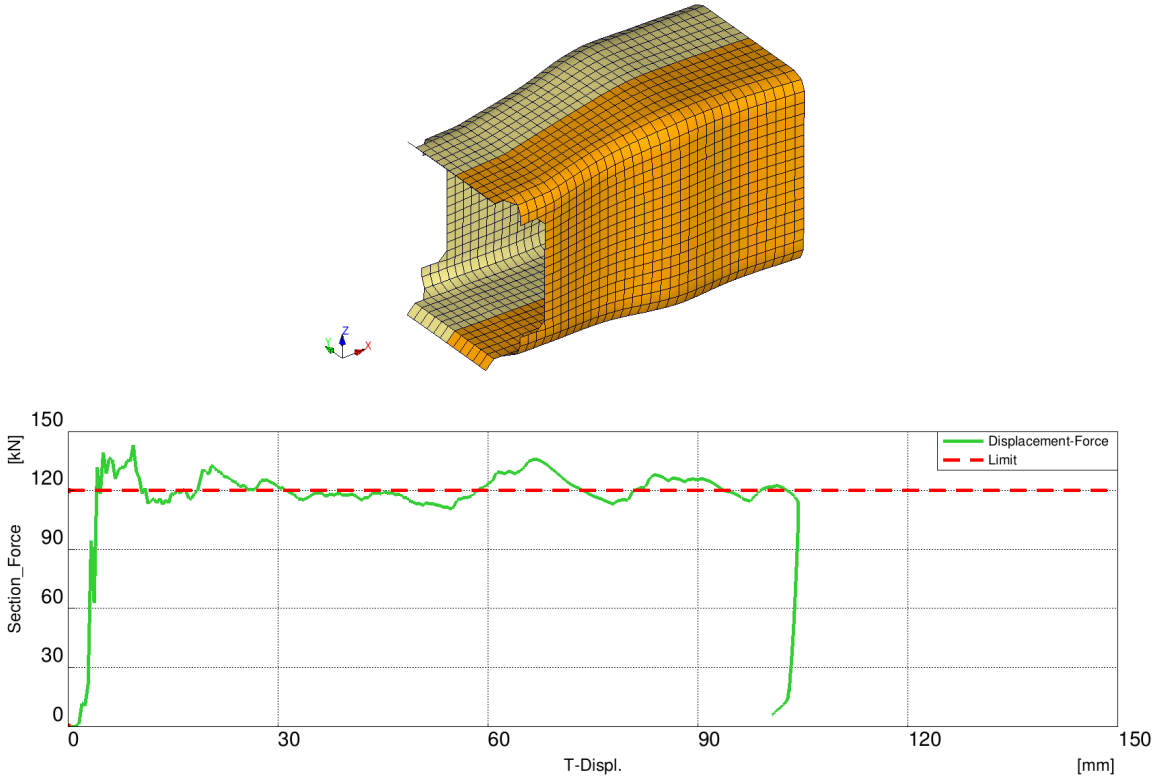

Figure 11.5. The 5th step, score: 9139

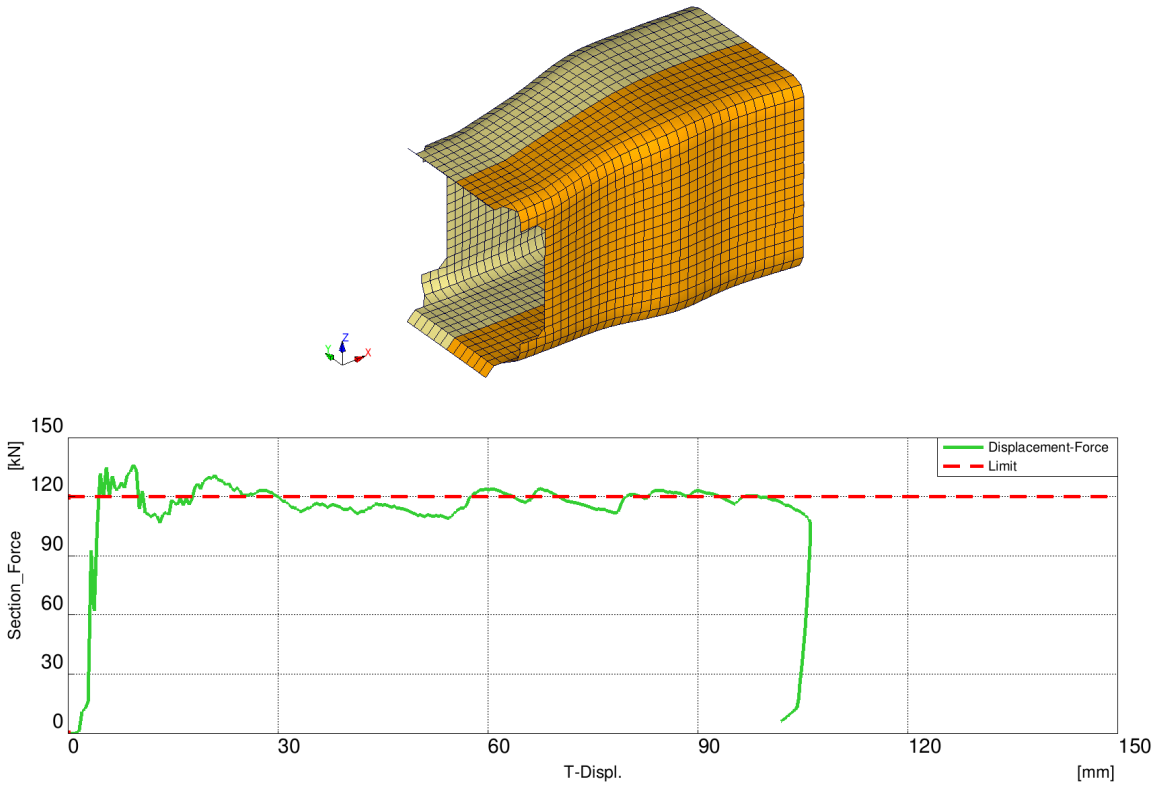

Figure 11.6. The 6th, final step, score: 6129

Chapter 12

Appendix C

This appendix contains the "Displacement-Force" curves of the Run 5-6-7 local optima, briefly shown in Figure 7.9. The score of each variant is included in the caption. As we can see from the scores, all of them are good, especially for not refined solutions.

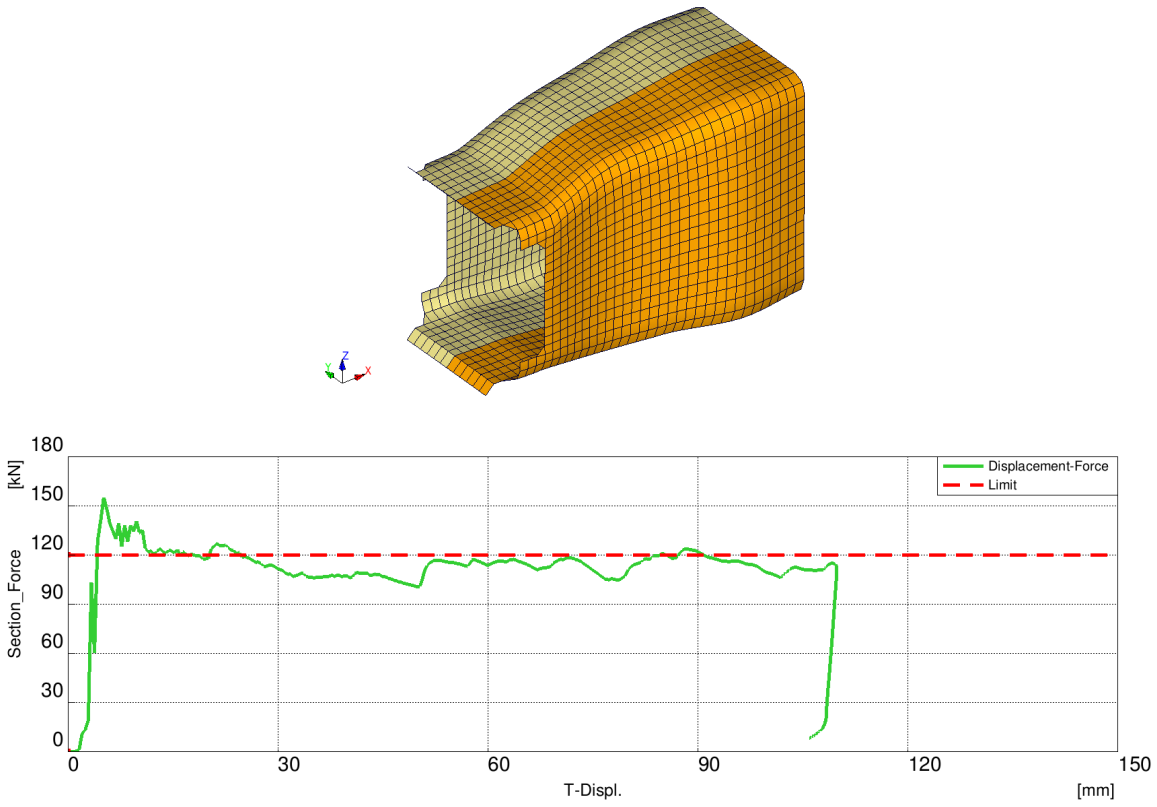

Figure 12.1. The 1st presented local optima, score: 13529

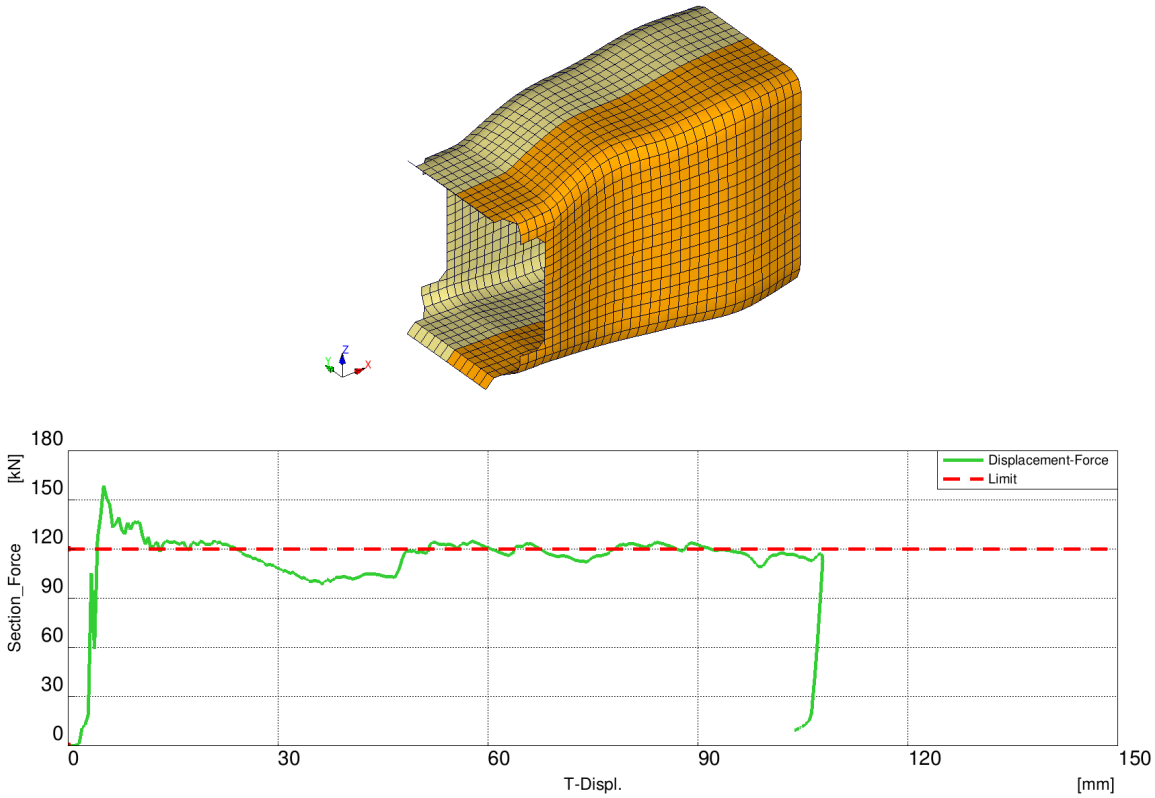

Figure 12.2. The 2nd presented local optima, score: 13465

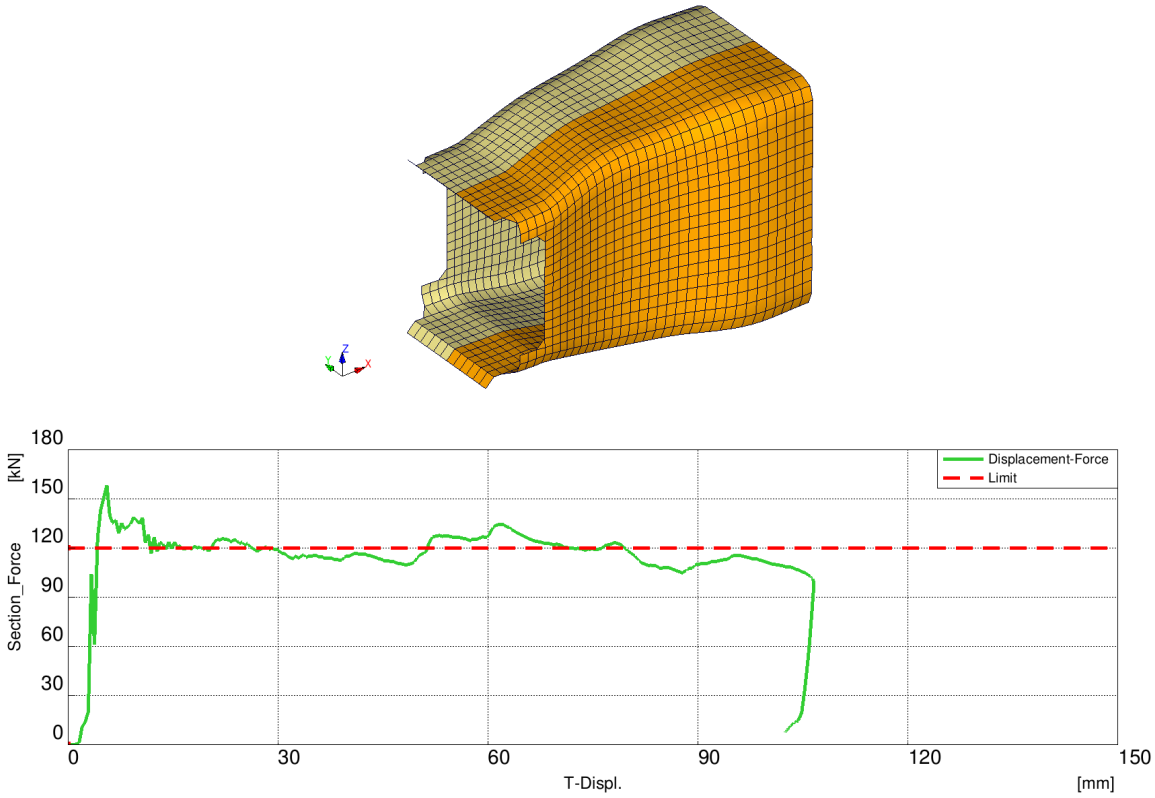

Figure 12.3. The 3rd presented local optima, score: 12997

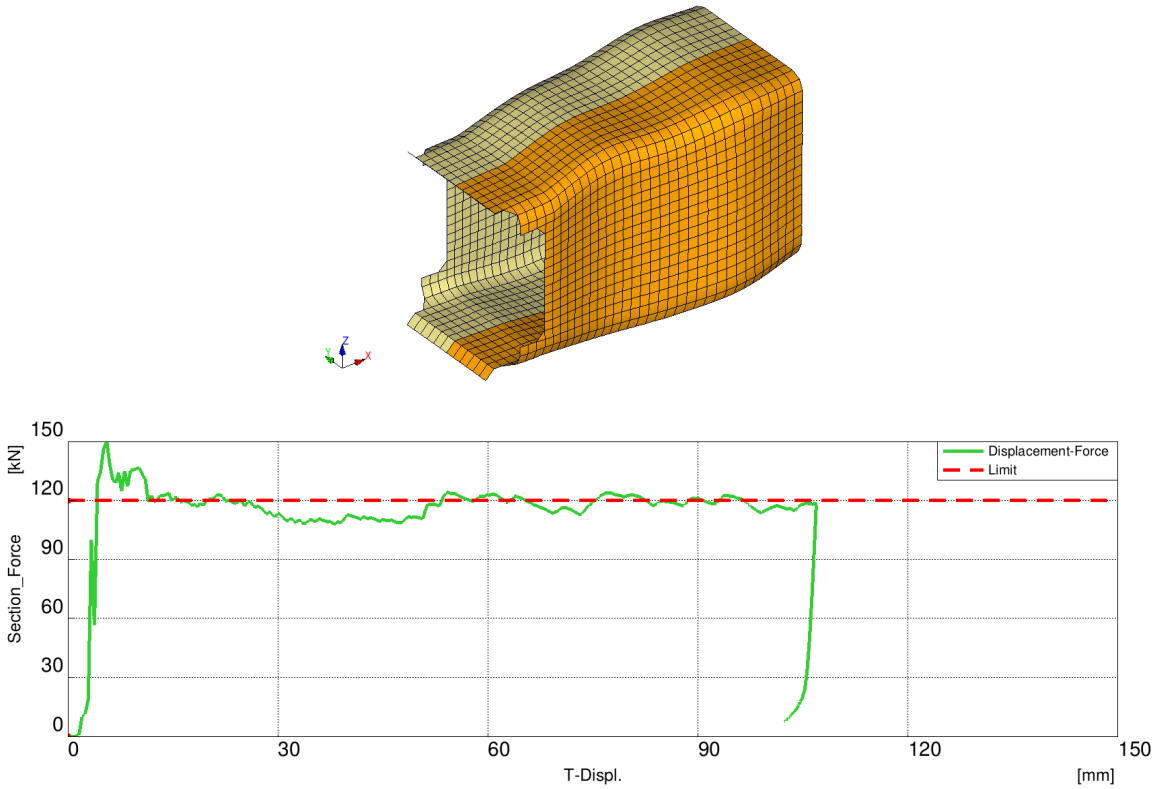

Figure 12.4. The 4th presented local optima, score: 8380

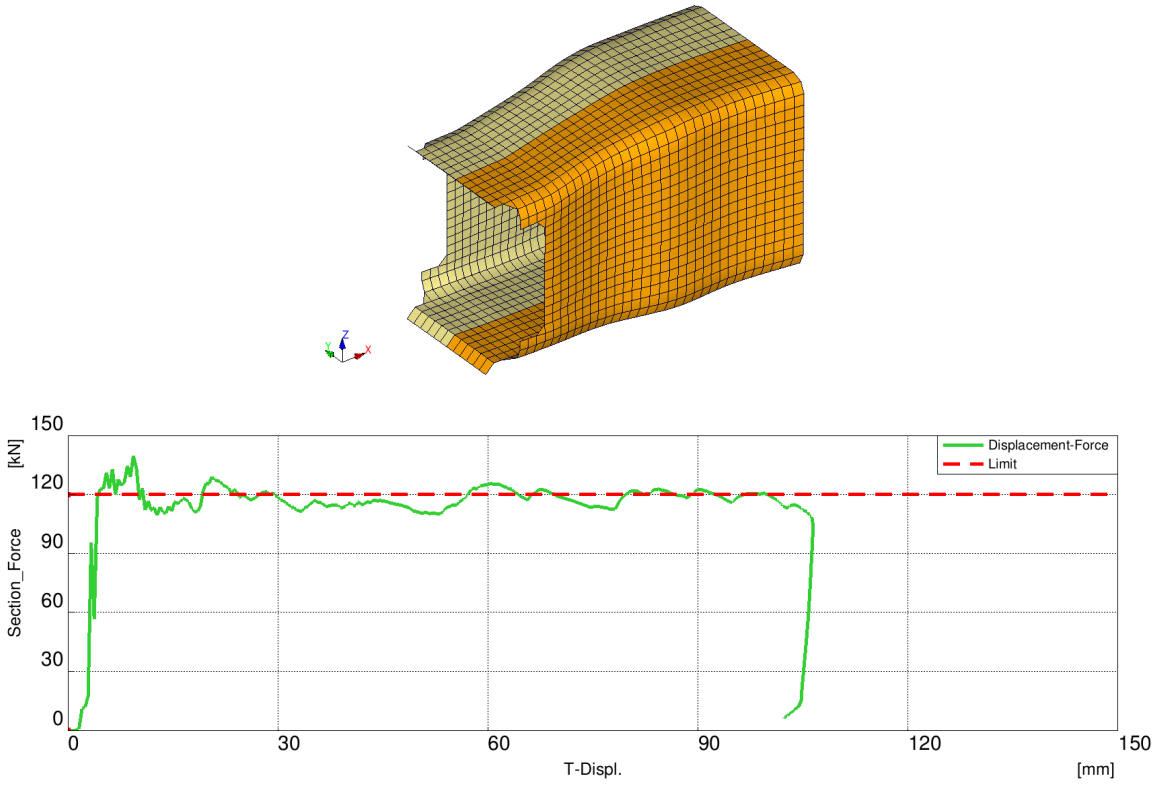

Figure 12.5. The 5th presented local optima, score: 5685

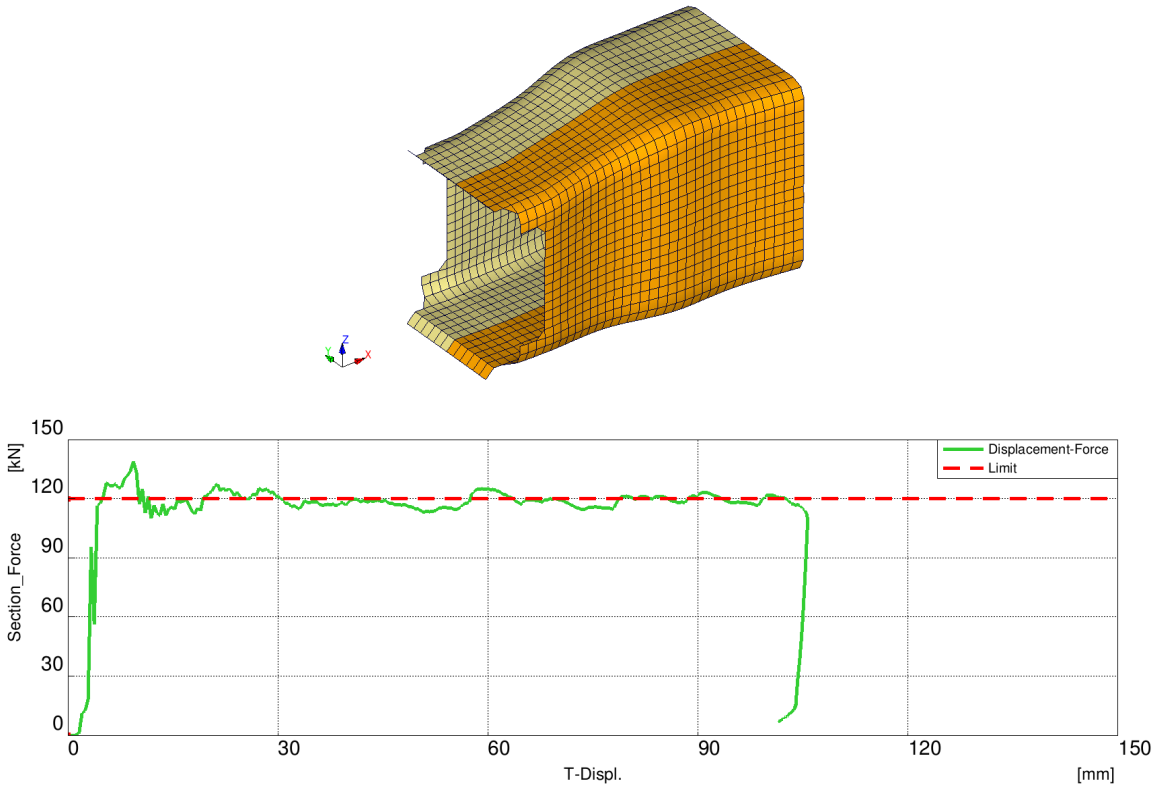

Figure 12.6. The 6th presented local optima, the best variant made, score: 4492

Chapter 13

Appendix D

This appendix contains the "Displacement-Force" curves of the Run 8 evolution steps, briefly shown in Figure 7.14. The score of each variant is included in the caption.

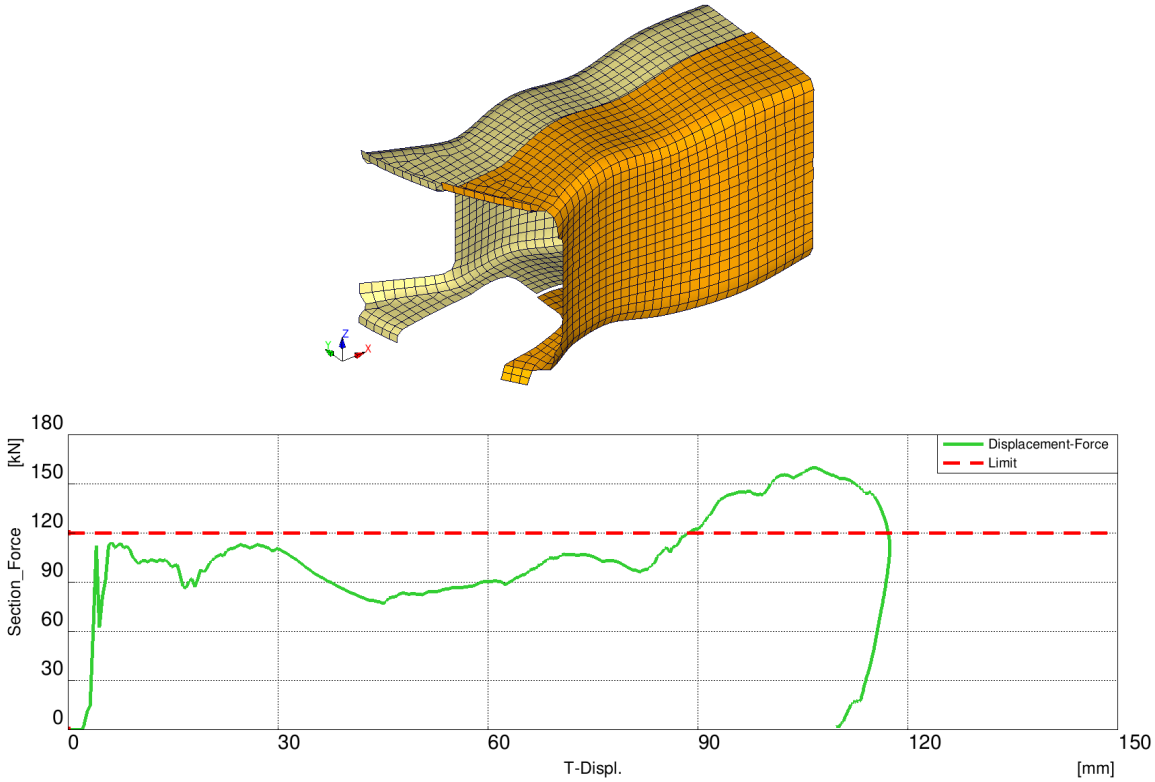

Figure 13.1. The 1st step, score: 98482

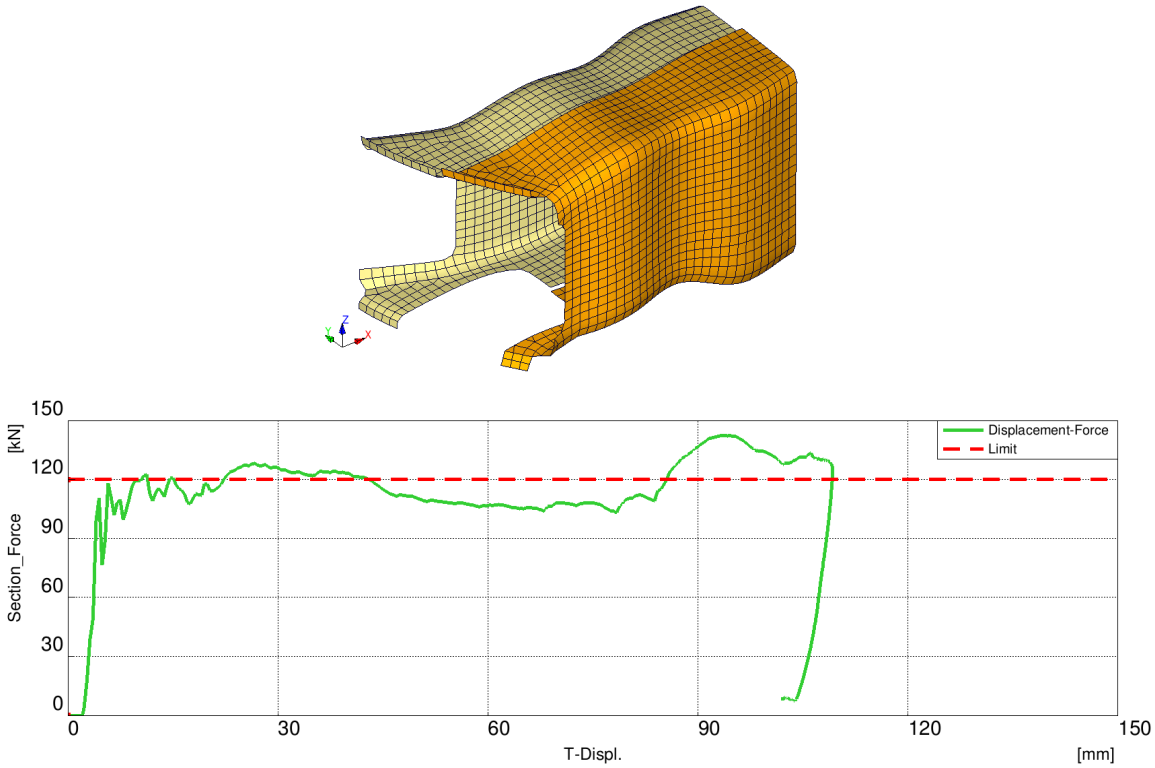

Figure 13.2. The 2nd step, score: 21933

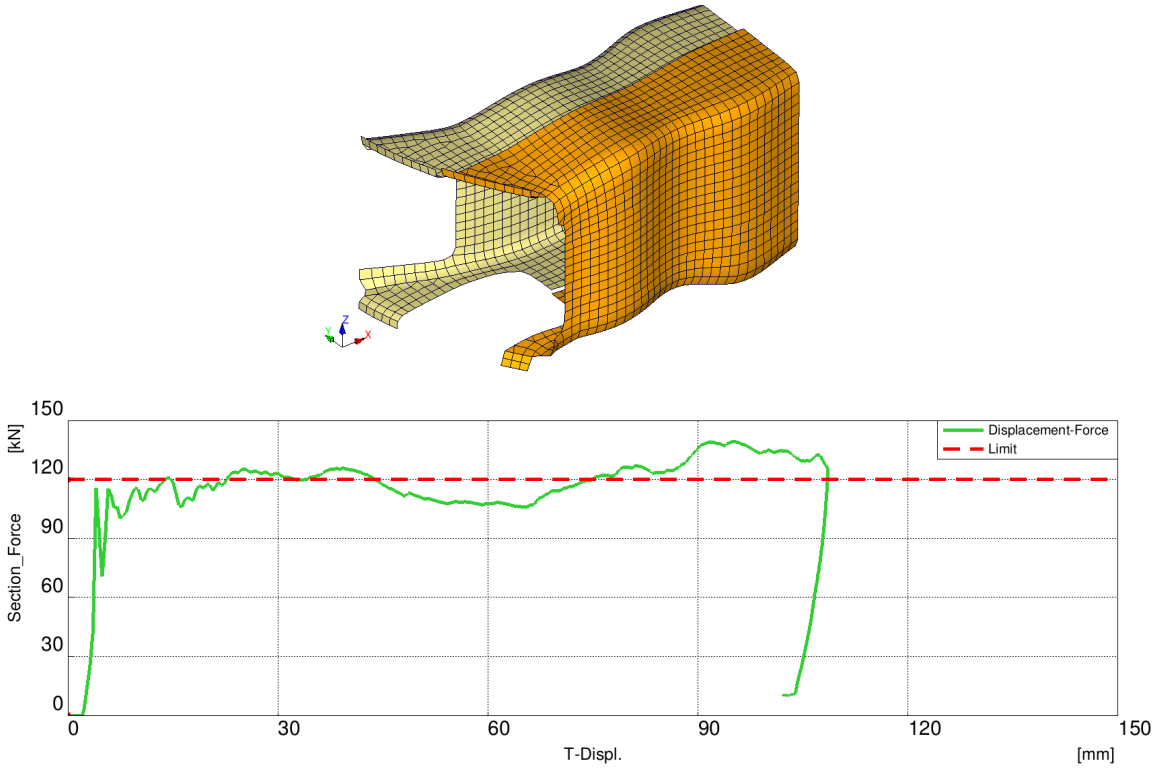

Figure 13.3. The 3rd step, score: 18533

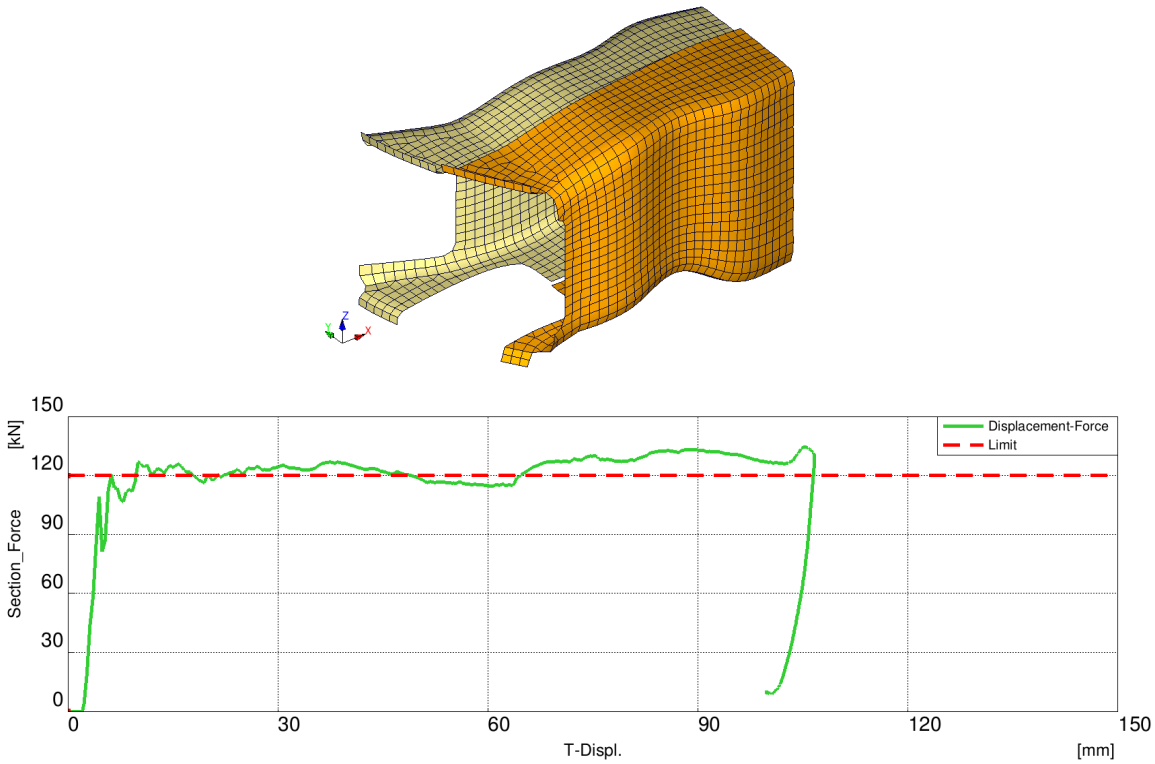

Figure 13.4. The 4th step, score: 15122

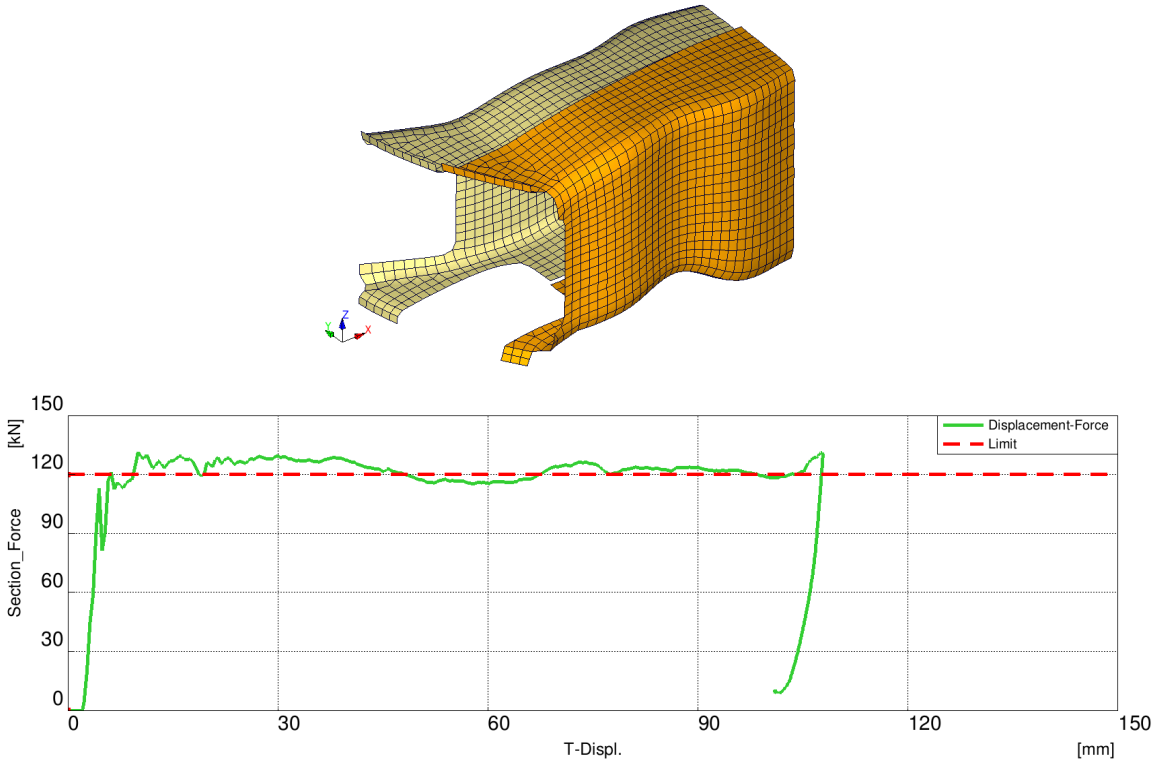

Figure 13.5. The 5th step, score: 10259

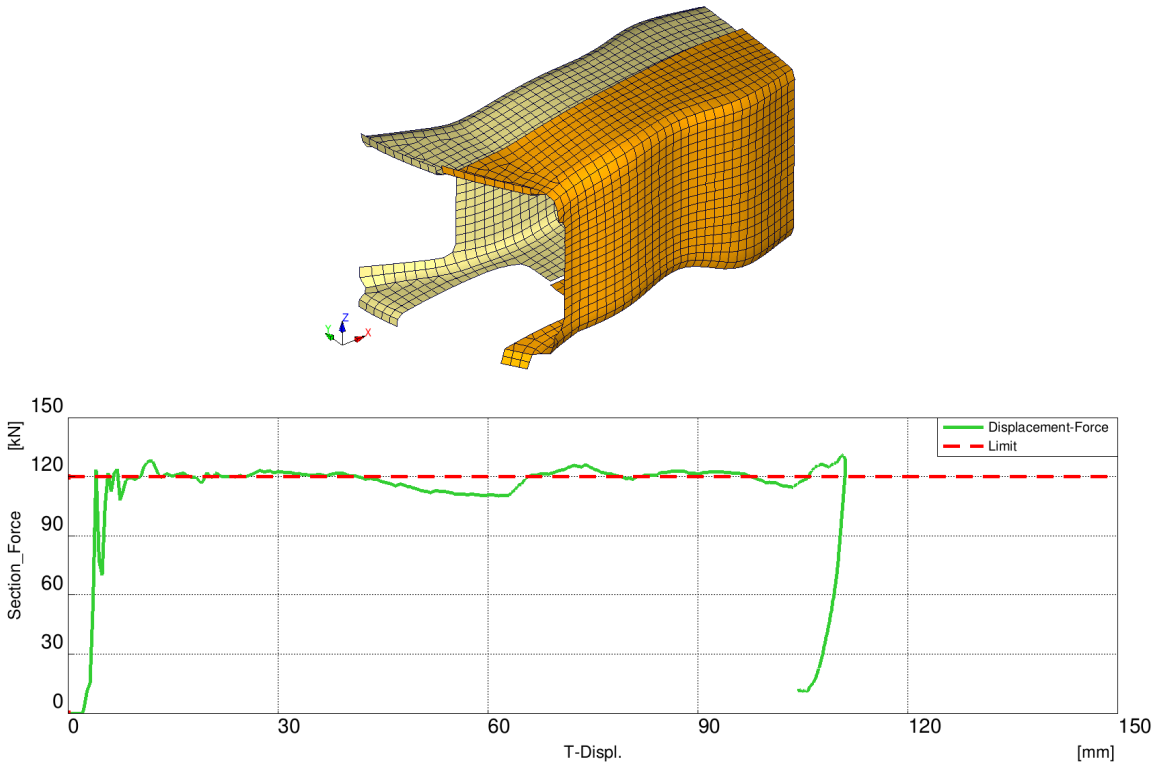

Figure 13.6. The 6th, final step, score: 6264