Bavarian Graduate School of Computational Engineering

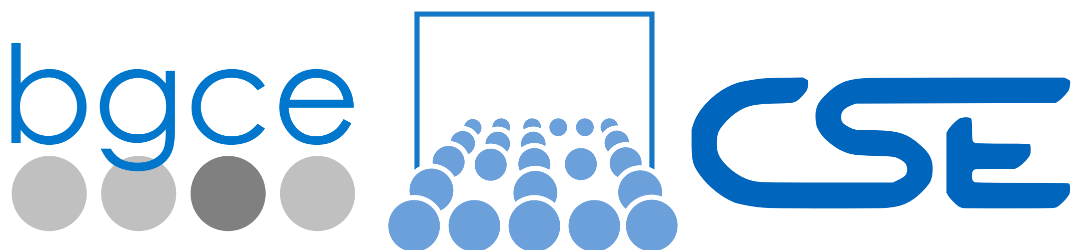
Technische Universität München

BGCE Honours project report

CAD-integrated Topology Optimization

Authors: Saumitra Joshi,
Juan Carlos Medina,
Friedrich Menhorn,
Severin Reiz,
Benjamin Rüth,
Erik Wannerberg,
Anna Yurova

Advisors: Arash Bakhtiari (TUM),
Dirk Hartmann (Siemens AG),
Utz Wever (Siemens AG)



Preface

The Bavarian Graduate School of Computational Engineering (BGCE) honours project at the Computational Science and Engineering (CSE) Institute of Technische Universität München (TUM) is a 10-month project where students conduct research on cutting-edge topics in the field of Computational Engineering, in cooperation with a partner in industry or academia. The 2015–16 project is titled *CAD-Integrated Topology Optimization* and is initiated and supervised in a cooperation between TUM and Siemens AG in Munich.

Acknowledgments

This Honour's project is carried out under the supervision of Dr. Dirk Hartmann, Dr. Utz Wever (Siemens AG) and Arash Bakhtiari (TUM). We also thank the Bavarian Graduate School of Computational Engineering for providing us an opportunity to participate in a project closely related to the industry in a highly relevant and challenging topic.

Abstract

Topology optimization is becoming an increasingly important tool in CAD. Several open-source topology optimization tools already exist, but are generally unsuitable for efficient incorporation in a design process, as there is no straightforward way to reacquire an editable CAD format. For this purpose, the software CADO (Computer Aided Design Optimizer), was developed. The software incorporates a topology optimiser, which works on voxelized CAD designs, and gives back outputs in voxel grid representations. An algorithm to retrieve a CAD-ready surface representation of this data was designed. From the voxel data, a surface is extracted using Dual Contouring. This is reconstructed into a network of tensor product NURBS surface patches using a linear least-squares fitting scheme. The constraint to get smooth (C^1) connections between the patches is applied by fitting to another network of points, related to the surface through a slightly modified version of a scheme by Jörg Peters. The NURBS surface patches are then readily converted to a standard CAD format, and other constraints are taken into account. In this report, we present the theory behind the methods used and the implementations thereof. We conclude with a summary of the capabilities of CADO, and how it can be extended in the future.

Contents

Preface ii

Acknowledgements iii

Abstract iv

Outline and Overview of the document vii

1. Introduction 1
- 1.1. Motivation . . . 1
- 1.2. Project Structure . . . 1

2. Background Theory 3
- 2.1. CAD Overview . . . 3
  - 2.1.1. Geometry Representations . . . 3
  - 2.1.2. Data Exchange Interfaces . . . 5
- 2.2. Topology Optimisation . . . 6
  - 2.2.1. Minimum Compliance: Problem Formulation . . . 6
  - 2.2.2. Physical and Mathematical Simplifications . . . 7
  - 2.2.3. Solid Isotropic Material with Penalization (SIMP) . . . 8
  - 2.2.4. Solution and Implementation . . . 8
- 2.3. Voxel Data Surface Reconstruction . . . 9
  - 2.3.1. Marching Cubes . . . 9
  - 2.3.2. Dual Contouring . . . 10
- 2.4. Bézier Curves and NURBS . . . 12
  - 2.4.1. Parametric Curves . . . 13
  - 2.4.2. Peters' Scheme for G^1 Bézier Surface Reconstruction . . . 14
  - 2.4.3. Fairness Functional . . . 18
- 2.5. Least-Squares Fitting of Parametrized Surfaces . . . 20
  - 2.5.1. Fitting Problem: Parametric Surfaces . . . 20
  - 2.5.2. Fitting Problem: Fairness . . . 21
  - 2.5.3. Fitting Problem: Peters' Scheme . . . 21

3. Implementation 23
- 3.1. Overview . . . 23
- 3.2. From CAD Model to Voxel Representation . . . 24
  - 3.2.1. Specification of Boundary Conditions for the Input Geometry . . . 24
  - 3.2.2. Face Extraction and Categorization . . . 25
  - 3.2.3. Voxelization . . . 27

3.3. Topology Optimization of Voxel Data 28
3.3.1. Topology Optimization Tool ToPy 28
3.3.2. Construction of ToPy Input File 28
3.3.3. Results of Topology Optimization 29
3.4. From Voxel Representation to Parametrized Surface Points 30
3.4.1. Surface Reconstruction 30
3.4.2. Parametrization of Datapoints 31
3.5. From Parametrized Surface Points to NURBS Representation 36
3.6. From NURBS to Standardized CAD File Format 38
3.7. Graphical User Interface 40

4. Results **42**
4.1. Product Overview 42
4.2. Test Cases 42
4.2.1. Cantilever 43
4.2.2. Bridge 44
4.2.3. GE Jet Engine Bracket 45

5. Summary and Future Work **46**
5.1. In a Nutshell: CAD-integrated Topology Optimization 46
5.2. Future Work 47

A. Surface Reconstruction **48**
A.1. Manifold Dual Contouring 48
A.2. Dual Marching Methods 48
A.3. Cubical Marching Squares 49

B. Installation Guide **50**

C. User Guide **59**

Bibliography **68**

Outline and Overview

The purpose of this document is to describe the implementation details of the *CAD-integrated Topology Optimization* software tool along with the theoretical background it relies on. The document is arranged in chapters, covering introduction to the field and the project, background theory and parts of implementation. The chapters are described in more detail below.

CHAPTER 1: INTRODUCTION

This chapter presents an overview of the motivation behind *CAD-integrated Topology Optimization*, including the current state of the field. It also provides general organizational information about project execution, timeline and structure.

CHAPTER 2: BACKGROUND THEORY

This chapter introduces the theoretical background for the implementation of the *CAD-integrated Topology Optimization* tool. It consists of five parts, each describing essential background of the topology optimization pipeline. Furthermore, detailed description of selected algorithms used in each step is given.

CHAPTER 3: IMPLEMENTATION

This chapter provides details on the implementation and structure of the *CAD-integrated Topology Optimization* tool itself. The different parts of the topology optimization and surface-fitting pipeline are presented along with underlying implementation details.

CHAPTER 4: RESULTS

This chapter presents the highlights of the *CAD-integrated Topology Optimization* tool in terms of user experience. This is followed by three different test cases that are carried out to show the performance of the integrated tool-chain.

CHAPTER 5: SUMMARY AND FUTURE WORK

The final chapter provides a summary of the project and states the scope of extensions and improvements.

1. Introduction

In this chapter, we develop the the motivation for the project and provide a short description of the problem task, together with a brief introduction to topology optimization and the project structure.

1.1. Motivation

A common problem in product design is to create a functioning structure using as little material as possible. Three decades ago, engineering design versions were drawn, prototypes created and experimental test performed. Nowadays, the field of topology optimization simplifies this process and stands as a powerful tool in engineering and design.

Topology optimization tackles the problem of material distribution in a structure in order to fulfil certain target loads. Several topology optimization open-source tools exist that are ready to use; however, it is still a challenge to incorporate these tools smoothly in the design process. The idea of this project is to allow these tools to work starting directly from *Computer Aided Design* (CAD) files and to transfer the resulting mesh-based solution back to the CAD world. Unfortunately, at the moment, there is no open-source solution for the conversion of mesh-based geometry to the spline-based CAD format. The common approach of converting each triangle of a mesh geometry directly into CAD format results in enormous file sizes. One of the biggest challenges of this project is thus to develop a conversion tool that feasibly provides a useful CAD-representation of the optimized surface.

1.2. Project Structure

The aim of the project was to provide a tool that allows to utilize topology optimization without leaving the CAD-framework. Therefore, the main goals of the project were as follows:

- Implementation of a topology optimization framework (by using and extending available open source libraries) that accepts geometry in CAD format.
- Development of a flexible tool for conversion of an optimized surface back to the CAD format.

The duration of the project had been set to 10 months. Hence, it was divided into 4 phases:

Phase 1: Getting familiar with the topic and agreement on the project specification.

Phase 2: Implementing the first part of the pipeline (Topology Optimization from CAD surface using existing tools); investigating the tools and algorithms available for the conversion of the geometry generated after topology optimization back to CAD format (later referred as

NURBS fitting pipeline); prototyping (using MATLAB) and evaluating results.

Phase 3: Implementing the prototypes developed on the previous stage, using non-proprietary languages; extending the NURBS fitting pipeline to more complex cases; finalising the first part of the pipeline.

Phase 4: Implementing the extended NURBS fitting pipeline; integrating with the topology optimization part and delivering the final product to costumer; providing a user-guide (see Appendix C) and an installation guide (see Appendix B)

2. Background Theory

In this chapter, the theoretical background for the implementation of the *CAD-integrated Topology Optimization* tool is presented. The chapter is divided into five sections: *CAD Overview* (section 2.1), *Topology Optimisation* (section 2.2), *Voxel Data Surface Reconstruction* (section 2.3), *Bézier Curves and NURBS* (section 2.4), and *Least-Squares Fitting of Parametrized Surfaces* (section 2.5).

2.1. CAD Overview

Computer Aided Design (CAD) refers to the process of designing a product using a computer. Until a few decades back, products were designed using a sketch board. It was a challenge to incorporate changes in construction drafts as well as to keep documentations up to date; hence, it is no surprise that CAD systems spread rapidly across all design development branches. They now have irreplaceable use in architecture, mechanical, electrical and civil engineering.

Depending on the discipline, different requirements are set on the virtual model. One may imagine that in a civil engineering model of a building a 2D floor plan is often sufficient; however, in the design of a mechanical motor a 3D model is always necessary. Given these circumstances, various CAD software bundles evolved in the different disciplines with completely different modelling approaches. Depending on the discipline, besides the geometry representation, additional parameters such as material properties or manufacturing information are stored. Standardized exchange interfaces are employed in order to switch efficiently between different data structures.

This section presents the relevant geometrical and computational aspects of CAD for the project; for a more thorough introduction we refer the reader to [2].

2.1.1. Geometry Representations

In general, two different ways of describing a geometry are used: a *constructive solid geometry* (CSG) or a *boundary representation* (BREP). Other approaches, such as a complete voxelized geometry are not common due to extensive memory consumption.

Constructive Solid Geometry

The core idea in this format is to start from a set of primitives, e.g. spheres, cylinders and/or cubes. Basic Boolean operations link these primitives towards a complex geometry, as illustrated in Figure 2.1.1.

Precise representation using very little storage memory is the key advantage of this format. However, not all desired forms can be represented through this format. Hence, a second type of geometry description is needed.

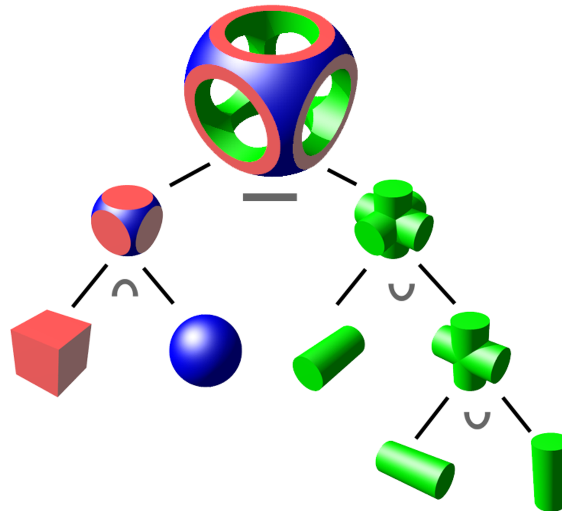


Figure 2.1.1.: CSG object tree. The picture shows the construction of a complex object from a cube, a sphere, and a set of cylinders. Figure from [3].

Boundary Representation

In this format, instead of storing the geometry information as geometrical objects, only the boundary surfaces of the body are saved. The interior is assumed to be uniformly filled. Especially in complex geometries, this approach simplifies the model to an extent where the amount of data becomes much easier to handle. Surfaces can then be for example stored as a set of triangles (as in *stereo lithography* (STL) files, see section 2.1.2 below) or in NURBS patches (see section 2.4). Furthermore, holes in the body are made possible by saving the surface normal of the respective boundary.

Through boundary representation, arbitrary geometries can be created. While the data sizes are usually larger than in CSG representation, BREP files are usually easier to work with. One also has to keep in mind, that non-physical geometries can result from BREP formats through a non-closed surface.

Voxel Raster

A very straightforward approach is to store a geometry shape as a regular grid of cubes, so called voxels. In the core, a raster of cubes is placed on the shape, and for each of the voxels the material information is saved. One can imagine that there are various alternatives: a boolean voxel grid, a grid with respective mass densities or saving arbitrary additional information to each voxel. As mentioned before, this representation is not common in CAD systems due to massive memory needs. However, modifications of the representation are more common which mostly deal with saving the shape surface in voxels as treated in [4] or OctTree representations.

2.1.2. Data Exchange Interfaces

CAD software programs usually use their own data formats; in order to exchange models standardized interface formats have been developed. Geometric models are compressed to certain geometry descriptions; transferring additional information, such as material properties or manufacturing information, is in general a difficult task and in some exchange file formats even prohibited. A few common exchange file types are described below, as also compared in [5].

STL File Format

The *stereo lithography* (STL) file format describes the model only by its boundary and is thus a BREP format. The idea behind its files is simple: the geometric model is discretized into a cloud of points, where sets of three vertices form a triangle; hence, a connected surface of triangles emerges which describes the geometry. The procedure is shown in Figure 2.1.2 for a two dimensional circle. The aforementioned triangles boil down to lines in two dimensions. The advantages and disadvantages of this approach become clear: It can be applied to an arbitrary geometry, but accuracy causes difficulties. In order to transfer high precision, geometries many vertices are necessary, resulting in big files. Still, as is illustrated in Figure 2.1.2, a perfect circle can never be represented.

ASCII STL files begin with a name and the data on the triangles is constructed as follows:

- a facet normal pointing outward
- a sequence of vertex coordinates

As this is the only information provided, no additional data such as material properties are transferred through STL files, reducing file size but also range of usage.

STEP and IGES File Formats

To overcome issues of insufficient precision, there exist more elaborate exchange formats; these save e.g. a circle as a parameter where no discretization step is involved. Also, the possibility of passing additional parameter information (e.g. density, manufacturing information) is required by certain users. Two popular file types that offer these functionalities are *standard for the exchange of product model data* (STEP) and *Initial Graphics Exchange Specification* (IGES) files.

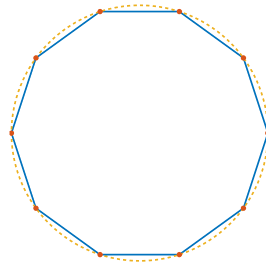


Figure 2.1.2.: STL discretization for a circle, self-made in MATLAB [6]. Note that the circle cannot be exactly represented by the vertices and edges.

The STEP file format is a realtively newly developed data exchange standard and documented in the ISO 10303 norm. Contrary to STL files, it uses a combination of CSG and BREP to store the geometry. Additional information (e.g. density, color) are passed through attribute sets that are stored besides geometry instances (e.g. a circle). A key disadvantage, however, is that they carry much more redundant information [5].

The Initial Graphics Exchange Specification is an American National Standard since 1981 to exchange graphics information. In the same way as the STEP format it uses a combination of CSG and BREP for geometry representation. Unlike the former, however, it is built only to exchange graphics information and does not store any manufacturing information. For example, the STEP file transfers information on the density; in the IGES format the only additional parameter stored on a node is the coloring information. Consequentially, file sizes are significantly smaller compared to the STEP file format [5].

2.2. Topology Optimisation

Topology optimization describes the process of finding the optimal distribution of a limited amount of material for a given area or volume based on a predefined constraint/minimization problem. Possible optimization goals are for example [7]:

- **Minimum compliance**, in which one seeks to find the optimal distribution of material that returns the stiffest possible structure. The structure is thereby subjected to loads (forces) and supports (boundary conditions). By maximizing the stiffness, the compliance is minimized. This is also analogous to minimizing the strain energy stored by the applied loads.
- **Heat conduction**, where one tries to optimize the domain of a conductive material with respect to conductivity for the purpose of heat transfer. This maximization problem is the same as minimizing the temperature gradient over the domain — a poor conductor will create a large gradient.
- **Mechanism synthesis**, where the objective is to obtain a device that can convert an input displacement in one location to an output displacement in another location. Thus, one hereby seeks the optimal design which maximizes the output force for a given input, or respectively, minimizes the input force for a given output.

As one can already imagine by this short list of optimization goals, topology optimization has a wide field of possible applications. Hence, it has become a well established technology used by engineers in the fields of aeronautics, civil engineering, materials, and mechanical and structural optimization. Furthermore, due to the rising significance of additive manufacturing techniques in industry, the realisation of complex optimized designs is now much easier [8]. For the rest of this section, and the rest of the document, we will concentrate on the *Minimum compliance* problem. Note however, that almost all parts in the *CAD-Integrated Topology Optimization* tool could just as well be applied to any other topology optimization problem.

2.2.1. Minimum Compliance: Problem Formulation

In order to constrain the resulting structure as little as possible, the formulation of the topology optimization problem is generally given as follows: for a given set of external fixture points,

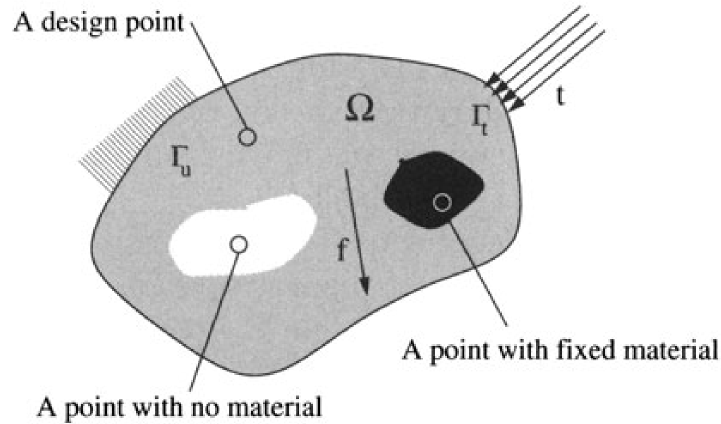


Figure 2.2.1.: The reference domain Ω for the minimum compliance problem. The problem is formulated such that for a set of external loads t on boundaries λ_t, body forces f and a set of fixed support points λ_u, the material distribution within Ω is such that the stiffness with regards to these loads and forces is maximal and the energy stored by the application of those forces is minimal. The problem also allows defining areas which either cannot or must be filled with material. Figure from [9].

external loads and/or body forces, the distribution of material within the reference domain should be found such that the structure has maximum stiffness. This is obtained when the structure has the minimum energy stored by external work for the applied forces. The problem is also usually formed to allow for regions in the domain to be specified as filled or empty of material (see Figure 2.2.1).

The formulation allows the problem to be cast as finding a displacement field u and a stiffness tensor field E that is in equilibrium with the applied loads, and that minimizes the external work done by these external loads do to reach that equilibrium.

2.2.2. Physical and Mathematical Simplifications

Typically, to turn this into a more tractable mathematical problem, a few physical assumptions are made: the material be isotropic and linearly elastic. From the assumptions of isotropy and linear elasticity of the material, the stiffness field becomes a constant of the material, defined where there is material in the domain.

The problem is also easy to cast into a weak form. First of all, we compute the integrated internal virtual work and external work. The former is the work of deforming the elastic material from equilibrium by an admissible displacement. The latter is done by the loads and forces to bring out this displacement. Having computed these, we set them equal to one another in order to conserve energy. As a result we obtain an equation that relates the equilibrium displacement, stiffness tensor, and the forces and loads. We then cast this into the weak form, which can be solved using Finite Element Methods (FEM). These can also incorporate the calculation of the external work done.

2.2.3. Solid Isotropic Material with Penalization (SIMP)

As described in the previous section, we aim to minimise the external work done by looking at different material distributions. However, the usual problem of finding an optimum arises: the search space is vast. After discretising the domain with FEM, the possibilities of where to put material at least are not infinite — but they still grow exponentially with the number of elements; hence, trying out one-by-one is not going to prove efficient. One popular way of recasting the problem to allow for easier solving is the SIMP model. Here, instead of either being present or not at a point, the material presence can take a continuous set of values between one and zero. The total final volume is then obtained and fixed by integrating this presence variable over the domain, instead of constraining the allowed occupied space. This allows for the interpretation as some kind of density.

In order to still obtain topologies where material is predominant in certain areas — of densities one, with the rest being empty at densities close to zero — a ”penalty” is applied to the intermediate values. This is effected by raising the density to a power > 1 in the elastic energy calculation, but not in the volume calculation. That way, an intermediate density value provides less elastic support, but still ”costs” as much volume, and will thus be suboptimal.

2.2.4. Solution and Implementation

In typical implementations, a heuristic iterative scheme is then used for finding a solution. The optimal solution is assumed to be stresses over all included parts (as they would otherwise be unnecessary, not providing any support). Thus, at places where the elastic energy is high, material is added if possible, and where it is low, material is likewise removed, with the values ”high” and ”low” being determined dynamically to keep the total volume constraint.

This whole scheme is one of the simpler topology optimisation schemes to implement, and has been done so in several pieces of open-source software, including a known 99-line Matlab code by Sigmund [10] and ToPy described in section 3.3. An example optimised topology is shown in Figure 2.2.2. For an extended explanation and discussion, as well as further alternative methods for topology optimisation, the interested reader is referred to [9].

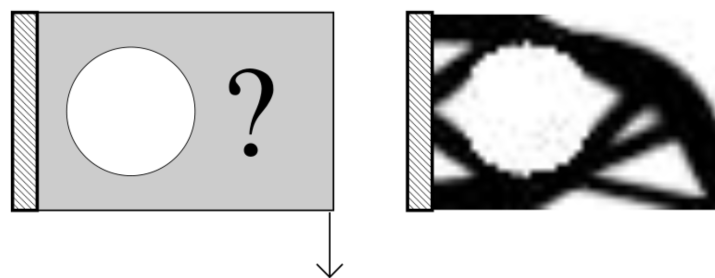


Figure 2.2.2.: Topology optimisation of end-loaded cantilever with fixed hole. The optimisation of a loaded cantilever is one of the model problems in topology optimisation, due to its simplicity and its multitude of used solutions throughout the history of engineering. The picture is taken from a known paper by Sigmund [10] where a 99-line Matlab code for topology optimisation is introduced.

2.3. Voxel Data Surface Reconstruction

A *mesh-based geometry*, which is a representation of the object at a set of (connected) points, is typically needed to fit NURBS and other curves to an optimized geometry. Conversion to a voxel-based surface reconstruction is necessary, since the topology optimization process results in a volumetric representation of density in each voxel.

In order to produce the mesh-based geometry, the data can be represented by a contour at a value of a smooth function in space, that is, an isosurface. Below, we describe two methods that solve this problem, Marching Cubes and Dual Contouring. This section is only intended for giving a brief overview of the different methods and the most important properties concerning our application. For more details we refer to [11, 12].

One should also note that surface reconstruction from voxel data is a special case of surface reconstruction and should not be mixed up with other surface reconstruction classes (like for example surface reconstruction from surface points in [13]). Voxel surface reconstruction works on datasets on Cartesian grids with either boolean inside-outside information or floating point values representing a certain quantity – e.g. density – on each gridpoint. The aim of voxel data surface reconstruction is to find the surface dividing the inside from the outside region (for boolean datasets) or denoting a certain isovalue (for floating point datasets). One example application for voxel data surface reconstruction is the process of generating isosurfaces from computer tomography data, which makes structures likes bones visible.

2.3.1. Marching Cubes

The *Marching Cubes* (MC) method [11] takes as an input a set of scalar function values on a Cartesian mesh and extracts an approximate isosurface in the form of a mesh of triangles. The method starts by dividing the space into cubes with the set of function values as cube vertices. These values are determined to be above or below the desired isovalue. According to which corners are set to be above or below, the corner configuration is then mapped to a polygon inside the cube, with vertices on the cube's edges. On an edge between a vertex above and a vertex below the desired isovalue, the exact location of the surface is determined via linear interpolation. Then this location is set as the polygon's vertex on that edge. A result of the MC algorithm is shown in Figure 2.3.1.

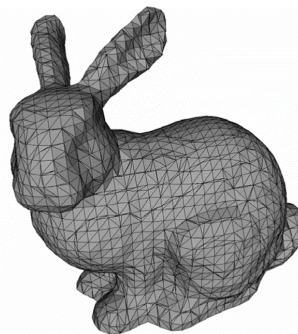


Figure 2.3.1.: The famous Stanford Bunny, a popular computer graphics test object, here after application of MC. Figure from [14].

The Marching Cubes Cases

Since there are 8 vertices on each cube, either above or below the isovalue, $2^8 = 256$ possible polygon configurations exist. However, many of these can be constructed by rotating or reflecting other configurations. Therefore 15 base cases which represent all the surface polygons of the Marching Cubes are sufficient to take into consideration. Figure 2.3.2 shows how these base cases look like. Notice that they are composed of triangles.

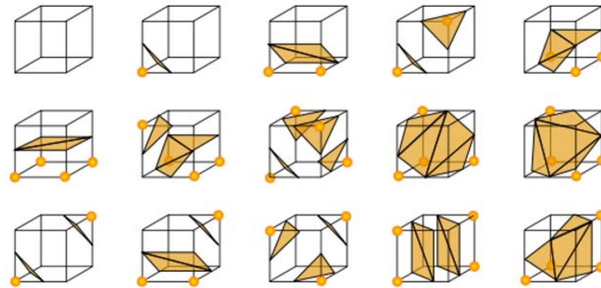


Figure 2.3.2.: The base cases of MC. These are drawn with each polygon vertex intercepting its edge in the middle between the cube's corners, as in the case when the isovalue is exactly halfway between the function values at the vertices. Figure from [11].

Cracks and Ambiguities

The original *Marching Cubes* (MC) algorithm presents two main problems. Firstly, it guarantees neither correctness nor topological consistency, which means that holes may appear on the surface due to inaccurate base case selection. The second problem is ambiguity, which appears when two base cases are possible and the algorithm chooses the incorrect one, or cannot decide on one. There are many extended MC algorithms that tackle the problems of the original one, getting rid of the ambiguities and providing correctness (see for example [15]).

2.3.2. Dual Contouring

The idea of *dual algorithms*, to which *Dual Contouring* (DC) belongs, is similar to MC. However, instead of generating polygon vertices on the edges of the cubes, this method locates them inside the cubes that have at least one edge which has vertex values both above and below the isovalue (sign changing edge). The basic algorithm can be summarized in these two steps:

1. Locate the position of the vertex inside each cube which has at least one sign changing edge.

2. Join the vertices associated with four cubes sharing a common edge to form a *quadrilateral face* (quad).

The approach can be seen in Figure 2.3.3, with a similar MC illustration for comparison.

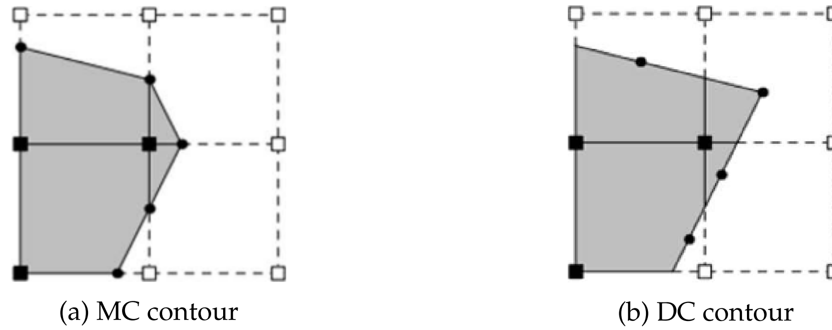


Figure 2.3.3.: Comparision of MC and DC for identical datasets. The vertices are created on the edges of the cubes for MC (Figure 2.3.3a) and inside the cubes for DC (Figure 2.3.3b). Note that the sharp feature in the top right cube is only be reconstructed by DC. Figure from [16].

Minimizing the Quadratic Error Function

We now wish to determine where in the cube the ideal place for the vertex is located – this is where different dual algorithms are distinguished. DC in particular generates a vertex positioned at the minimizer of a certain quadratic function. This function depends on the (interpolated) isosurface intersection points as well as the gradient at these points. Both quantities represent the first order Hermite data of the set. The quadratic error function defined in [12] is as follows:

$$E(x) = x^T A^T A x - 2x^T A^T b + b^T b \tag{2.3.1}$$

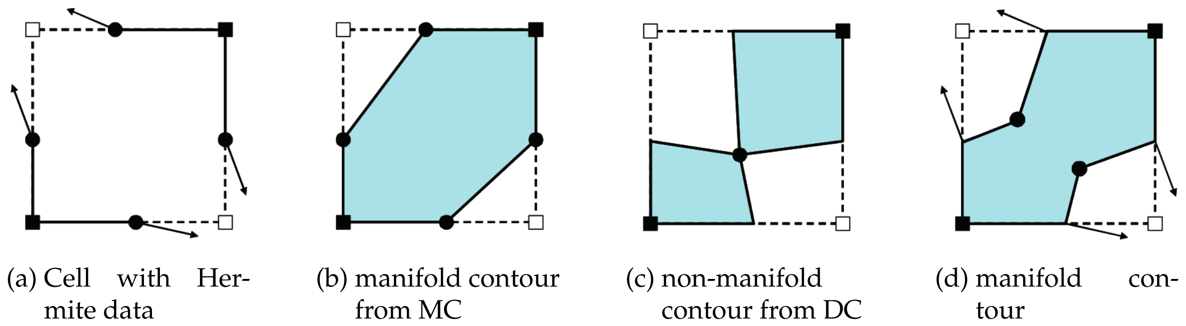
where the columns of the matrix A are the isosurface normals at the intersection points, and b is a vector containing the scalar product of the normals and the intersection points. This system can be solved numerically, for example as proposed in [12] by computing the singular value decomposition of A and forming the pseudo-inverse, truncating its small singular values. The effect of considering the gradient for the calculation of the vertex is huge: DC has the ability to represent sharp features like edges and corners. Computing the position of the new node by just taking the mean value of all the roots on the sign changing edges of one cube represent an easier approach, which does not rely on gradient information, but has the disadvantage that it is not able to represent sharp features. The resulting vertex is inside the cube, because it represents the average position of the nodes lying on edges of the cube.

Non-Manifold Surfaces

The risk to also obtain non-manifold surfaces represents one of the big drawbacks of DC. A manifold surface is defined by the following topological property:

> *A d-dimensional contour is locally a* manifold *if it is topologically equivalent to a d-dimensional disc.* [12]

This means that for 2D data we only get a manifold isocontour, if each vertex is connected to exactly 2 edges. For 3D we only get a manifold isosurface, if each edge is at maximum shared by two quads. Since the original method does not deal with this issue, extensions have been



(a) Cell with Hermite data (b) manifold contour from MC (c) non-manifold contour from DC (d) manifold contour

Figure 2.3.4.: Comparison of contouring for Hermite data Figure 2.3.4a. By comparing Figure 2.3.4b and Figure 2.3.4c one can see that DC produces non-manifold surfaces under certain conditions, while MC always produces manifold surfaces, but does not incorporate gradient information. Modifying DC properly finally gives manifold surfaces in Figure 2.3.4d. Figures from [17]

developed to solve this problem. Figure 2.3.4 shows one configuration creating a non-manifold surface for the basic method from [12] and its correct resolution by [17].

Topology-Safe Adaptivity

For keeping a check on the size of files and the complexity of the mesh, it is usually desirable to have as few faces as possible. However, in its basic form DC works with uniformly distributed data. The reconstructed surface mesh has uniform resolution on the whole area – this results in huge files and a uniformly high resolution if one wants to resolve fine features of the geometry. As a post processing step one could try to simplify the mesh, but especially for quad meshes this is a very demanding task. There are cases where it could also be impossible. [18].

The solution to this problem is adaptivity. Referring to [12] it is possible to implement DC in an adaptive and topology-safe way. This means we can simplify the obtained quad mesh and therefore reduce the number of quads needed to represent the surface, while still conserving the topological structure of our surface.

2.4. Bézier Curves and NURBS

Bézier curves and *Non-uniform Rational B-spline* (NURBS) are two types of curves which are very important in CAD (see section 2.1 above) used mainly to model surfaces. They are both defined parametrically by creating linear combinations of a set of *control points*, with the coefficients in these linear combinations being functions of the input parameter. There are many reasons behind their popularity, some of them being their relatively straightforward way of calculation and approximation, and the intuitive way of modification by changing the control points. In this section, we provide an overview of these concepts, as well as a description of *Peters' scheme*, a scheme for constructing a smooth (G^1) surface from the vertices in a polygonal mesh. For a more in-depth introduction and further material about NURBS, we refer to [19], and for further reading about Peters' scheme, we refer to the original article [1].

2.4.1. Parametric Curves

To define NURBS from a mathematical standpoint, we first define so-called *Bézier curves* and use them later for the definition of NURBS.

Bézier Curves

A Bézier curve is a *parametric* curve, which is often used for producing a smooth approximation of a given set of data points.

An analytical expression for the Bézier curve parametrized by the variable u is given by:

$$\mathbf{B}(u) = \sum_{i=0}^{n} b_i^n(u)\mathbf{p}_i \tag{2.4.1}$$

where $\mathbf{p}_i$ is the ith control point, $i \in 0, 1, \ldots, n$ ($n+1$ control points in total), and

$$b_i^n(u) = \binom{n}{i}(1-u)^{(n-i)}u^i$$

with $\binom{n}{i}$ being a binomial coefficient, is the ith *Bernstein polynomial* (see [20]) of degree n.

In addition to the expression with Bernstein polynomials, one can use a recursion formula (so-called *de Casteljau Algorithm*) for the construction of the Bézier curve, which we will not cover here.

Analogous to Bézier curves, one can also define a *Bézier surface*. One way of doing this is by extending the set of control points indexed in one dimension, to a two-dimensional mesh of $n \times m$ control points $\mathbf{p}_{i,j}$. Likewise, we extend the Bernstein polynomial basis to 2D by taking its tensor product with itself. The resulting *tensor product Bézier surface* is then given by the analytical expression

$$\mathbf{S}(u, v) = \sum_{i=0}^{n}\sum_{j=0}^{m} b_i^n(u)b_j^m(v)\mathbf{p}_{i,j} \tag{2.4.2}$$

B-Splines and NURBS

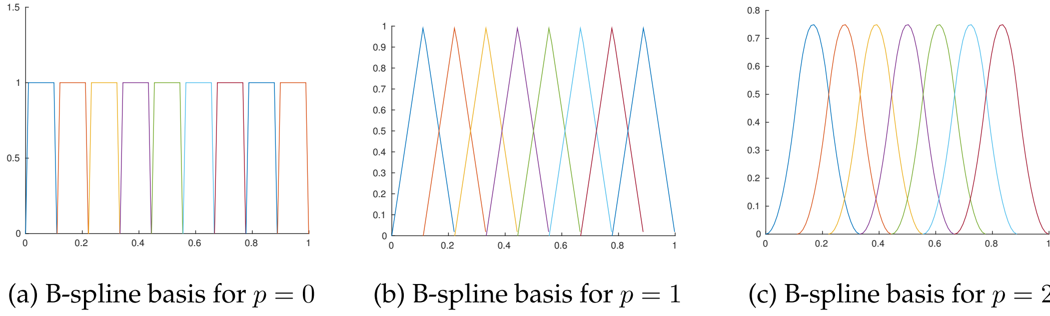
Extending the idea described in previous section, one could use *B-spline basis functions* (see below) instead of the Bernstein polynomial basis.

Unlike Bézier curves, the parameter domain for B-splines is subdivided by so-called *knots*. For the one-dimensional parameter domain $[u_0, u_m]$, the *knot vector* will be given by $u_0 \leq u_1 \leq ... \leq u_m$. In most cases $u_0 = 0, u_m = 1$ is chosen, so that we get a unit interval for our parameter values. For the case of NURBS, the knots $u_0, ..., u_m$ need not be equidistant – hence the "NU" (for Non-Uniform) in the name "NURBS".

Given a knot vector $[u_0, u_m]$ and a degree of B-spline p, the i-th B-spline basis function is then defined recursively as follows:

$$N_{i,0}(u) = \begin{cases} 1, & \text{if } u_i \leq u < u_{i+1} \\ 0, & \text{otherwise} \end{cases} \tag{2.4.3}$$

$$N_i^p(u) = \frac{u - u_i}{u_{i+p} - u_i} N_i^{p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1}^{p-1}(u) \tag{2.4.4}$$



(a) B-spline basis for $p = 0$ (b) B-spline basis for $p = 1$ (c) B-spline basis for $p = 2$

Figure 2.4.1.: B-spline basis functions, of degree $p = 0$ (left), $p = 1$ (middle) and $p = 2$ (right).

For $p = 0$ the basis fucntions are simply step functions, and for $p = 1$ we end up with so-called "hat" functions. Quadratic basis functions ($p = 2$) look more complicated (Figure 2.4.1).

By giving each of these basis functions a weight ω_i and normalizing them at each point by dividing by the total sum, we get the rational basis functions. Writing them out explicitly, in terms of B-spline basis functions $N_{i,p}$, the nth-degree NURBS surface with k control points P_i is finally given by:

$$\mathbf{C}(u) = \frac{\sum_{i=1}^{k} N_i^n(u)\,\omega_i \mathbf{p}_i}{\sum_{i=1}^{k} N_i^n \omega_i}. \tag{2.4.5}$$

B-splines have the following properties, which are useful for our problem:

- Degree n and number of control points $\mathbf{P}_{i \cdots m}$ are independent.
- B-Splines only change locally (depending on the degree n) when a control point is changed.

Analogous to the tensor product Bézier curve surfaces (see Equation 2.4.2), one can define tensor product B-spline or NURBS surfaces:

$$\mathbf{S}_{\text{NURBS}}(u, v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} N_i^n(u) N_j^m(v)\omega_{i,j}\mathbf{p}_{i,j}}{\sum_{i=0}^{n}\sum_{j=0}^{m} N_i^n(u) N_j^m(v)\omega_{i,j}}, \tag{2.4.6}$$

where the case with all $\omega_{i,j} = 1$ corresponds to a B-Spline surface; respectively a NURBS surface if any $\omega_{i,j} \neq 1$. With varying degrees and number of control points, these can be made to fit a variety of shapes. However, as the parameters u and v define a square in their two-dimensional parameter domain, there is a limit to what topologies may be realized with just one such NURBS surface. For example, an open cylinder could be constructed by one such surface where one of the sides meets its own beginning, whereas something with multiple holes - like a double torus, or a non-flat 8-shaped surface, would be impossible. Therefore, when using NURBS, surfaces are most often modelled using a network of connected patches. For more information about NURBS, see [21].

2.4.2. Peters' Scheme for G^1 Bézier Surface Reconstruction

Although the process of generating a NURBS surface may seem trivial (placing the control points near the desired surface location), getting it to assume a specified shape can be quite

a task. Generating a topology more complex than a torus requires several NURBS surfaces joined together. Thus, one needs to fulfil certain requirements in order for these surfaces to remain connected. For simple surface continuity (C^0), it is enough that the control points and knots on the edges of the two patches are the same, since then on both edges the surface follows a 1D–NURBS-curve from these points and knots. Smooth surfaces require higher-order continuity, which creates much more complex requirements. Several schemes have been created to automate such tasks.

The approach sometimes referred to as *surface splines* or *G-splines* [13] solves the task of generating a smooth surface by starting from a *control mesh* M of points, and computes Bézier surfaces by setting their control points to be linear combinations of the points in M. The coefficients are determined such that the resulting surfaces will be *tangent plane continous*, or G^1, or other desired degrees of smoothness.

One such scheme is the scheme of Peters, described in [1], which starts from an unstructured mesh of polygonal faces, and creates a G^1–continous surface from the location and connectivity of its vertices. This means that the normal vector to the plane is countinous, resulting in a smooth surface without sharp corners. The process consists of two steps, described below for a mesh of quadrilateral faces (quads) [13]. However, the scheme could also be applied for a mesh with any mixture of polygons.

Step 1: Mesh Refinement

In the first step, the mesh is refined through two iterations of *Doo-Sabin refinement*, as first described in [22]. This refinement is done by creating new points $\mathbf{m}_{ref}$ around the vertices $\mathbf{m}$ in the control mesh M. One such point is created for every face $f_{\mathbf{m}}$ that $\mathbf{m}$ corners, the new point $\mathbf{m}_{ref}$ being placed between $\mathbf{m}$ and the centroid $\mathbf{c}_{f_{\mathbf{m}}}$ of the bordering face $f_{\mathbf{m}}$ (the centroid of a face being the position of the face's vertices). After having done this for all the points in the control mesh M, we group all the points $\mathbf{m}_{ref}$ into a new refined mesh M_{ref}. Describing this mathematically for the faces F, with face $\hat{f} \in F$ having vertices $V_{\hat{f}}$:

$$M_{ref} = \{\mathbf{m}_{ref}; \mathbf{m}_{ref} = \alpha\mathbf{m} + (1-\alpha)\mathbf{c}_f; f \in F_{\mathbf{m}}; \\ ; \mathbf{m} \in M, \alpha \in (0,1)\} \tag{2.4.7}$$

$$\text{where} \qquad \mathbf{c}_{\hat{f}} = \text{average}\left(\mathbf{m}_{\hat{f}}\right)_{\mathbf{m}_{\hat{f}} \in V_{\hat{f}}} \tag{2.4.8}$$

$$\text{and} \qquad F_{\hat{\mathbf{m}}} = \left\{\hat{f} \in F; \hat{\mathbf{m}} \in V_{\hat{f}}\right\} \tag{2.4.9}$$

where α is a smoothening parameter, controlling the sharpness of the corners and edges, which we for simplicity set to $1/2$, to get a simple midpoint.

Thus, in every refinement step on an n–gon, n vertices are created, giving 4 vertices for a quad in the original control mesh. These are then joined up with the neighbours on the quad to form a smaller quad, and with the neighbouring points from the same vertex on the neighbouring quads, forming a quad along each edge. Around a quad corner, where n quads meet (or n edges in the general case), we instead get an n–gon around the corner vertex. After two refinements, we thus get a mesh of vertices $M_{2ref} := V_x$ that mainly consists of quads, with possibilities of getting polygons with other number of edges around the vertices of the original mesh M. The mesh and the resulting structure after two subdivisions can be seen in Figure 2.4.2b.

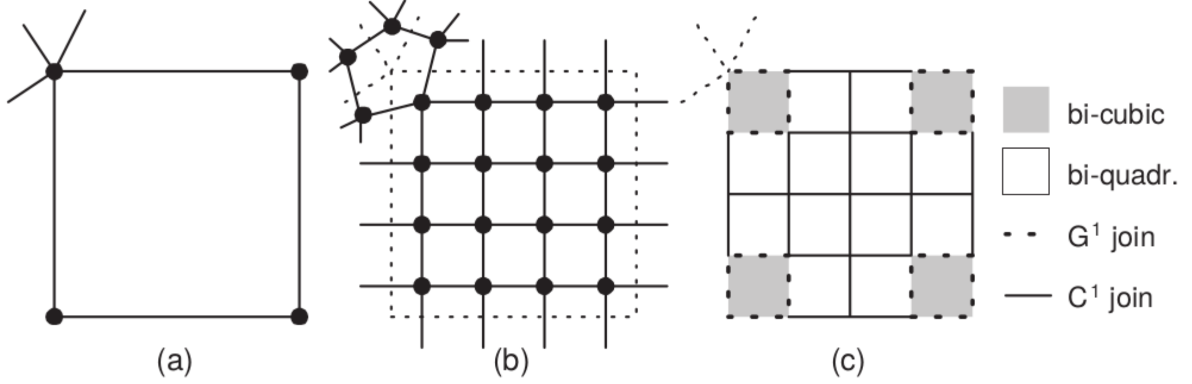


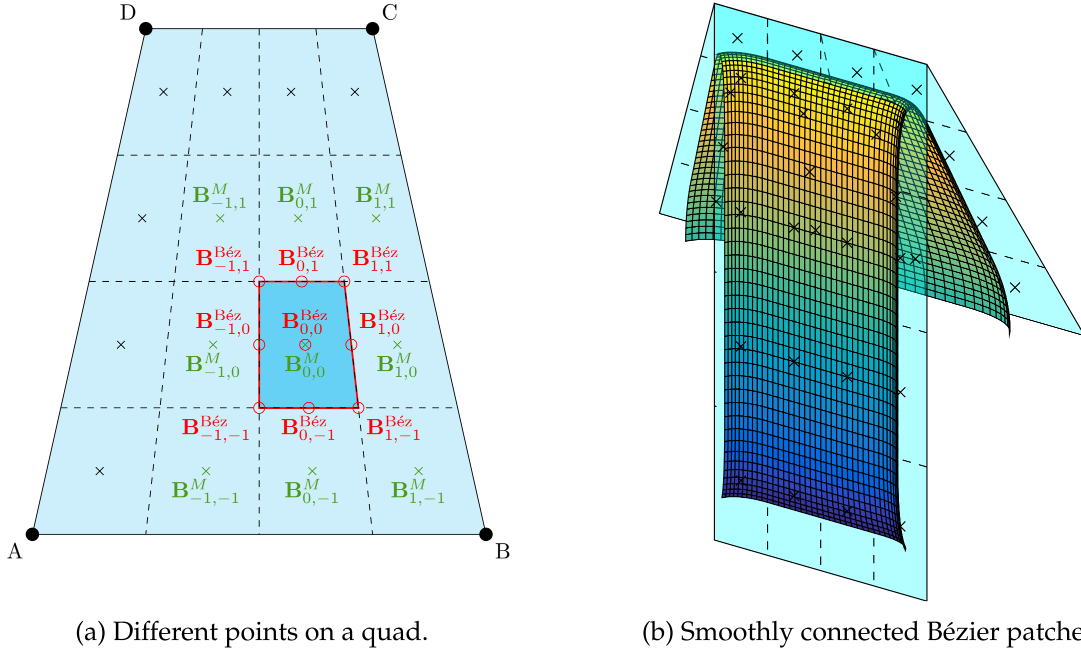
Figure 2.4.2.: The subdivision step in Peter's scheme. a): A quad in the original mesh, with edges and vertices marked out. Note that there are 5 edges connecting the top-left corner, making this a non-regular mesh corner. b): The mesh resulting from two Doo-Sabin subdivisions, denoted V_x in the text. The mesh is regular, except for around the original mesh corners with other than 4 edges. In the case of the top-left corner, the 5 edges result in a pentagon. c): The resulting Bézier patch structure. The surface is at least G^1 continous everywhere. Figure from [13].

Step 2: Bézier Patch Creation

In this step, we create one Bézier patch for every vertex in the double refined mesh V_x.

To begin with, we can recognize that each original quad now has a 4×4 grid of vertices, where the 4 vertices along each original quad edge match the 4 vertices on the neighbouring quad (see Figure 2.4.2b). From this, we see that most of the cells will be locally in a regular grid, in the sense that it can be seen as the center grid point in a 3×3 two-dimensional structured grid. For illustration, we can look at Figure 2.4.3a, where this 4×4 grid of vertices is marked with black or green crosses. Here, the green crosses illustrate one such local neighbourhood, where the green point in the middle of the darker patch $\mathbf{B}^M_{0,0}$ can be seen as the centre point of the 3×3 green points $\mathbf{B}^M_{i,j}$, $i, j \in \{-1, 0, 1\}$, around it (although the global mesh might look very different further away).

Now, for any of these local neighbourhoods of 3×3 points $\mathbf{B}^M_{i,j}$, $i, j \in \{-1, 0, 1\}$, we can now interpret them as the control points for a biquadratic (that is, second-order) tensor product B-spline surface (meaning with uniform knots). If a neighbouring point is also in a locally regular 3×3 point neighbourhood, we could then extend the B-spline surface to this local neighbourhood by adding them as control points in that direction, using some extra knots. This way, we could build up a network of patches using all the points that are also locally in a regular grid (and their neighbours in their own local 3×3 grid). This would results in a C^1 continous surface around all vertices where the mesh is locally regular. Now, this surface can also be represented by a network of a biquadratic tensor product Bézier surface patches (see Figure 2.4.2c for the patch structure), which we will use to describe the whole surface. This means, around a vertex $\mathbf{B}^M_{0,0}$, we create Bézier surface, with 3×3 control points for each patch, that we choose to call $\mathbf{B}^{\text{Béz}}_{i,j}$ $(i, j \in \{-1, 0, 1\})$. If we do this, the 3×3 Bézier control points $\mathbf{B}^{\text{Béz}}_{i,j}$ will lie at positions



(a) Different points on a quad.

(b) Smoothly connected Bézier patches.

Figure 2.4.3.: (a): The different points used in Peter's scheme. For the quad in the picture with vertices A, B, C, D in M, the refined mesh vertices in V_x created by two Doo-Sabin refinements are marked out with crosses. The extent of each of their Bézier surface patches is also marked on the quad with dotted lines (see Figure 2.4.2b,c for comparison). Additionally, for the highlighted patch, the tensor product Bézier surface points $\mathbf{B}^{\text{Béz}}_{i,j}$ are marked with red circles, with the corresponding refined mesh points $\mathbf{B}^M_{i,j}$ being the green crosses. (b): The resulting smoothly connected surface after creating Bézier surface patches. For visibility, two quads have been cut-out, although the surface is part of a bigger figure. The refined points in V_x are again marked with crosses.

in-between the center vertex $\mathbf{B}^M_{0,0}$ and the 3×3 local grid vertices $\mathbf{B}^M_{i,j}$, as shown in [1]:

$$\mathbf{B}^{\text{Béz}}_{i,j} = \frac{1}{4}\left(\mathbf{B}^M_{i,j} + \mathbf{B}^M_{i,0} + \mathbf{B}^M_{0,j} + \mathbf{B}^M_{0,0}\right) \tag{2.4.10}$$

or, writing it out explicitly for $i, j \in \{-1, 0, 1\}$,

$$\begin{aligned}
\mathbf{B}^{\text{Béz}}_{-1,1} &= \tfrac{1}{4}\left(\mathbf{B}^M_{-1,1} + \mathbf{B}^M_{-1,0} + \mathbf{B}^M_{0,1} + \mathbf{B}^M_{0,0}\right) & \mathbf{B}^{\text{Béz}}_{0,1} &= \tfrac{1}{2}\left(\mathbf{B}^M_{0,1} + \mathbf{B}^M_{0,0}\right) & \mathbf{B}^{\text{Béz}}_{1,1} &= \tfrac{1}{4}\left(\mathbf{B}^M_{1,1} + \mathbf{B}^M_{1,0} + \mathbf{B}^M_{0,1} + \mathbf{B}^M_{0,0}\right)\\
\mathbf{B}^{\text{Béz}}_{-1,0} &= \tfrac{1}{2}\left(\mathbf{B}^M_{-1,0} + \mathbf{B}^M_{0,0}\right) & \mathbf{B}^{\text{Béz}}_{0,0} &= \mathbf{B}^M_{0,0} & \mathbf{B}^{\text{Béz}}_{1,0} &= \tfrac{1}{2}\left(\mathbf{B}^M_{1,0} + \mathbf{B}^M_{0,0}\right)\\
\mathbf{B}^{\text{Béz}}_{-1,-1} &= \tfrac{1}{4}\left(\mathbf{B}^M_{-1,-1} + \mathbf{B}^M_{-1,0} + \mathbf{B}^M_{0,-1} + \mathbf{B}^M_{0,0}\right) & \mathbf{B}^{\text{Béz}}_{0,-1} &= \tfrac{1}{2}\left(\mathbf{B}^M_{0,-1} + \mathbf{B}^M_{0,0}\right) & \mathbf{B}^{\text{Béz}}_{1,-1} &= \tfrac{1}{4}\left(\mathbf{B}^M_{1,-1} + \mathbf{B}^M_{1,0} + \mathbf{B}^M_{0,-1} + \mathbf{B}^M_{0,0}\right)
\end{aligned}$$

The creation of these different types of points is also illustrated in Figure 2.4.3a.

To summarize, we have thus just created a surface around all the vertices that are locally in a regular mesh. The exceptions that cannot be placed as such, are now the points residing on the corners of the original quads, since any number of quads may be meeting there. For n quads sharing a corner vertex, the refinement steps will create a polygon with n sides, as mentioned above. If $n \neq 4$, the vertices of these corner polygons cannot be placed in a locally regular mesh, and we cannot apply the previous technique – see for example the upper-left corner of Figure 2.4.2b.

However, we can still create a Bézier patch which connects smoothly to the locally regular points. To do this, we create a bicubic Bézier patch (of polynomial order 3, one higher than around the regular points, meaning it has 4×4 control points) for every vertex in the corner polygon. We then evaluate how the surface position and normal direction depends on the Bézier control points along all edges of the patches. In order to have a smooth connection, we want these to match, and thus, we can produce constraints for the positions of the Bézier control points. Since the surfaces are defined at each point as linear combinations of the Bézier control points (which in turn are linear combinations of the vertices in the doubly refined mesh V_x), the constraints result in an underdetermined system of linear equations for all the Bézier control points on the bicubic Bézier patches on the vertices in the doubly refined mesh V_x.

As the resulting formulae for the locations of the Bézier control points are rather lengthy and complex, we refer to the original paper (ref. [1]). Here, it is also proven that the surfaces have an overall G^1 connectivity. A cut-out sample can be seen in Figure 2.4.3b.

To summarize, Peters' scheme is a mathematical algorithm for creating biquadratic and bicubic Bézier patches that join with G^1 continuity, from a mesh of polygons. Firstly, a set of refined mesh points is created on each polygon. Then, Bézier control points defining the patches are created as linear combinations of the vertices in this refined mesh. The G^1 continuity results from the interpretation of the refined mesh as a regular biquadratic tensor product B-spline surface, and where this is not possible, bicubic Bézier patches are constrained to join smoothly to the surrounding biquadratic patches.

2.4.3. Fairness Functional

Aside from recognising discontinuities in the first derivative (kinks), the human eye is usually able to recognise large values in curvature, that is, properties depending on the second derivatives of a surface. For example, we can have a surface that is entirely continous in the first derivative of the surface, that also has large wiggles and sharp turns, that on a small scale are smooth, but do not seem so on the large scale. To give a measure for this, one can for example use the bending energy of a thin plate, which depends on the squared magnitude of the second derivatives. In the special case of a rectangular patch, parametrized by $u, v \in [0, 1]$, we can easily integrate this over the whole area. The energy can then be expressed as a functional of $\mathbf{S}$,

$$E_{patch}[\mathbf{S}] = \int_0^1 \int_0^1 \left(\left(\frac{\partial^2}{\partial u^2} \mathbf{S}(u, v) \right)^2 + 2 \left(\frac{\partial^2}{\partial u \partial v} \mathbf{S}(u, v) \right)^2 + \left(\frac{\partial^2}{\partial v^2} \mathbf{S}(u, v) \right)^2 \right) \mathrm{d}u \mathrm{d}v. \quad (2.4.11)$$

One case in which this equation can be simplified is when we have explicit definitions of the surface. For example, for the rectangular Bézier tensor product surface curves in subsection 2.4.1,

we express the surface as a weighted sum of polynomials in u and v (see Equation 2.4.2 for a reminder). For a Bézier surface of order N, with $N+1$ points $\mathbf{p}_{i,j}$ in each direction, we can thus reexpress Equation 2.4.11 as coefficients times a dot product

$$E_{fair} = \sum_{i_1,j_1=1}^{N+1} \sum_{i_2,j_2=1}^{N+1} \left(c^{uu}_{(i_1,j_1),(i_2,j_2)} + 2c^{uv}_{(i_1,j_1),(i_2,j_2)} + c^{vv}_{(i_1,j_1),(i_2,j_2)} \right) \mathbf{p}_{i_1,j_1} \cdot \mathbf{p}_{i_2,j_2} \tag{2.4.12}$$

where the coeficcients $c^{st}_{(i_1,j_1),(i_2,j_2)}$ are obtained by of multiplying out the squares in the energy functional, inserting the definitions of the Bernstein polynimials $b^N_i(u)$:

$$c^{st}_{(i_1,j_1),(i_2,j_2)} = \int_0^1 \int_0^1 \frac{\partial^2}{\partial s \partial t} \left[b^N_{i_1}(s) b^N_{j_1}(t) \right] \frac{\partial^2}{\partial s \partial t} \left[b^N_{i_2}(s) b^N_{j_2}(t) \right] \mathrm{d}s\mathrm{d}t \tag{2.4.13}$$

Although this is already very much simpler to evaluate, given one already has the coefficients $c_{(i_1,j_1),(i_2,j_2)}$, we can do a further reformulation by reindexing (i,j) to k where k goes from 1 to $(N+1) \times (N+1)$. By forming a vector P_{patch} containg all the points as 3-dimensional entries, we can then reexpress Equation 2.4.12 as a vector-matrix-vector product:

$$\begin{aligned} &\sum_{i_1,j_1=1}^{N+1} \sum_{i_2,j_2=1}^{N+1} c_{(i_1,j_1),(i_2,j_2)} \mathbf{p}_{i_1,j_1} \cdot \mathbf{p}_{i_2,j_2} = \\ =&\sum_{k_1=1}^{(N+1)^2} \sum_{k_2=1}^{(N+1)^2} c_{k_1,k_2} \mathbf{p}_{k_1} \cdot \mathbf{p}_{k_2} = \\ =&P^T_{patch} C^{fair}_{patch} P_{patch} \end{aligned}$$

Now, since we know the original expression of E_{fair} and the contribution stemming from each point $\mathbf{p}_{i,j}$, we know the matrix C^{fair}_{patch} to be positive semi-definite and symmetric. Hence, it should be diagonalizable by an orthogonal matrix, and have only nonnegative eigenvalues. Thus, expressing $C^{fair}_{patch} = S^T \Lambda S$, we end up with:

$$\begin{aligned} E_{fair} =& P^T_{patch} C^{fair}_{patch} P_{patch} = \\ =& P^T_{patch} S^T \Lambda S P_{patch} = \\ =& P^T_{patch} S^T \Lambda^{\frac{T}{2}} \Lambda^{\frac{1}{2}} S P_{patch} = \\ =& \left[\Lambda^{\frac{1}{2}} S P_{patch} \right]^T \left[\Lambda^{\frac{1}{2}} S P_{patch} \right] = \\ =& \left\| \Lambda^{\frac{1}{2}} S P_{patch} \right\|^2 \end{aligned} \tag{2.4.14}$$

which allows us to express and calculate the thin plate energy using a matrix-vector product and a squared norm.

Since surfaces with low curvatures and therefore softer curves are typically considered more pleasing to the eye, the functional in Equation 2.4.11 is typically named *fairness functional*.

2.5. Least-Squares Fitting of Parametrized Surfaces

In order to make a surface adhere as closely as possible to our desired form, some fitting is usually required. This typically involves varying some parameters in order to minimize some error. As minimizing the error squared of a function – for example distance between a calculated point and its desired location – is an important building block in many practical applications, extensive literature can be found regarding this. The treatment is especially well described for when the function depends linearly on its input (see for example [23]). This is called *linear least-squares fitting*. In this section, some selected subtopics relevant to the fitting of parametric surfaces are discussed.

2.5.1. Fitting Problem: Parametric Surfaces

In order to fit a Bézier surface or NURBS surface to a set of datapoints with fixed locations and parameters on this surface, linear least-squares fitting can be applied, using an approach of mapping the parameters of the datapoints to control points on the surface. The base of the approach is using either Bernstein polynomials (Bézier curves) or B–spline basis functions (NURBS) to evaluate how control points should be combined for those surface parameters, and obtain this as a linear combination on the control points. That means, that from the parameters (u, v) of each data point on the surface, we get a set of coefficients on the control points that define the surface. Expressing this as a sum, we have for the point $\mathbf{d}_k$ with the parameters (u_k, v_k) on the surface defined by the $N \times M$ control points $\mathbf{p}_{i,j}$:

$$\mathbf{d}_k = \sum_{i,j=1}^{N,M} c_{i,j}(u_k, v_k)\mathbf{p}_{i,j} \tag{2.5.1}$$

where $c_{i,j}(u, v)$ are the coefficients calculated on control point $\mathbf{p}_{i,j}$ from the parameters (u, v), using the definition of the parametric curve (see subsection 2.4.1). Realizing that the matrix indexing i, j can be flattened to a vector index $p = 1, 2, ..., S$, where $S = N \times M$, mapping to i and j, we can express this as:

$$\mathbf{d}_k = \sum_{p=1}^{S} c_p(u_k, v_k)\mathbf{p}_p \equiv \mathbf{c}(u_k, v_k)P \tag{2.5.2}$$

where in the last equality, we have expressed the control points as a vector-matrix product, where the $S \times 3$ matrix P has the pth row as the position of the pth control points as a 3D row-vector, and $\mathbf{c}(u_k, v_k)$ is the row vector whose entries are the coefficients $c_p(u_k, v_k)$ on the control points. Now, again realizing that for Q datapoints $\mathbf{d}_k$ we can view them as the columns in a $Q \times 3$ matrix D, and create an analogous matrix $C(\mathbf{u}, \mathbf{v})$ with the kth containing the coefficients calculated from the parameters (u_k, v_k) of datapoint $\mathbf{d}_k$, resulting in the matrix equation:

$$D = C(\mathbf{u}, \mathbf{v})P \tag{2.5.3}$$

Now, in the case that we have a set of datapoints that we manage to parametrize to the surface, we can calculate the positions of the control points by calculating the control point matrix C using these parameters, and then solving the system for P:

$$P = [C(\mathbf{u}, \mathbf{v})]^{-1} D$$

However, this requires that there be a solution, which typically relies on the specific number of datapoints being exactly equal to the number of control points, and on the fact that here we also interpolate the points exactly. In most cases there are many more datapoints, and we try to find the P that minimizes the least-squares error:

$$\|D - C(\mathbf{u}, \mathbf{v})P\|^2 \tag{2.5.4}$$

This again is a central problem in computing, and can be solved by many standard libraries with excellent performance.

2.5.2. Fitting Problem: Fairness

As discussed in subsection 2.4.3, it can also be important to have surface patches that are not only continous and smooth, but also have a low curvature. Fortunately, this is relatively simple to incorporate in the above Least-Squares formulation. Starting from the end result of Equation 2.4.14, we see that the minimisation of the fairness functional can already be expressed as the minimisation of a squared norm of a matrix-vector product. In order to combine this with the fitting to a surface, we begin by writing it in the form of Equation 2.5.4. To do that we simply replace D with a 3-dimensional 0-vector (thus a 3-column 0-matrix), as we want to get the matrix-vector product as close to 0 as possible, and minimize

$$\begin{aligned} &\left\|0 - \Lambda^{\frac{1}{2}} S P\right\|^2 = \\ &= \left\|0 - C^{fair} P\right\|^2, \end{aligned}$$

where $C^{fair} = \Lambda^{\frac{1}{2}} S$ has the same number of columns as $C(\mathbf{u}, \mathbf{v})$ in Equation 2.5.4. To minimize both equations simultaneously, we notice that both terms are positive, and we can add them together, giving a weight $\lambda \geq 0$ to the fairness term:

$$\|D - C(\mathbf{u}, \mathbf{v})P\|^2 + \lambda \left\|0 - C^{fair} P\right\|^2 \tag{2.5.5}$$

Noticing that the terms in the sums each depend on one row of D or 0 and $C(\mathbf{u}, \mathbf{v})$ or $C^{fair}P$ respectively, we see that we could also join the equations into one norm, by concatenating D and 0, and $C(\mathbf{u}, \mathbf{v})$ and $C^{fair}P$ vertically, giving a minimization of

$$\begin{aligned} &\left\| \begin{pmatrix} D \\ 0 \end{pmatrix} - \begin{pmatrix} C(\mathbf{u}, \mathbf{v}) \\ \lambda C^{fair} \end{pmatrix} P \right\|^2 = \\ &= \left\| \tilde{D} - \tilde{C}(\mathbf{u}, \mathbf{v}) P \right\|^2 \end{aligned} \tag{2.5.6}$$

where we have denoted the final concatenated matrices with $\tilde{D}$ and $\tilde{C}(\mathbf{u}, \mathbf{v})$ respectively.

2.5.3. Fitting Problem: Peters' Scheme

One of the major drawbacks with the approach above is that it is typically very complicated to impose smoothness constraints on structures of multiple patches, something that is essential for representing more complicated geometries. One of the ways of avoiding this is to use an

existing scheme for creating a smooth surface, such as the scheme of Peters [1, 13], which we introduced in subsection 2.4.2. Here, we create smoothly connected square Bézier patches by letting their control points be linear combinations of vertices in a refined control mesh V_x. We ensure smoothness by the way we calculate these coefficients on the vertices in V_x – we use the information on their local positions in the mesh to determine which neighbouring vertices they should be influenced by. Once more, we can express this mathematically with $\mathbf{p}_p$ denoting the pth control point, and $\mathbf{v}_l$ being the lth refined control mesh vertex:

$$\mathbf{p}_p = \sum_l c_{p,l}^{PS} \mathbf{v}_l \tag{2.5.7}$$

with $c_{p,l}^{PS}$ being the coefficient for the pth Bézier control point on the lth refined control mesh vertex as obtained from Peters' scheme. Analogous to equations 2.5.2 and 2.5.3, we can extend this to all Bézier control points on all patches, and cast it in a matrix form to obtain:

$$P = C^{PS} V \tag{2.5.8}$$

where P as before is the Bézier control point position matrix, C^{PS} is the matrix of coefficients for the Bézier control points from on the refined control mesh vertices of V_x, and V is the matrix of the positions of these vertices.

Finally, we can now substitute this into our equations 2.5.3 and 2.5.4 above, to solve the system:

$$\tilde{D} = \tilde{C}(\mathbf{u}, \mathbf{v}) P = \tilde{C}(\mathbf{u}, \mathbf{v}) C^{PS} V \tag{2.5.9}$$

or minimize the least squares error for the only unknowns in this case, the positions of the refined control mesh vertices V:

$$\left\| \tilde{D} - \left[\tilde{C}(\mathbf{u}, \mathbf{v}) C^{PS} \right] V \right\|^2 \tag{2.5.10}$$

To summarize, using the standard methods of minimisation of least-squares errors, we compute a set of points, which when used in Peters' scheme produces a surface with the smallest error to a set of datapoints. By including a fairness term, we also reduce the curvature of the surface.

For the computation of $C(\mathbf{u}, \mathbf{v})$ (and therefore of $\tilde{C}(\mathbf{u}, \mathbf{v})$), we require the parametrisations of these datapoints on the surface, and in order to obtain C^{PS}, we also need to know how the patches connect with each other.

It is worth noting that all matrices $C(\mathbf{u}, \mathbf{v})$, C^{fair} and C^{PS} are, although likely very large for large amounts of datapoints and patches, also very sparse. This fact comes from that both $C(\mathbf{u}, \mathbf{v})$ and C^{fair} will only contain coefficients on one patch, and that the algorithms for computing C^{PS} only use information of a few vertices around the patch. Thus, the final combined coefficient matrix $\tilde{C}(\mathbf{u}, \mathbf{v}) C^{PS}$ will also have high sparsity. For solving the minimisation problem of Equation 2.5.10, we will therefore be able to use specialised, fast solvers for sparse systems, with lower complexity, and therefore much lower runtime.

3. Implementation

Where the previous chapter introduced the fundamental concepts and ideas used in this project, this chapter will describe how these are actually put to use. In the text below, we present details of all four sections of the pipeline, starting with an overview.

3.1. Overview

Using the different concepts introduced in the previous chapters, we built a software pipeline starting from a CAD design via topology optimization back to CAD. We start from a CAD geometry input, obtain a voxelized input for the topology optimizer, extract the surface from an optimized voxel grid, and then fit a network of smooth NURBS surface patches to it.

A graphical overview over the pipeline implementation is shown in Figure 3.1.1. Here, the different steps and transformations applied on the geometry are described. In the following sections, we will look at these implementation details more closely.

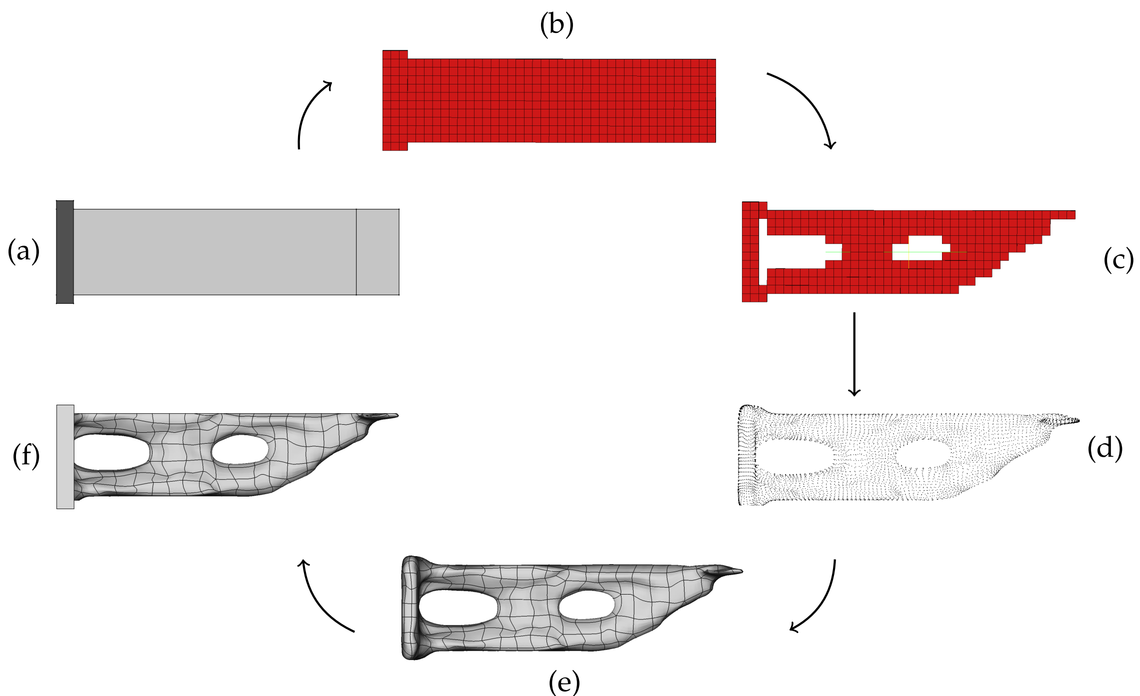


Figure 3.1.1.: Starting off with an input geometry (a), the first step to follow is voxelization, in which we convert the CAD geometry into three-dimensional voxel grid data (b). On this grid, topology optimization is done (c), after which the points for surface fitting are computed (d). This is followed by construction of NURBS surfaces through these points (e). Finally, post-processing with fixtures and optimization domain limits results in the final geometry (f).

3.2. From CAD Model to Voxel Representation

Converting the CAD input to ToPy input along with all the boundary conditions is the first problem that needs to be addressed in our pipeline. The procedure has been described in four sections: the first section explains the logic behind specifying boundary conditions using a CAD software. Subsequent sections then talk about what the code does with this CAD input. The second section explains the extraction of different boundaries from the CAD geometry with extensive use of the OpenCascade C++ library. This is followed by two more sections on voxelization and writing of the output file. Finally, the last section explains the input file format for ToPy, and describes its operation. Figure 3.2.1 shows an overview of the structure of the code used for this part of the pipeline.

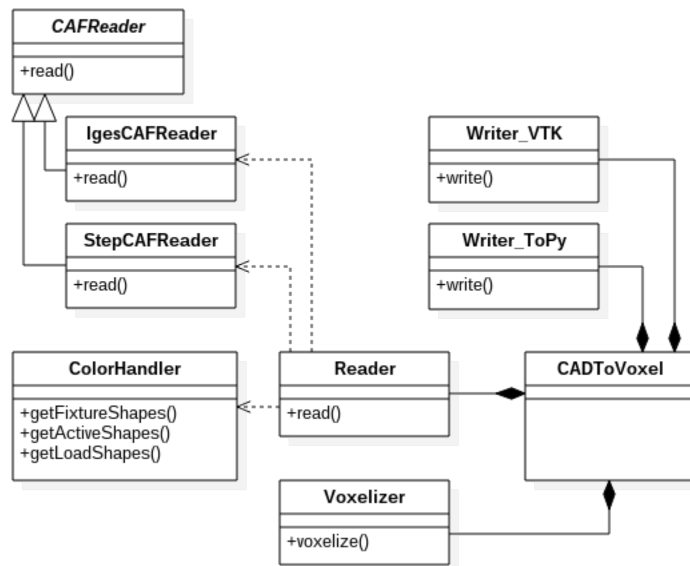


Figure 3.2.1.: UML diagram of the pipeline from CAD to a voxelized output. Many sections of the code use data-structures and algorithms from OpenCascade[24].

3.2.1. Specification of Boundary Conditions for the Input Geometry

As pointed out in the 2.1, STEP and IGES files can store color information for each face. This is the attribute that is used to specify the identity of the face, and can be modified using standard CAD design software such as FreeCAD (see [25]). When analysing a structural problem, a designer typically needs to provide information on up to five parameters:

- **Optimized domain**: This refers to the main geometry that needs to be optimised. These faces are simply colored absolute white, i.e. $(255, 255, 255)$.
- **Fixture faces**: These faces of the geometry are meant to be fixed in space, and undergo zero displacement. These faces are colored absolute red, i.e. if the color space of red ranges in $[0, 255]$, then the face is assigned the extremum 255. The blue and green color components

are set to 0. Thus, the color array is set to $(255, 0, 0)$. These are also stored in the file holding the optimized domain.

- **Load faces and load value**: These faces bear the forces acting on the body. Loads are three-dimensional, and can take both positive and negative values depending on their direction. To accommodate all possibilities, each color component (i.e. red, blue and green) is assigned a direction (x, y, and z). The range $(0, 255)$ is split in two: $(0, 127]$ representing the negative force range and $[128, 255)$ representing the positive force range. The color is then offset by -127.

  Internally, OpenCascade transforms the colors from range $(0, 255)$ to $(0, 1)$ which we then shift to $(-0.5, 0.5)$. Of course, force values may also lie outside this range. The user then needs to scale the required force to fit in the color range, and provide a scaling factor as an input.

  For example, to assign a force value of $(-200.5, 172.0, -10.75)$:

  Table 3.1.: Fitting a force value to a color

Original vector	Scaling factor	OpenCascade values	Final color
-200.5	401	-0.49	001
$+172.0$	401	$+0.43$	235
-10.75	401	-0.03	120

  The load faces are given in a specified input file such that for example inner loads can be described.

- **Non-changing geometry**: Sometimes, certain parts of the geometry cannot accommodate changes. This could be for several reasons, for example, due to compatibility issues with other components. To define the geometry accurately and independently of the main geometry, it is provided in a distinguished file.

- **Solution limits:** The designer may be interested in restricting the solution to a certain region in space. This can be done by creating CAD geometry enclosing this region. Since this region could be different from the main geometry, it is specified through an additional file.

Figure 3.2.2 illustrates in more detail how the inputs needs to be defined to describe the constraint geometry.

3.2.2. Face Extraction and Categorization

Using the input geometry, the first task to parse the geometry, while extracting the faces and sorting them out based on their type. In the software, this is done as follows:

1. An instance of `Reader` is created with the CAD source directory and file name as input. `Reader` wraps the OpenCascade classes for reading STEP and IGES files into a single class,

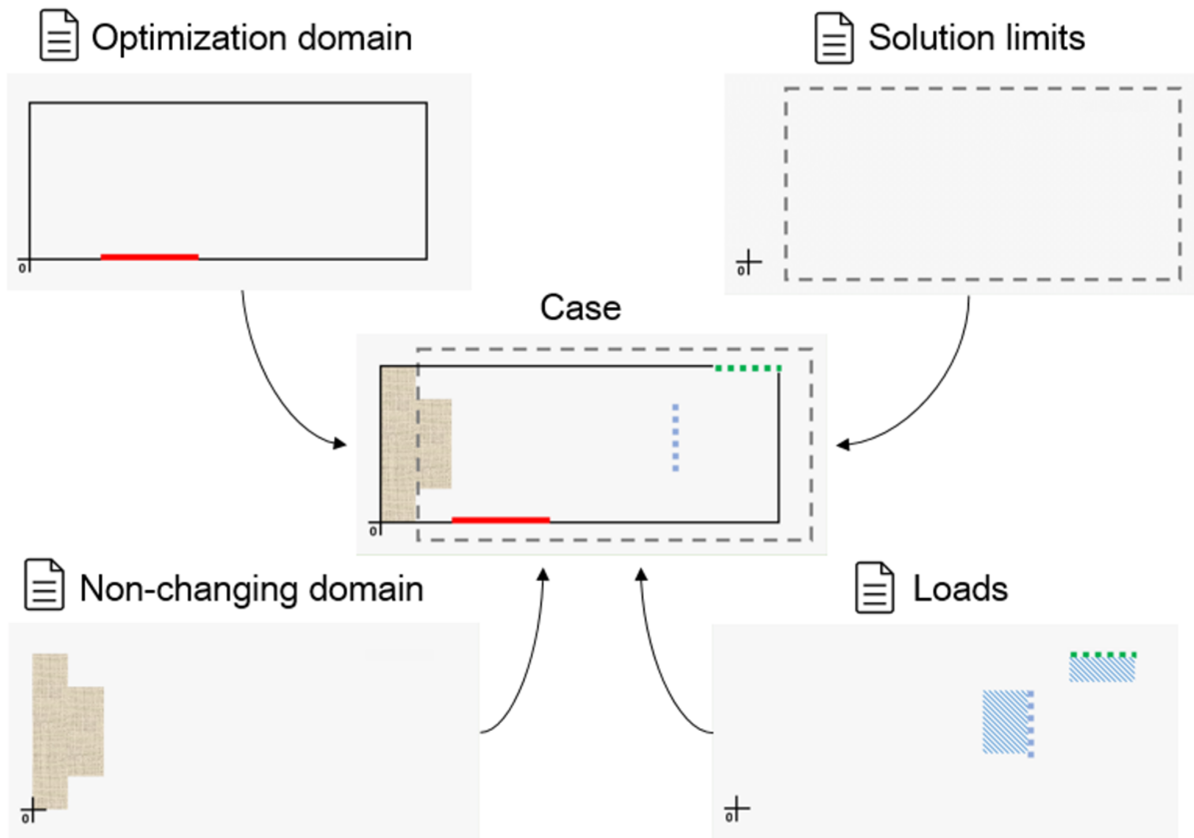


Figure 3.2.2.: Assume the input file is called `input_file`. The optimization domain(top left) describes the main geometry and has to be provided in files `input_file.step` and `input_file.iges`. Here, files of type `.step` store the structural description while `.iges` hold the color information. Also the fixture faces are stored in these files. The solution limits are given in a file `input_file_toOptimize.step` (top right). Similarly, the non-changing domain stored as `input_file_Fixed.step` (bottom left). Finally, also the loads are specified in a seperate input file named `input_file_load.step` (bottom right).

reads the two files, and holds them in two handles. Also `Reader` instances are created for the non-changing domain and load file input.

2. The `ColorHandler` class takes over the handles from `Reader`. `ColorHandler` provides methods, each of which returns a list of faces (see Figure 3.2.1). Depending on which method is called the returned list contains groups of fixtures, loads, passive faces, or all faces of the body.

3. Each of the methods mentioned above internally calls the hidden function `findColoredFaces ()`. This method takes as input a color, and returns all faces that match it. It also takes as input a boolean variable `isLoadSeeked` - if true, then the function returns all faces with load on them, and also a vector of the corresponding loads.

4. The load vector is then scaled with respect to the scaling factor provided as input by the user.

These face lists are then transferred to the voxelization pipeline.

3.2.3. Voxelization

As pointed out in section 2.2, a very common formulation of domains for topology optimization is through specifying regions as either filled or empty. The minimum compliance problem is then solved on a discretized grid; the most common one is a volume raster in the form of cubes, or so called *voxels*. Thus, the next step is to render the geometry with a 3D raster of voxels.

From the face extraction pipeline, the geometry shape and faces for each boundary condition type are stored in OpenCascade through the internal data type `TopoDSShape`. The `voxelise` function is called internally for the complete geometry and each face separately (see Figure 3.2.1) since boundary conditions may consist of more than one face. The voxelisation is then performed as follows:

1. In order to combine the 3D voxel raster consistently, a bounding box is introduced. This allows keeping the coordinate system uniform, making voxel numbers consistent between different voxelization types (faces and shapes).

2. In each dimension, $2^n \cdot l_d$ voxels are created, where n is the user specified refinement level and l_d is the size of the bounding box in the respective dimension d.

3. Voxelization is performed with the OpenCascade `Voxel_FastConverter.hxx` class creating a `VoxelShape`.

Consequentially, a 3D boolean voxel raster is created for each type.

3.3. Topology Optimization of Voxel Data

The next step in the pipeline is performing topology optimization on the voxelized geometry (2.2). In line with our policy to use open-source software, we decided to adopt *ToPy*, a free-licensed topology optimizer.

3.3.1. Topology Optimization Tool ToPy

ToPy [26] is a python library/program, written by William Hunter and documented in [7], implementing the SIMP model and method (subsection 2.2.1). It is based on the 99-line Matlab code by Sigmund's for minimum compliance [10]. The program can optimize for minimum compliance, heat conduction and mechanism synthesis — in two- and three-dimensions. It uses highly optimized open source python libraries such as Pysparse [27] and Numpy [28], leading to improved speed, portability and scalability.

We use ToPy as a black-box topology optimizer. This means, we launch the program with an input file based on our scenario and let ToPy run. The output of ToPy is then plugged in to the next module. The intention is to create separate modules to be able to easily switch them later on if need be.

3.3.2. Construction of ToPy Input File

The topology optimization library ToPy works with ToPy Problem Definition files (*.tpd*, see Figure 3.3.1). The file starts with a header and internal parameters; they are used to define the domain size and to steer the topology optimization using grey-scale filters. Geometry shape and boundary conditions are passed specifying the type (load, active, passive, force) followed by lists of element indexes. The numbering is defined as follows:

- Voxel Zero lies at the bounding box corner with the minimal coordinate values
- Neighboring elements in y-direction are then numbered 1,2,3, ...
- After reaching the domain bounds the numbering continues with the next neighboring row in x-direction.
- after finishing the first "plate" the numbering continues with the next neighboring layer in z-direction.

```
[ToPy Problem Definition File v2007]
PROB_TYPE:   comp
PROB_NAME:   topy
...
Q_CON       : 1
Q_MAX       : 5
ACTV_ELEM: 0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12;
...
LOAD_VALU_Y: -1@100
```

Figure 3.3.1.: Sample ToPy input file: Active, fixture or force elements are passed as lists of voxel numbers; load strengths are specified with corresponding value.

It must be stated that ToPy modifies the coordinate system: flips the Y-coordinate and interchanges the X and Z coordinate.

Topy Problem Definition files are created invoking the bash script `CADTopOpt.sh` (see section 3.2.2). The main file `CADToVoxel` invokes a function `write`, that creates a *.tpd* file (see Figure 3.2.1). The functionality is implemented in a class `Writer_ToPy`. `Writer_ToPy` opens an output file, writes the *.tpd* header and grey-scale filters. In the next step it deals with the voxelized shapes: active (filled voxels subject to optimization), passive (filled voxels not subject to optimization), fixture and load elements are written in the file by writing lists of element node numbers. This file is then ready to be used; the bash script `CADTopOpt.sh` invokes ToPy to perform the topology optimization.

3.3.3. Results of Topology Optimization

A sample of ToPy's optimization process can be seen in figure 3.3.2. Here, a voxelized star was given as input with its end points as fixtures, and a central load normal to the star's plane. The optimization process "cuts" away unnecessary material, returning an optimally stiff structure for the specified volume fraction.

It is immediately evident that the voxel-format output of ToPy is a dead-end for designers in terms of modification and manufacturing. Thus, the next part of the workflow is fitting a smooth surface through this discrete data and arrive at a CAD format of the optimized geometry.

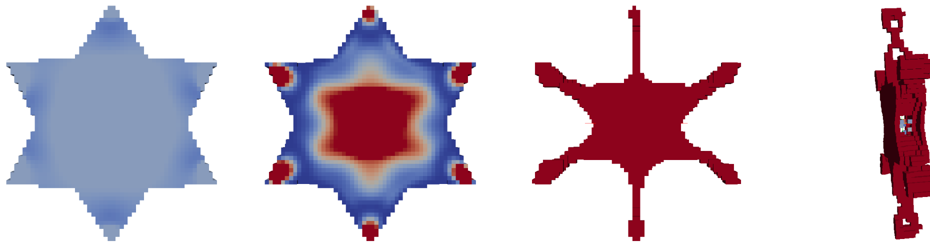


Figure 3.3.2.: Topology Optimization by ToPy [26], with minimum compliance. *From left to right*: increasing number of SIMP iterations until convergence. The star-shaped structure was given by an STL-file which was processed into input readable by ToPy, with fixtures in the corners, and a load in the middle. Throughout the SIMP iterations, one can see how material from the less dense regions (blue) is concentrated into denser regions (red) that carry the load. The last picture gives a rotated view, to illustrate how material has been eliminated even from the inside of the star.

3.4. From Voxel Representation to Parametrized Surface Points

Surface extraction is an intermediate step between topology optimization and NURBS representation. This intermediate step is crucial, because it generates the data for NURBS surface fitting and for the underlying topology of the NURBS patches. Broadly, the process consists of two steps: Surface reconstruction and parameterization of datapoints.

3.4.1. Surface Reconstruction

In section 2.3 different surface reconstruction schemes were introduced. Here, we discuss our choice for an appropriate surface reconstruction scheme and explain necessary modifications to it.

Discussion on Surface Reconstruction Schemes

Since the surface reconstruction is just an intermediate step before our final NURBS fitting procedure, it is not sufficient to only produce a good surface approximation of our optimized topology, but additional constraints have to be kept in mind:

- NURBS have a rectangular topology, therefore our surface reconstruction should also be able to provide a surface consisting of rectangular patches.
- Peters' Scheme only covers manifold surfaces. This means that each edge must be shared by exactly two patches.

The first requirement is met by DC, while the second one is met only by MC. Nevertheless, we decided to use the DC method because its basic version already creates quads, while MC creates a mesh of triangles. This mesh can only be changed into a mesh of quads by considerably increasing the number of faces.

Our Implementation of Dual Contouring

We use the open-source language PYTHON [29] for the implementation of DC. Compared to the version described in [12] the following simplifications were applied:

- We use the simple averaging scheme described in subsection 2.3.2 instead of Equation 2.3.1, since we do not consider sharp features and cannot easily access gradient information in our algorithm.
- Since our dataset only consists of boolean values instead of real valued quantities, we locate our surface at the material/non-material interface instead of a certain isovalue.
- Our implementation does not support adaptivity or topology safety.

This leads to the following modified DC scheme:

1. Find all sign-changing edges that connect material and non-material voxels.
2. On each sign-changing edge, find the root (i.e. the interface of material and non-material voxels) using bisection. We assume our surface to lie exactly in the middle of material and non-material voxels.

3. Take the mean value of these root positions for determining the position of the newly introduced vertex.

4. Join the vertices associated with four cubes sharing a common sign-changing edge to form a quad.

Thus our procedure generates quad surfaces for boolean datasets on uniform Cartesian grids. But unlike the original DC algorithm we cannot guarantee that these are manifold surfaces (see section 2.3.2).

Obtaining Manifold Surfaces

Since we want to deduce a first estimate for the topology of the NURBS surface output, non-manifold surfaces cannot be accepted[1]. We therefore use a remeshing procedure for generating manifold surfaces out of non-manifold surfaces. Our procedure has the following steps:

1. Find all non-manifold edges by searching for edges that are connected to more than two quads.

2. For every non-manifold edge found, make a copy. Link the quads to the two resulting edges to create a manifold surface. Now, exactly two quads are connected to each edge.

3. The member vertices of the original edge and those of its copy are moved in opposite directions. This is done to separate the overlapping edges.

We illustrate the procedure of remeshing for a 2D example in Figure 3.4.1. In 3D we do not have non-manifold vertices, but edges, which have to treated. Please note that for the 3D case additional patterns come up. In 3D not only the quads which are connected to the whole non-manifold edge have to be considered, but also the quads connected to only one vertex of the edge. This also implies the introduction of new quads at other locations. For an overview over some of the possible patterns in 3D see Figure 3.4.2.

3.4.2. Parametrization of Datapoints

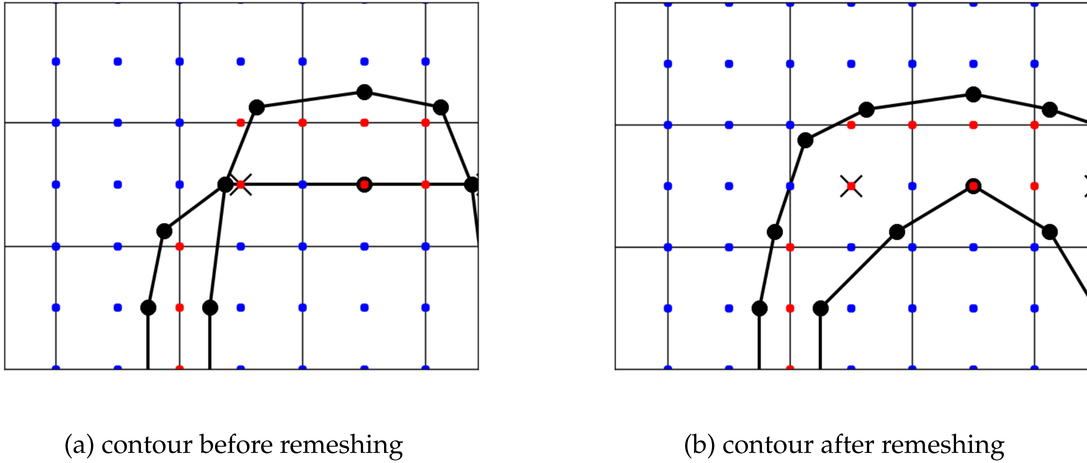
In addition to the reconstructed surface we need the following information for the least squares fit:

- Which NURBS-patch does each datapoint of the reconstructed surface belong to?
- What are the values of u, v parameters of the datapoint on the patch?

Two-scale Dual Contouring

Before we can distribute datapoints to NURBS-patches, we first have to find out how these patches look like. Since we want to have as few patches as possible we do not simply turn every quad from the surface reconstruction into a patch. Instead we try to find a surface with as few

[1]Surfaces consisting of smoothly connected NURBS patches are always manifold surfaces. Therefore we cannot start with a non-manifold surface and assume we will end up with a manifold surface.



(a) contour before remeshing

(b) contour after remeshing

Figure 3.4.1.: Illustration of the remeshing process. Blue dots denote outer voxels, red dots inner voxels. Crosses denote voxels considered for resolving ambiguities. We first copy the non-manifold vertex and then move the resulting pair of vertices in opposite directions. The direction is determined from the gradient at the datapoint with the cross, which goes in the direction of the blue dots (from inside to outside). Please note that we can only estimate the gradient if additional information is available in the middle of the cube which contains the non-manifold vertex. Otherwise it is not possible to resolve the ambiguity in a proper way and therefore we cannot eliminate the non-manifold vertex.

quads as possible, while keeping the same topology as our initially reconstructed surface. This coarse surface will be assumed to be the patch distribution for the later steps.

Therefore, in our algorithm we are reconstructing the surface on two different scales: a coarse and a fine scale. The coarse scale data is deduced from the fine scale data by recursively applying the following algorithm:

1. Combine sets of eight connected voxels with egde-length a into a coarse voxel with edge-length $2a$ (see Figure 3.4.3a).

2. Decide whether the new voxel resembles an inner ($= 1$) or outer ($= 0$) voxel. This is done by taking the mean value of the contributing eight voxels. If the mean value is above a certain threshold t, the new voxel is considered as an inner voxel. We picked $t = \frac{1}{8}$, i.e. if at least two voxels out of eight are inner voxels, the resulting voxel is considered being an inner voxel.

3. For the resolution of non-manifold edges, additional coarse voxel grids are generated. These grids are shifted by edge-length a (see Figure 3.4.3b).

One iteration of this coarsening scheme is referred to as one *coarsening step*. Applying this scheme recursively allows higher coarsening and results in multiple coarsening steps.

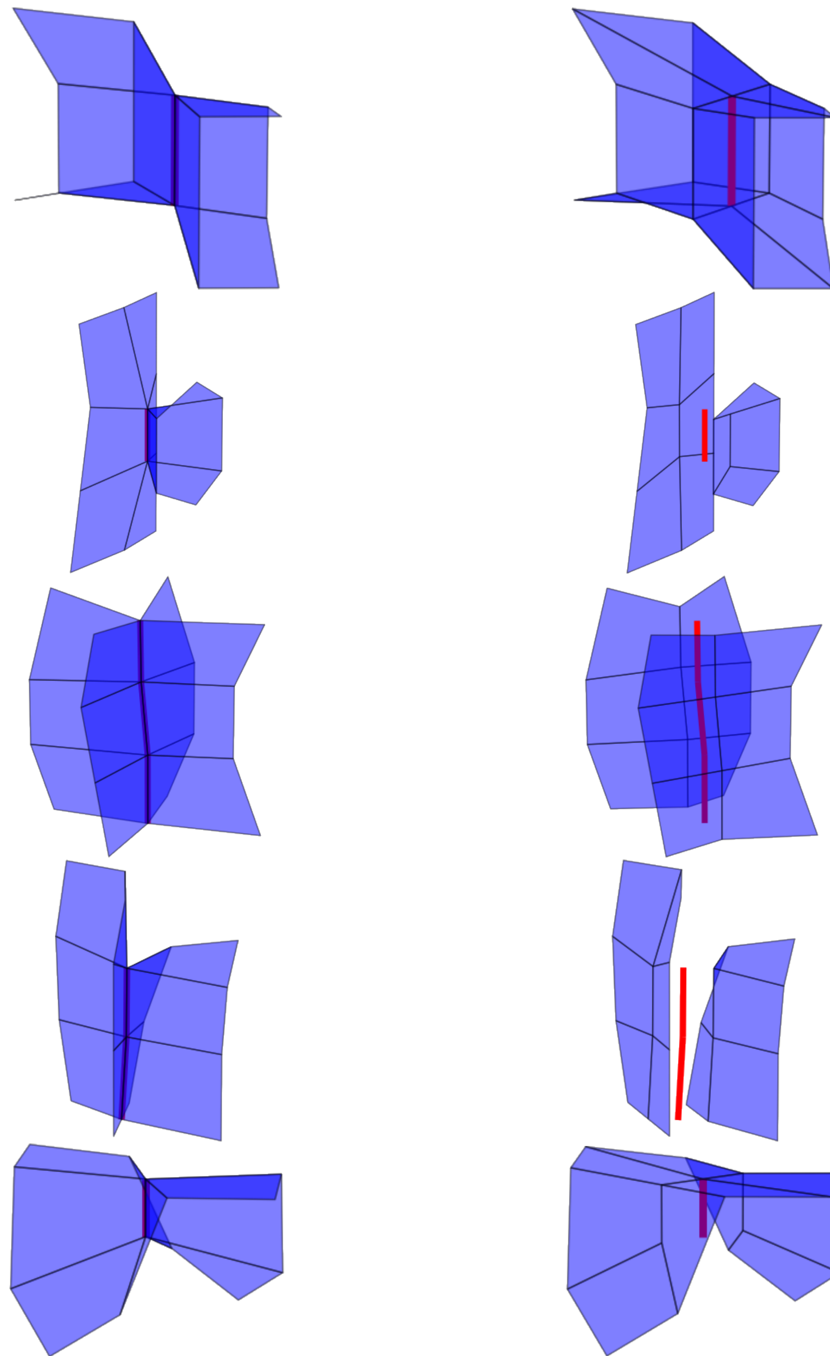
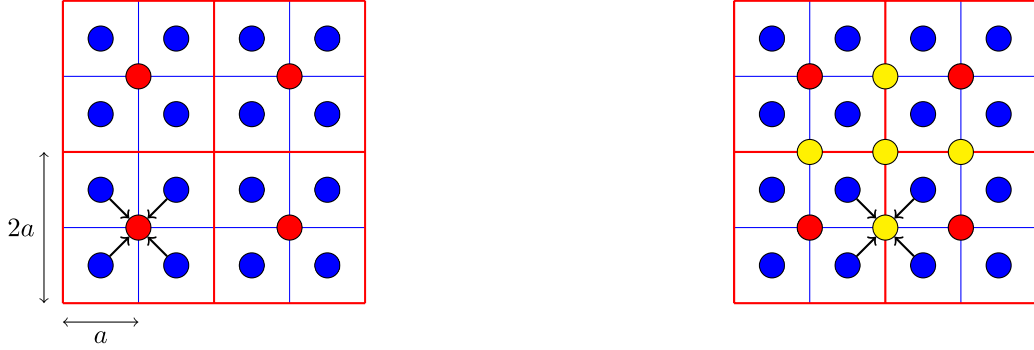


Figure 3.4.2.: Different patterns before (left) and after (right) remeshing. Depending on the neighbourhood of the non-manifold edge (red) different patterns are applied. The edge is always copied. The resulting two edges are moved in opposite directions. A non-manifold edge is connected to outside-outside (1.row), inside-inside (2.row), two other manifold edges (3.row), inside and another manifold edge (4.row), inside-outside (5.row). In the first and last row copying the edge is not sufficient; new quads have to be introduced. Please note that this figure does not cover all possible patterns.



(a) Sets of 8 (4 in 2D) fine voxels (blue) are recursively combined to form one coarse voxel (red).

(b) Addition voxels (yellow) are introduced in a non-aligned grid for non-manifold resolution on the coarse scale.

Figure 3.4.3.: Coarsening scheme applied in *Two-scale Dual Contouring*.

Here, we apply DC to both datasets and obtain two different reconstructed surfaces. The coarse scale surface is used as a patch distribution and from the fine scale we obtain our datapoints. The NURBS will be fitted to these datapoints. We call this approach *Two-scale Dual Contouring* (see Figure 3.4.4). Of course this simple approach comes with drawbacks:

- We cannot guarantee that the coarse and the fine scale have exactly the same topology. Topological details of the fine scale, which do not exist on the coarse scale, are lost.
- Both resolutions have to be chosen manually, since we do not have a criterion for evaluating the quality of the coarse scale surface reconstruction.

These drawbacks are especially bad for complex surfaces – like the output of topology optimization – where the topological details mentioned above are not an exception, but the default case. There are more elaborate surface extraction schemes that handle these problems. For the interested reader, alternative schemes such as *Dual Marching Cubes* [30] or *Manifold Dual Contouring* [17], are included in Appendix A.

Projection of Datapoints onto Quads

Now that we have constructed a NURBS-patch distribution, we can estimate the parametrization of the datapoints on the fine scale by projecting them onto the patches:

For this procedure we do the following steps:

1. Find out onto which of these patches a datapoint should be projected. This can be done by simply measuring the distance from the datapoint to the centroid of each patch and deciding to project onto the patch with the smallest distance.
2. Project the point onto the target patch. For this, we want the whole quad to be parametrized on $(u, v) \in [0, 1]^2$. This is done by first approximating the quad (which may not necessar-

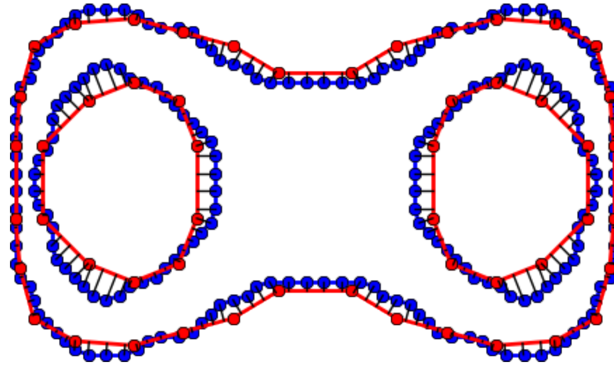


Figure 3.4.4.: Twoscale Dual Contouring with a coarse surface reconstruction (red) and a fine one (blue). The datapoints from the fine scale are projected onto the edges from the coarse scale (black lines).

ily be planar) with its least squares fit plane. Then a projection onto this plane is done by applying a simple basis transformation[2].

The projection might lead to parameters $(u, v) \notin [0, 1]^2$ for some of the datapoints. As we only produce surfaces for $(u, v) \in [0, 1]^2$, these are then not on the surface. If we still include them in the fitting, these will then influence parts of the surface that are not there, leading to unwanted fitting behaviour such as wiggles and turns.

We have tried several methods to solve this problem. For example, one might consider scaling the parameter space of each quad to $[0, 1]^2$, which however leads to inconsistencies in parameter assignemt between neighbouring quads with different scaling. Another solution, cropping the parameter domain by shifting all parameters outside $[0, 1]^2$ to the closest border (i.e. for u or $v < 0$ is assigned to 0, u or $v > 1$ is assigned to 1), also leads to problems, as points in different regions of space then can end up having the same parameter values. To simplify the treatment, we therefore currently only consider the points parametrized inside $[0, 1]^2$, and omit all points outside from this step onwards.

After completing all these steps we obtain the following data for the subsequent steps of our algorithm:

- a coarse surface delivering the topology for our NURBS-patches in Peters' Scheme
- a set of datapoints from the fine scale with coordinates $(x, y, z)^T \in \mathbb{R}^3$ and parameters $(u, v) \in [0, 1]^2$, where each datapoint is associated with a NURBS-patch.

For a sample output of our algorithm for a 2D case, see Figure 3.4.4.

[2]This basis transformation is computed in a very efficient way by computing the QR-decomposition for the basis of each patch only once and applying it to each datapoint projected onto this patch.

3.5. From Parametrized Surface Points to NURBS Representation

As of yet, there is no open-source software which provides the conversion from a *mesh-based* geometry to NURBS representation. Hence, one of the main challenges of both the algorithmic and implementation part of this project has been to develop one from scratch. Due to a variety of possible approaches to tackle this problem (e.g. [13, 23]), we conducted extensive prototyping work in MATLAB [6] to avoid cumbersome and time-consuming implementation overheads during the prototyping phase. Once the algorithms were finalized, the prototypes were implemented in a non-proprietary language, Python.

In the following section, we will thus present the final algorithm, based on the theory in section 2.4 and section 2.5[3]. For a short overview, we settled on using Peters' scheme (subsection 2.4.2) to fit NURBS smoothly to datapoints (subsection 2.5.3) with an included fairness term (subsection 2.4.3 and subsection 2.5.2). A similar approach was described by Eck and Hoppe in [13], for unstructured point clouds.

Algorithm

For the input:

- a quadrilateral mesh of vertices ($\mathbf{m} \in M$) and how they are connected into quads ($V_{\hat{f}}, \forall \hat{f} \in F$) as an ordered list of four vertex indices,
- a set of a datapoints ($\mathbf{d}_k \in \mathbb{R}^d$) and how they are parameterized (parameters u_k, v_k on quad $\hat{f} \in F$),

the algorithm proceeds as follows (see Figure 3.5.1 for a graphical overview):

1. For every quad in the mesh, a 4×4 grid of points (V_x) is created. In addition, several lists are created for labelling these points. These labels define the position of the points with respect to each other and to corners on the quads. They are used later in the algorithm, where information about neighbouring points is needed[4].
2. The matrix $C(\mathbf{u}, \mathbf{v})C^{PS}$ containing coefficients between the datapoints $\mathbf{d}_k$ and the fitted points in V_x is created.
   (a) For every datapoint $\mathbf{d}_k$ in the input, its parameters are first scaled from $[0, 1]$ on the quad to $[0, 1]$ on a local tensor product Bézier surface patch, related to the closest point in V_x. Then, the coefficients on the Bézier control points of this patch (corresponding to the datapoint's row in $C(\mathbf{u}, \mathbf{v})$) are calculated using the Bernstein polynomials, as described in section 2.4.
   (b) The corresponding row in $C(\mathbf{u}, \mathbf{v})C^{PS}$ is calculated, using a matrix-free formulation of C^{PS}. This matrix-free formulation uses a combination of a precalculated table of coefficients for the points in Peter's scheme (calculated according to [13]), together with the lookup of neighbouring points in V_x saved in the beginning of the algorithm.
   (c) The row of $C(\mathbf{u}, \mathbf{v})C^{PS}$ is saved in a sparse matrix[5].

[3] For reference, the nomenclature used in these sections is also used here.
[4] These labels are correlating with points A, B and C in [1] and [13], for reference.
[5] The sparse matrices for the surface reconstructed part are implemented using SciPy [31].

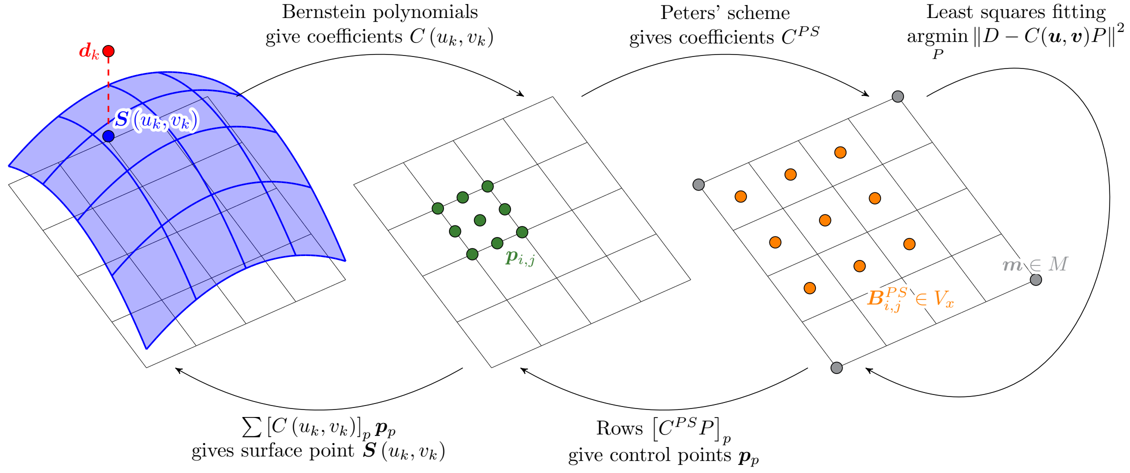


Figure 3.5.1.: Visualisation of the Bézier surface fitting algorithm, shown for one datapoint $\mathbf{d}_k$. After the algorithm, the sum of squared distances $(\mathbf{d}_k - \mathbf{S}(u_k, v_k))^2$ is minimized. The algorithm takes the parameterized input points $\mathbf{d}_k \in \mathbb{R}^3$, and produces a network of Bézier patches with control points $\mathbf{p}_p$, that can be used to create a surface.

3. For every point in V_x (that is, for every Bézier patch), the precalculated coefficients of the fairness functional (calculation described in subsection 2.5.2) are applied to the Bézier control points of the patch to create one row of C^{fair}. Using the same matrix-free formulation as before, the corresponding row of $C^{fair}C^{PS}$ is calculated, and saved in another sparse matrix.

4. The fitting to datapoints and fairness functional parts are combined by concatenating the datapoint matrix D with a 3-dimensional 0–vector, to form $\tilde{D}$, and by concatenating $C(\mathbf{u}, \mathbf{v})C^{PS}$ and $C^{fair}C^{PS}$ vertically to form $\tilde{C}(\mathbf{u}, \mathbf{v})C^{PS}$.

5. The actual locations of the points in the mesh V_x is calculated using sparse linear least squares[6].

6. The Bézier points of the patches are reconstructed from the points in V_x, quad-wise, using a matrix-free formulation of the second part of Peters' scheme, again together with the lookup of neighbouring points saved in the beginning of the algorithm.

At this point, we already have the components necessary for a reconstructed surface, now formulated as a network of Bézier patches[7]. A sample result at this stage, showing the fitting of a surface to a toroidal shape defined implicitly for the DC algorithm, is shown in Figure 3.5.2.

[6] Also using SciPy.

[7] Or *tensor produduct Bézier surface patches*, of second and third order (quadratic and cubic). See section 2.4.1 for definition (Equation 2.4.2).

However, for conveniece and usability, we convert them into one cubic tensor product NURBS curve surface patch per quad. That means the 4×4 Bézier patches per quad are turned into one cubic NURBS patch.

7. The network of Bézier patches are combined to a network of NURBS patches.
   (a) First, for each quad, the quadratic Bézier patches on this quad are degree-raised[8] to cubic in both directions.
   (b) Then, we can combine the 4×4 Bézier patches of the quad into one cubic NURBS patch. This is done by inserting the Bézier control points in a 13×13 array, where the shared control points along the Bézier patch edges are only inserted in once (instead of once for each Bézier patch), and using the knot vector

$$(0, 0, 0, 0, 0.25, 0.25, 0.25, 0.5, 0.5, 0.5, 0.75, 0.75, 0.75, 1, 1, 1, 1),$$

   in both directions.[9]
8. The NURBS patches are returned as a list of control points and an ordered array of indices for what control points belong to which quad.

3.6. From NURBS to Standardized CAD File Format

In order to create standardized CAD files from previously computed B-Spline control points, the scripting functionality offered by FreeCAD is employed. Almost all functions of FreeCAD can be called using a python script, which allows utilization FreeCAD functions within our automated workflow. [25]

The implemented python script is structured as follows:

1. The installation path for FreeCAD is specified and its modules are imported separately
2. A new document is opened and B-Spline patches are created consecutively using the control points obtained from Peters' scheme
3. The object is reoriented to revert coordinate changes imposed by ToPy (see section 3.3.2) and exported as a basic output (without appplying geometry and domain constraints)
4. Geometry and domain constraints are enforced with boolean operations
5. The active object is exported as step file, which is the final output of the program

[8] See for example [32] for how to compute the control points of an equivalent Bézier curve with a higher degree.

[9] That the surfaces are the same can be verified using for example the formulae in pp. 115-116 of [33]. A conceptual explanation would be as follows: we formulate the Bézier surfaces as NURBS surfaces, and then connect them along the edges by changing the knot multiplicities. This can be done since the Bernstein polynomials can be recovered from the NURBS basis functions (by clamping at both ends, thus using knots 0 and 1 with multiplicities $N + 1$ for an order N curve having $N + 1$ control points). Then, just as two clamped NURBS curves sharing one end point are combined easily into one curve (by concatenating the lists of control points, but putting the shared control point in only once, and concatenating the knot vectors, but reducing the multiplicity of the knot at the intersection to the order of the curve), we combine the curves along the surface in both directions to have a combined surface.

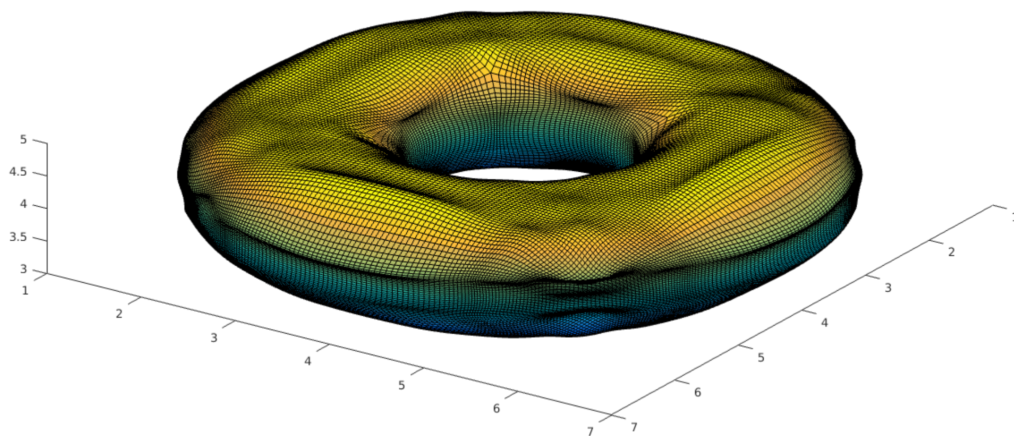


Figure 3.5.2.: A sample result from the Peters' scheme least-squares minimisation surface fitting, using the data provided by the DC algorithm for an implicit function describing a torus. The grid lines on the figure are following the constant lines of each parameter value. As they follow the patch edges, corners where other than 4 coarse quads are meeting can be recognized, as for example on the middle of the far side of the torus, on the side that's facing the viewer.

3.7. Graphical User Interface

In order to ensure convenient and intuitive user interaction with the program, a graphical user interface (GUI) was implemented using the Qt5.4 [34] framework. The GUI enables the user to enter all necessary files and parameters, eliminating any need for interaction through the command line. For supplying input files, the user needs to choose only the main *.step* file. Assuming all other files were named according to the naming convention (see section 3.2), activating the appropriate checkbox (for specifying fixtures or the optimization domain) should suffice. (see Figure 3.7.1).

All input parameters needed for topology optimization can be entered through text fields:

- Force Scaling - parameter for the scaling force magnitudes (see section 3.2).
- Resolution - parameter for calculating the voxel size for the voxelization. Voxel size is then equal to $2^{-(Resolution-1)}$. Hence, by increasing the resolution the user reduces the length of the edge of a voxel by half, thus improving the accuracy of the solution.
- Volume Fraction Limit - the fraction of the volume to be kept after the topology optimization process by ToPy (see sec. 3.3).

Similarly, all parameters necessary for surface fitting (see section 2.5) can be entered through text fields:

- Smoothing - parameter for the fairness functional (see section 2.4)
- Coarsening - the number of coarsening steps in Dual Contouring(see subsection 3.4.2).

After all parameters have been specified, the pipeline can be executed without any intervention by the user. Progress of the program can be monitored by the dials present at the bottom of the GUI window. After completion of the process, the GUI provides an option to launch FreeCAD directly from the GUI and analyze the results.

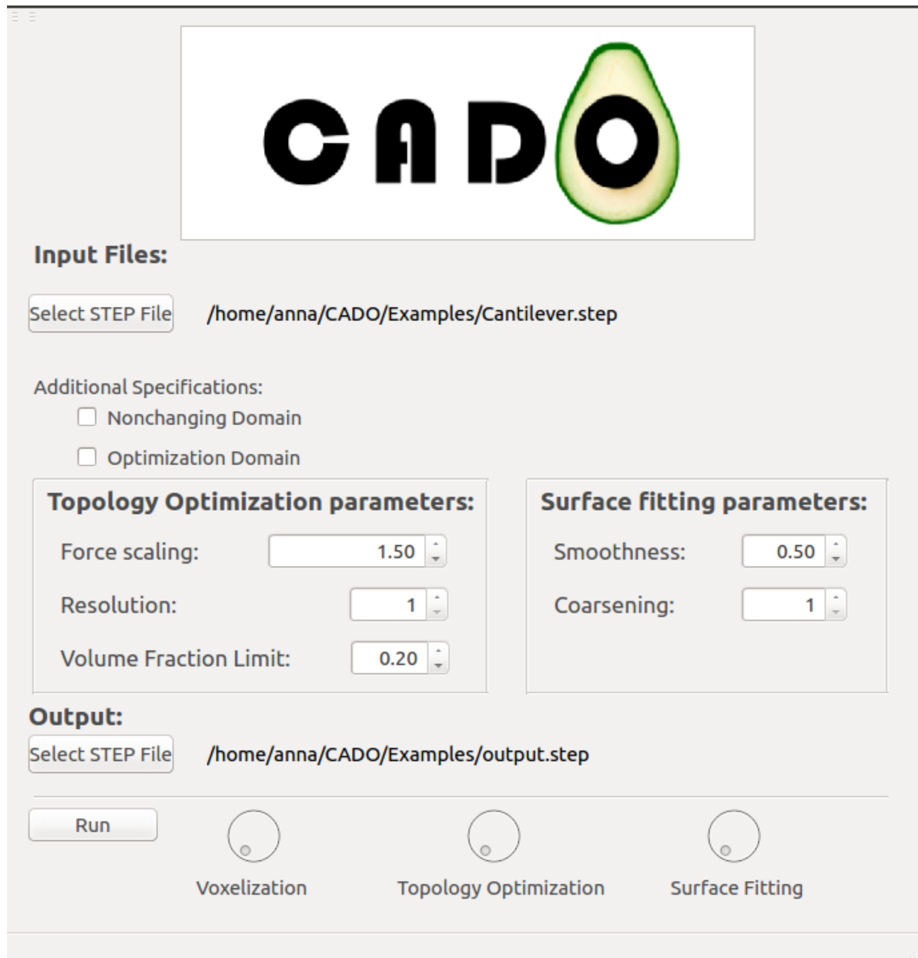


Figure 3.7.1.: Program GUI showing file input and parameters for topology optimization and surface fitting. During execution of the program, the dials at the bottom indicate progress of the operation.

4. Results

The six stages of the workflow (see chapter 3) are assembled into CADO, complete with GUI encapsulation. This software enables a less complicated design workflow for the user, which is described in section 4.1.

In order to assess CADO's performace, several test cases are carried out. These tests, with their initial and boundary conditions, as well as the resulting optimized structures are presented in section 4.2.

4.1. Product Overview

CADO, the computer-aided design optimization tool, provides users with a turnkey solution for their design problem. CADO takes as input the geometry and boundary conditions on one hand, and simulation and surface fitting parameters on the other. The former input is given through CAD files, and the latter as numeric entries in the GUI. Using these parameters, CADO computes the optimal material distribution and reconstructs a fully CAD-compliant geometry from it.

In terms of user experience, CADO offers simplicity and intuitiveness. It requires no knowledge of its algorithms and implementation from the user. Provision of all parameters and files necessary for the simulation is seamlessly integrated into the GUI (see Figure 3.7.1). Once under progress, the status of the simulation can be assessed throught the progress bars, corresponding to the three major steps:

- Voxelization
- Topology Optimimzation
- Surface Fitting

Further information and the detailed description of the user interface can be found in the user-guide (see Appendix C). CADO itself is licensed under the open source BSD license and can be found on Github [35].

4.2. Test Cases

The performance of CADO is evaluated through three tests. The first case is a standard Cantilever design problem in three dimensions. The second deals with optimization of a bridge design. Finally, the third case leans more towards real-world applications, and concerns a design challenge on a jet engine bracket. Here we describe the underlying boundary conditions and the outcome of the tests.

4.2.1. Cantilever

The first test case simulates a cantilever. A cuboidal block of material is provided as the initial condition. This block is fixed at one end to a wall. The load on the cantilever is modeled as a downward force on its other end. (see Figure 4.2.1).

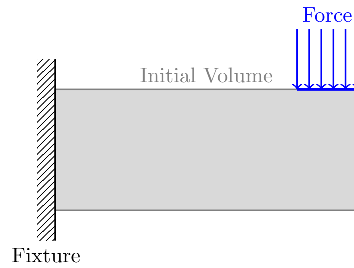
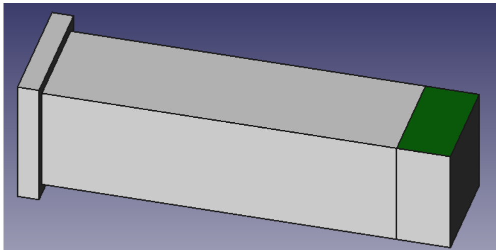
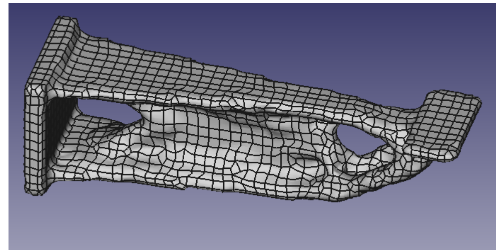


Figure 4.2.1.: Boundary conditions for the test case "Cantilever".

The cantilever is first modeled on a CAD designer (Figure 4.2.2a). With all necessary input files and parameters, CADO produces a topology-optimized structure (Figure 4.2.2b). There is a 78% reduction of volume as compared to the initial configuration. The NURBS surface enclosing this volume consists of 2202 single NURBS patches.



(a) CAD design



(b) Optimized structure

Figure 4.2.2.: Test case "Cantilever"

4.2.2. Bridge

The second test scenario simulates a bridge as a volume resting on two supports and a plane – the intended driving plane – in the middle of the volume. This plane is subjected to an area force that models the load of traffic as well as the bridge's self weight. The two fixtures are intentionally placed at non-symmetric positions ($a \neq b$, see Figure 4.2.3).

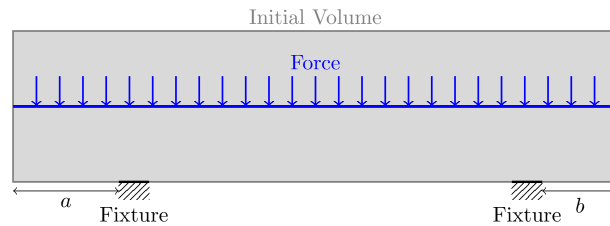
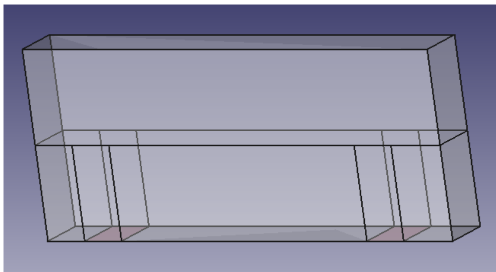
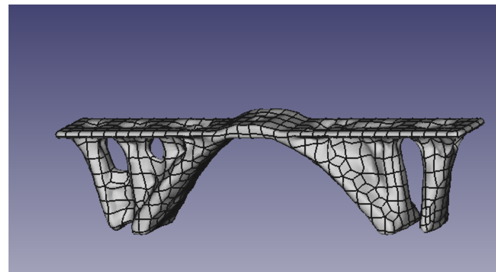


Figure 4.2.3.: Boundary conditions for the test case "Bridge".

The bridge modeled in CAD (see Figure 4.2.4a) and processed through CADO. The CAD output shows a wholesome balance between intuition and efficiency – it complies with the intuitive idea of a bridge, and also ensures minimal use of material. (see Figure 4.2.4b). The optimized structure is represented by 868 NURBS patches, with approximately 85% reduction of volume. We also observe "dents" on the surface of the driving plane.



(a) CAD design



(b) Optimized structure

Figure 4.2.4.: Test case "Bridge"

4.2.3. GE Jet Engine Bracket

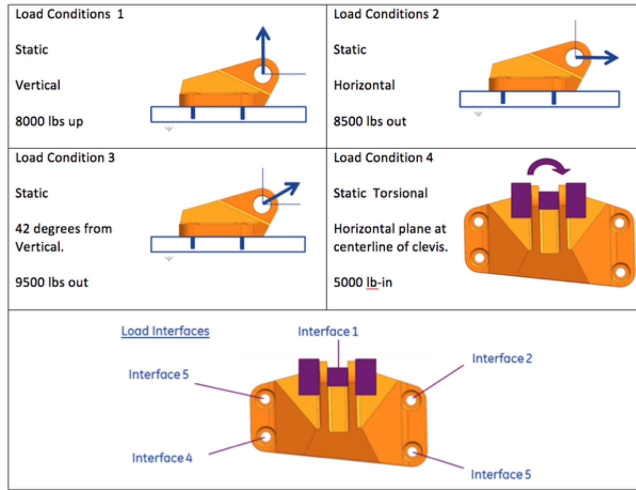
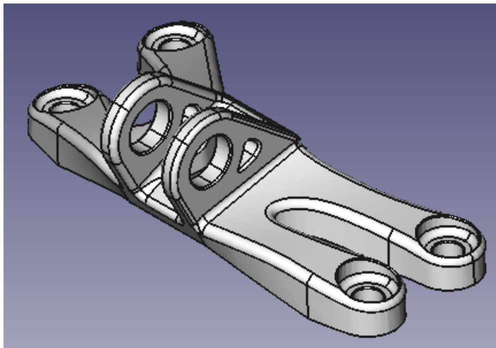


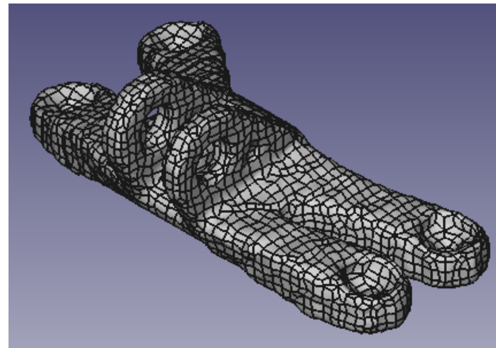
Figure 4.2.5.: Boundary conditions and different load cases for test case "GE Bracket". Figure from [36].

On the contrary to the previous two tests, the final test case resembles a real application scenario. The load conditions and the initial volume originate from a topology optimization challenge proposed by General Electric in 2013 [36]. The original goal of the challenge was the optimization of a jet engine bracket, that is subjected to four different load conditions and five different non-changing regions (see Figure 4.2.5).

In the following an already optimized design from [37] was chosen, since the scale of the scenario was too big to be handled by the topology optimizer ToPy, due to runtime and memory constraints. Therefore, the task for the last test case was only the reconstruction of the given geometry, in order to investigate the performance of the surface fitting algorithm for real-world applications. While the original design is described by 226 faces, the reconstructed version took 3202 NURBS patches. Additionally some topological features are lost in the region close to "Interface 1"(see Figure 4.2.5).



(a) CAD design. Initial geometry is already an optimized design from [37].



(b) Reconstructed structure

Figure 4.2.6.: Test case "GE Bracket"

5. Summary and Future Work

In the following section, we give a short summary of what was discussed in this report and what was accomplished in this project. We also provide a short discussion on potential improvements and extensions to CADO.

5.1. In a Nutshell: CAD-integrated Topology Optimization

To address a lack of software integrating Topology Optimization within a CAD framework, we have developed CADO, Computer Aided Design Optimizer. In summary, CADO works as a fully integrated tool-chain from the CAD input file to an optimized CAD file. It is fully Open Source, implemented using other freely available libraries and written in non-propietary languages. It is also developed in a modular fashion to allow further development and exchanging of parts of the software. An algorithm to fit a NURBS surface to a surface defined by voxel data was also developed, as a solution to this problem was not found in current software or literature.

The software pipeline executed in CADO from a specified input can be described as follows. Firstly, the input geometry undergoes voxelization using OpenCASCADE to ensure compatibility with the topology optimizer. Secondly, the topology is optimized by employing the open-source tool ToPy [26].

Next, we execute our algorithm to describe the optimized topology by NURBS surfaces. Firstly, a two-stage Dual Contouring surface reconstruction scheme is executed on the output of topology optimization. This gives a mesh of quadrilaterals, on which a set of datapoints describing the surface are parameterized.

This forms the inputs for a B-Spline surface fitting algorithm, which on each quadrilateral fits 4×4 Bézier patches, which are connected smoothly (with G^1 continuity) over the whole domain. This follows an similar approach to that by Eck and Hoppe in [13], by using least-squares fitting to an underlying set of points, from which the network of Bézier patches are built with guaranteed smoothness, as described by Peters in [1]. The sets of 4×4 Bézier patches per quad are then combined into single NURBS patches.

Lastly, a FreeCAD macro script performs boolean operations to enforce geometric constraints and exports the geometry to a standardized CAD file [25].

As already mentioned, the modular structure of the software allows for future replacement of parts with more appropriate or suitable solutions.

The functionality of the tool was tested through three test cases described in section 4.2. We made the following observations:

- The design problem could be easily formulated in the form of CAD files.
- The CAD files were parsed well to construct the topology optimization problem. Only small problems (Bridge, Cantilever) could be handled using ToPy. For bigger problems (GE Jet Engine Bracket) ToPy was not sufficient, due to runtime and memory constraints.

- Smooth NURBS surfaces of arbitrary topology were successfully reconstructed from the optimization solution, albeit with a high number of patches. Deformations of the resulting NURBS surfaces were observed ("dents" in the bridge). The reason for this is limited accuracy in the projection scheme described in section 3.4.2. Additionally, conservation of topological features was not guaranteed (GE Jet Engine Bracket).
- Finally, after post-processing for geometry constraints, the result was readily exported as a standard CAD *.step* file.

CADO is a strong proof of concept for CAD integrated topology optimization. It solved the proposed test scenarios to a qualitative level of satisfaction. Nevertheless, CADO's maturity to a full-fledged software package for engineering problems requires further improvements and additions.

5.2. Future Work

We consider three main areas of improvement in which a future project could evolve further. The first one deals with the *robustness* of the methods. In order to get better results, there are specific ways in which the algorithms we used, could provide more reliable results. Adaptivity is one of the best approaches to tackle robustness, therefore we propose to:

- Implement an adaptive DC algorithm
- Find a general approach to avoid special manifold treatment
- Implement an adaptive surface reconstruction. This could work by first fitting a surface on very coarse quads, relying on using an error measure (see below) to detect if the error could be improved. Afterwards subdividing these quads and finally rerun the fitting algorithm either locally or globally. For further ideas on surface reconstruction, refer to Appendix A.

The second area of improvement relies on *correctness*. Until now the output results have not been checked for accuracy. In order to find the optimal approach with the best parameters we would need a concrete measure of the error. Therefore we would suggest to:

- Find an error measure to analyze the deviations in the DC algorithm
- Implement an error measure for the surface reconstruction, for example the (squared) difference between the datapoint locations and the respective parameterized surface point
- Implement an improved parameter estimation

Finally, the toolkit could profit from using *alternative approaches* in the various subsystems. Replacing one section of the workflow, for example the surface extractor, with a faster and more robust method would provide a more rigorous mechanism . Some ideas for the interested reader would be to:

- Explore the advantages of an Isogeometric analysis
- Apply a faster Topology Optimization, which could deal with big size voxelized data
- Carry out a shape optimization as a postprocessing step

A. Surface Reconstruction

Our implementation of the surface reconstruction scheme (see subsection 3.4.1) and the two-scale approach (see subsection 3.4.2) so far only allows simple structures like spheres or tori if one is restricted to few NURBS patches. For more complicated structures – for example the output of a topology optimization tool – we currently have to use a very high resolution, which results in very costly calculations. An alternative approach is the application of a more elaborate surface extraction scheme. A short summary of different schemes is given in the following.

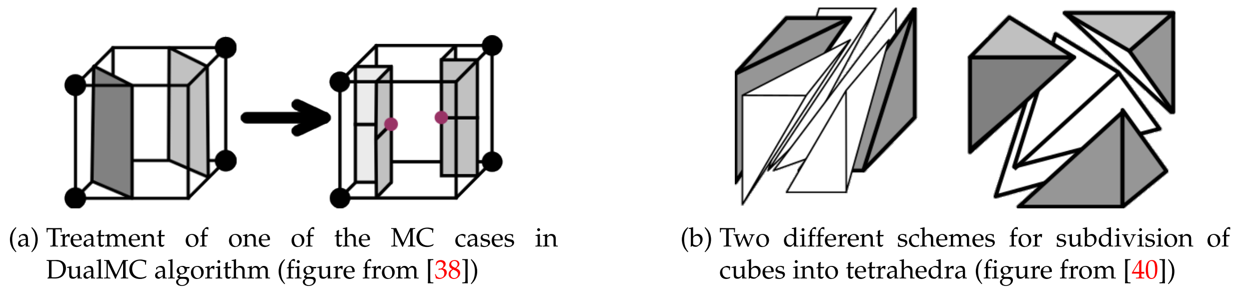
A.1. Manifold Dual Contouring

Our implementation of DC is a very basic one. This causes problems when it comes to the coarse resolution of our two scale approach: We need as coarse a mesh as possible, which has the same topology as our fine mesh. These goals cannot be reached with a basic approach, but only with an adaptive and topology safe DC algorithm like *Manifold Dual Contouring* [17].

A.2. Dual Marching Methods

The non-manifold edges generated by DC have been resolved by applying a remeshing scheme. But there are also hybrid methods of DC and MC, which have been developed to cope with the drawbacks of each. These methods use ideas from both the MC and the DC approaches. Using one of these methods would be a way of inherently avoiding non-manifold edges while keeping all the beneficial properties of DC. Here, we briefly introduce two of those hybrid methods:

- **Dual Marching Cubes:** The *Dual Marching Cubes* (DualMC) method is a hybrid of MC and DC: We traverse the cubes like in MC and insert vertices and connect them like in DC. The combination of the 256 different cases from the basic MC, the extension for creating non-ambiguous surfaces, and the framework of DC – these result in a very effective but also complex algorithm. A drawback of this method is that for certain configurations it ends up creating non-quad faces. We refer the interested reader to [38, 39, 30].
- **Dual Marching Tetrahedra:** While DualMC is a very complicated algorithm, *Dual Marching Tetrahedra* (DualMT) uses tetrahedra instead of cubes and therefore reduces the 256 different cases from MC to $2^4 = 16$ cases. Even though the method is working on tetrahedra, we can still apply it to a voxel dataset, by composing each cube out of 5 or 6 tetrahedra (Figure A.2.1b). Nevertheless, this high amount of simplification comes with a drawback: the treatment of ambiguous cases depends on the splitting scheme applied to a cube. Further details on this method can be found in [40].



(a) Treatment of one of the MC cases in DualMC algorithm (figure from [38])

(b) Two different schemes for subdivision of cubes into tetrahedra (figure from [40])

Figure A.2.1.: Illustrations of Dual Marching methods.

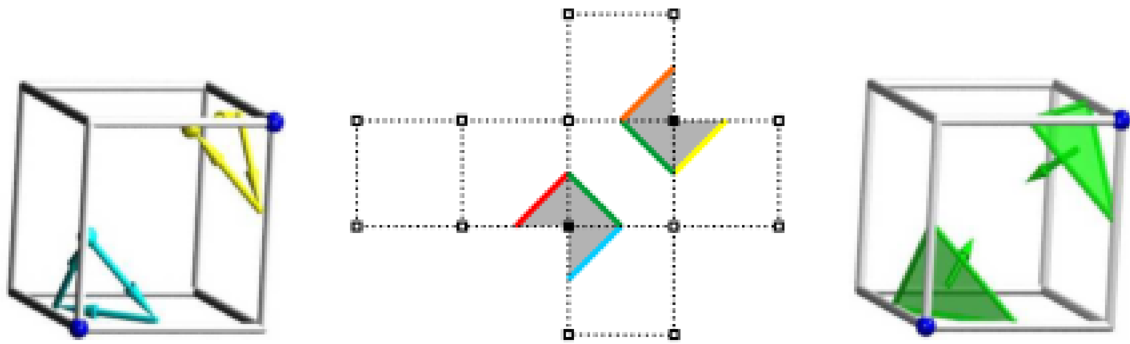


Figure A.3.1.: Illustration of the unfolding technique applied in *Cubical Marching Squares* from [41]

A.3. Cubical Marching Squares

Another promising, yet not perfectly fitting method is the *Cubical Marching Squares* method. Here one unfolds each cube and first investigates its six faces. On the faces one determines the necessary edges and finally the region inside the cube, that is defined by the edges, is polygonized (see Figure A.3.1). The main drawback of this method is that it only produces surfaces composed of triangular faces. Still it might be possible to modify this algorithm in a way that it outputs surfaces of quadrilateral faces. For detailed information we again recommend the original paper [41].

B. Installation Guide

In this appendix we provide a copy of the CADO Installation Guide for completeness of the report. This document is provided with the standard distribution of CADO.

INSTALLATION GUIDE

CONTENTS

1 ToPy 2
1.1 Prerequisites . . . 2
1.2 Install ToPy . . . 2
1.3 Test ToPy . . . 3
2 OpenCascade 4
2.1 Install OpenCascade . . . 4
2.2 Test OpenCascade . . . 5
3 Miscellaneous 6
3.1 Qt & QtCreator . . . 6
3.2 FreeCAD . . . 6
4 CADO 6
4.1 Prerequisites . . . 6

ABOUT

This document provides general information about using the CADO software. CADO is a fully CAD-integrated topology optimization tool under the open source BSD license. This project is a part of the Bavarian Graduate School of Engineering at TU München and was developed by Saumitra Joshi, Juan Carlos Medina, Friedrich Menhorn, Severin Reiz, Benjamin Rüth, Erik Wannerberg and Anna Yurova in 2015-2016. The figures in this document are provided as reference output for the user.

1 TOPY

In our tool we use ToPy (https://github.com/williamhunter/topy) for topology optimization.

1.1 Prerequisites

In order to install ToPy, make sure that the following software is installed on your computer:

- Python (version 2.7)
- NumPy (Usually provided by Python distribution)
- PyVTK tool (https://pypi.python.org/pypi/PyVTK)
- Pysparse library (http://pysparse.sourceforge.net/)

Here are some recommendations for the installation of the tools/libraries mentioned above.

To install PyVTK tool, please run the following commands in your terminal:

```
> sudo apt-get install python-pip
```

```
> pip install pyvtk
```

The installation of the Pysparse library is a bit more cumbersome, since the pip-installation (like in the previous case) fails most of the times. So, here we provide an alternative way of installing Pysparse from the *.git* repository.

To install Pysparse (assuming the pip installation fails), make sure that *git* (https://git-scm.com/) is installed on your computer and then run the following commands in your terminal:

```
> git clone git://pysparse.git.sourceforge.net/gitroot/
pysparse/pysparse/
```

```
> cd pysparse
```

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

1.2 Install ToPy

If all the tools specified in the section 1.1 are installed, we can now proceed to the installation of ToPy itself. For that download ToPy from https://github.com/williamhunter/topy.

```
159
160 def _write_legacy_vtu(x, fname):
161     """
162     Write a legacy VTK unstructured grid file.
163
164     """
165     # Voxel local points relative to its centre of geometry:
166     voxel_local_points = asarray([[-1,-1,-1],[ 1,-1,-1],[-1, 1,-1],[ 1, 1,-1],
167                                   [-1,-1, 1],[ 1,-1, 1],[-1, 1, 1],[ 1, 1, 1]])\
168                                     * 0.5 # scaling
169     # Voxel world points:
170     points = []
171     # Culled input array -- as list:
172     xculled = []
173
174     try:
175         depth, rows, columns = x.shape
176     except ValueError:
177         sys.exit('Array dimensions not equal to 3, possibly 2-dimensional.\n')
178
179     for i in xrange(depth):
180         for j in xrange(rows):
181             for k in xrange(columns):
182                 if x[i,j,k] > THRESHOLD:
183                     xculled.append(x[i,j,k])
184                     points += (voxel_local_points + [i,j,k]).tolist()
185
186     voxels = arange(len(points)).reshape(len(xculled), 8).tolist()
187     topology = UnstructuredGrid(points, voxel = voxels)
188     file_header = \
189     'ToPy data, created '\
190     + str(datetime.now()).rsplit('.')[0]
191     scalars = CellData(Scalars(xculled, name='Densities', lookup_table =\
192     'default'))
193     vtk = VtkData(topology, file_header, scalars)
194     vtk.tofile(fname, 'ascii')
195
196 def  timestamp():
```

Figure 1: Changing of the output type of ToPy to `'ascii'`

For CADO it is necessary to have an output in the *ascii* format. By default the output *.vtk* files from ToPy are binary, so we need to change them to *ascii*. In order to do that, please perform the following actions:

- Open the ToPy source file *core/visualization.py*
- Go to the method `_write_legacy_vtu(x, fname)` in line 160
- Change `'binary'` to `'ascii'` in line 194 (see fig. 1)

After making the following edit, run the following command from the root directory of ToPy:

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

1.3 Test ToPy

In order to test whether the installation of ToPy was completed successfully it is possible to run some test cases provided in *examples* folder. For that, do the following:

- Enter one of the folders in examples
  (e.g. examples/cantilever)

```
                    python optimise.py cantilvr_3d_etaopt_gsf.tpd

===============================================================================
ToPy problem definition (TPD) file successfully parsed.
TPD file name: cantilvr_3d_etaopt_gsf.tpd (v2007)
-------------------------------------------------------------------------------
Domain discretisation (NUM_ELEM_X x NUM_ELEM_Y x NUM_ELEM_Z) = 28 x 37 x 111
Element type (ELEM_K) = H8
Filter radius (FILT_RAD) = 1.5
Number of iterations (NUM_ITER) = 50
Problem type (PROB_TYPE) = comp
Problem name (PROB_NAME) = cantilvr_3d_etaopt_gsf
GSF active
Damping factor (ETA) = 0.40
No passive elements (PASV_ELEM) specified
Active elements (ACTV_ELEM) specified
===============================================================================
 Iter | Obj. func.   | Vol.  | Change     | P_FAC | Q_FAC | Ave ETA | S-V fra
-------------------------------------------------------------------------------
ToPy: Solution for FEA converged after 451 iterations.
   1  | 5.231335e+01 | 0.150 | 8.5000e-01 | 1.000 | 1.000 |  0.400  |  0.010
ToPy: Solution for FEA converged after 452 iterations.
   2  | 3.023124e+01 | 0.150 | 2.0000e-01 | 1.000 | 1.000 |  0.400  |  0.010
```

Figure 2: Output of the ToPy test

```
-- Processing ToolKit: TKVRML (VrmlConverter;VrmlAPI;Vrml;VrmlData)
-- Processing ToolKit: TKXCAF (XCAFApp;XCAFDoc;XCAFPrs)
-- Processing ToolKit: TKXCAFSchema (MXCAFDoc;PXCAFDoc;XCAFDrivers;XCAFSchema)
-- Processing ToolKit: TKXmlXCAF (XmlXCAFDrivers;XmlMXCAFDoc)
-- Processing ToolKit: TKBinXCAF (BinXCAFDrivers;BinMXCAFDoc)
-- Processing ToolKit: TKXDEIGES (IGESCAFControl)
-- Processing ToolKit: TKXDESTEP (STEPCAFControl)
-- Processing ToolKit: TKDraw (Draw;DBRep;DrawTrSurf)
-- Processing ToolKit: TKTopTest (TestTopOpeDraw;TestTopOpeTools;TestTopOpe;BRepTest;GeometryTest;HLRTest;MeshTest;GeomliteTest;DrawFairCu
rve;BOPTest;SWDRAW)
-- Processing ToolKit: TKViewerTest (ViewerTest)
-- Processing ToolKit: TKXSDRAW (SWDRAW;XSDRAW;XSDRAWIGES;XSDRAWSTEP;XSDRAWSTLVRML)
-- Processing ToolKit: TKDCAF (DDF;DDocStd;DNaming;DDataStd;DPrsStd;DrawDim)
-- Processing ToolKit: TKXDEDRAW (XDEDRAW)
-- Processing ToolKit: TKTObjDRAW (TObjDRAW)
-- Processing application: DRAWEXE (DRAWEXE)
-- Configuring done
-- Generating done
-- Build files have been written to:                                git/oce-master/build
```

Figure 3: OpenCascade installation: cmake

- Execute a ToPy test run by running the following command in your terminal:

```
> python optimize.py <example.tpd-file>
```

The output should look as showed in picture 2.

2 OPENCASCADE

OpenCascade (http://www.opencascade.com/) is an open-source CAD kernel. It is widely used in engineering and design for geometry construction and editing.

2.1 Install OpenCascade

For technical reasons, we do not use OpenCascade from the official webpage, but from the *.git* repository.

To install OpenCascade this way, make sure that git (https://git-scm.com/) is installed on your computer and then run the following commands in your terminal:

- Clone the repository:

```
[100%] Building CXX object adm/cmake/TKXDESTEP/CMakeFiles/TKXDESTEP.dir/__/__/__/drv/STEPCAFControl/STEPCAFControl_DataMapOfShapeSDR_0.cxx
.o
[100%] Building CXX object adm/cmake/TKXSDRAW/CMakeFiles/TKXSDRAW.dir/__/__/__/src/SWDRAW/SWDRAW_ShapeFix.cxx.o
[100%] Building CXX object adm/cmake/TKXDESTEP/CMakeFiles/TKXDESTEP.dir/__/__/__/drv/STEPCAFControl/STEPCAFControl_DataMapNodeOfDataMapOfL
abelShape_0.cxx.o
[100%] Building CXX object adm/cmake/TKXSDRAW/CMakeFiles/TKXSDRAW.dir/__/__/__/src/SWDRAW/SWDRAW_ShapeExtend.cxx.o
[100%] Building CXX object adm/cmake/TKXSDRAW/CMakeFiles/TKXSDRAW.dir/__/__/__/src/XSDRAW/XSDRAW_Functions.cxx.o
Linking CXX shared library ../../../Unix/x86_64-Release-64/libTKXDESTEP.so
[100%] Building CXX object adm/cmake/TKXSDRAW/CMakeFiles/TKXSDRAW.dir/__/__/__/src/XSDRAW/XSDRAW_Vars.cxx.o
[100%] [100%] Building CXX object adm/cmake/TKXSDRAW/CMakeFiles/TKXSDRAW.dir/__/__/__/src/XSDRAW/XSDRAW.cxx.o
Built target TKXDESTEP
[100%] Building CXX object adm/cmake/TKXSDRAW/CMakeFiles/TKXSDRAW.dir/__/__/__/src/XSDRAWIGES/XSDRAWIGES.cxx.o
[100%] Building CXX object adm/cmake/TKXSDRAW/CMakeFiles/TKXSDRAW.dir/__/__/__/src/XSDRAWSTEP/XSDRAWSTEP.cxx.o
[100%] Building CXX object adm/cmake/TKXSDRAW/CMakeFiles/TKXSDRAW.dir/__/__/__/src/XSDRAWSTLVRML/XSDRAWSTLVRML_DataSource3D.cxx.o
[100%] Building CXX object adm/cmake/TKXSDRAW/CMakeFiles/TKXSDRAW.dir/__/__/__/src/XSDRAWSTLVRML/XSDRAWSTLVRML_ToVRML.cxx.o
[100%] Building CXX object adm/cmake/TKXSDRAW/CMakeFiles/TKXSDRAW.dir/__/__/__/src/XSDRAWSTLVRML/XSDRAWSTLVRML.cxx.o
[100%] Building CXX object adm/cmake/TKXSDRAW/CMakeFiles/TKXSDRAW.dir/__/__/__/src/XSDRAWSTLVRML/XSDRAWSTLVRML_DrawableMesh.cxx.o
[100%] Building CXX object adm/cmake/TKXSDRAW/CMakeFiles/TKXSDRAW.dir/__/__/__/src/XSDRAWSTLVRML/XSDRAWSTLVRML_DataSource.cxx.o
```

Figure 4: Sample output of the OpenCascade building process

```
> git clone git://github.com/tpaviot/oce.git
> cd oce
> mkdir build
> cd build
```

- Execute `cmake`:

  ```
  > cmake ..
  ```

  Sample output: see fig. 3

- Build OpenCascade:

  ```
  > make ..
  ```

  To speed up the process, build can be done in parrallel:

  ```
  > make -j<number_of_processors>
  ```

  Sample output: see fig. 4

- Install OpenCascade:

  ```
  > sudo make install ..
  ```

These steps are in accord with the installation guide on the Github page of OpenCascade itself. One can also use the CMake-GUI to change some of the build configuration if need be (e.g. include OpenMP support).

2.2 Test OpenCascade

In order to test whether the installation of OpenCascade was completed successfully it is possible to run a test provided by OpenCascade.

For that, run the following command from your terminal:

```
> make test
```

All performed tests should be successful (See fig. 5)

```
41/48 Test #41: STEPImportTestSuite.testImportAP214_1 ..........   Passed    0.15 sec
      Start 42: STEPImportTestSuite.testImportAP214_2
42/48 Test #42: STEPImportTestSuite.testImportAP214_2 ..........   Passed    0.07 sec
      Start 43: STEPImportTestSuite.testImportAP214_3
43/48 Test #43: STEPImportTestSuite.testImportAP214_3 ..........   Passed    0.05 sec
      Start 44: TestSuite.testNullPointer
44/48 Test #44: TestSuite.testNullPointer ......................   Passed    0.00 sec
      Start 45: TestSuite.testFloatEq
45/48 Test #45: TestSuite.testFloatEq ..........................   Passed    0.00 sec
      Start 46: TestSuite.testFloatNeq
46/48 Test #46: TestSuite.testFloatNeq .........................   Passed    0.00 sec
      Start 47: TestSuite.testBoolean
47/48 Test #47: TestSuite.testBoolean ..........................   Passed    0.00 sec
      Start 48: TestSuite.testIntegerLighter
48/48 Test #48: TestSuite.testIntegerLighter ...................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 48

Total Test time (real) =   3.98 sec
```

Figure 5: Sample output of the OpenCascade test

3 MISCELLANEOUS

3.1 Qt & QtCreator

To install the the newest version of **Qt**, visit the page http://ftp.fau.de/qtproject/archive/qt/5.4/5.4.2/ and download the *.run* file suitable for your computer. After that, change the rights for the installer file and install **Qt** by following instructions of the installation manager:

```
> chmod +x qt-opensource-linux-x64-5.5.0-2.run

> sudo ./qt-opensource-linux-x64-5.5.0-2.run
```

3.2 FreeCAD

Download and install FreeCAD following the instructions from the official FreeCAD webcite:
http://www.freecadweb.org/wiki/?title=Download.

It can also be installed directly from the command line as follows:

```
> sudo apt-get install freecad
```

4 CADO

4.1 Prerequisites

In order to install CADO the following tools should be installed on your computer:

- Topy (see Sec. 1)

- OpenCascade (see Sec. 2)
- QtCreator (see Sec. 3.1)
- FreeCAD

After having installed all the prerequisites, CADO is ready to install. To do that, perform the following command from the repository main folder:

```
> make
```

After the installation process has completed run the program from the command line:

```
> ./cado
```

C. User Guide

In this appendix we provide a copy of the CADO User Guide for completeness of the report. This document is provided with the standard distribution of CADO.

USER GUIDE

CONTENTS

1 Preparing your computer 2
1.1 Technical requirements & Installation 2
2 Structure 3
2.1 Input files 3
2.2 Topology Optimization Parameters 4
2.3 Surface Fitting Parameters 4
2.4 Output 5
3 Example 6

ABOUT

This document provides general information about using the CADO software. CADO is a fully CAD-integrated topology optimization tool under the open source BSD license. It resulted from a project as part of the Bavarian Graduate School of Engineering at TU München and was developed by Saumitra Joshi, Juan Carlos Medina, Friedrich Menhorn, Severin Reiz, Benjamin Rüth, Erik Wannerberg and Anna Yurova in 2015-2016.

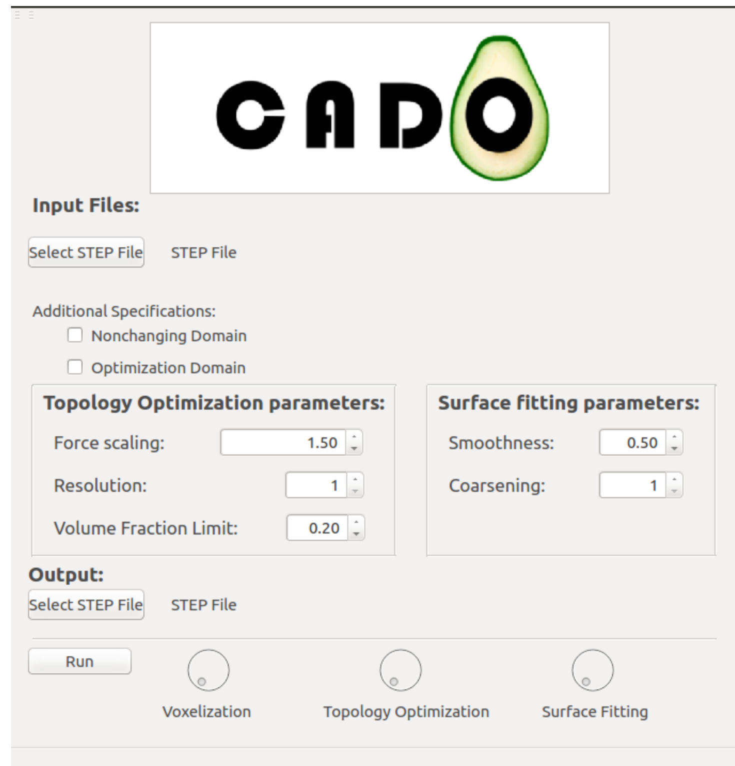


Figure 1: CADO main window

1 PREPARING YOUR COMPUTER

1.1 Technical requirements & Installation

Please make sure that the following software is installed on your computer in order to fulfil CADO's dependencies:

- ToPy
- OpenCascade
- FreeCAD
- Qt (version > 5.4.2)
- QtCreator (version > 3.4)

For the detailed installation instructions, please refer to:

CADO_InstallationGuide.pdf

Once all necessary software is installed CADO can be run from the terminal by issuing:

```
> ./cado
```

Once CADO is running, the main window appears on the screen (see fig. 1).

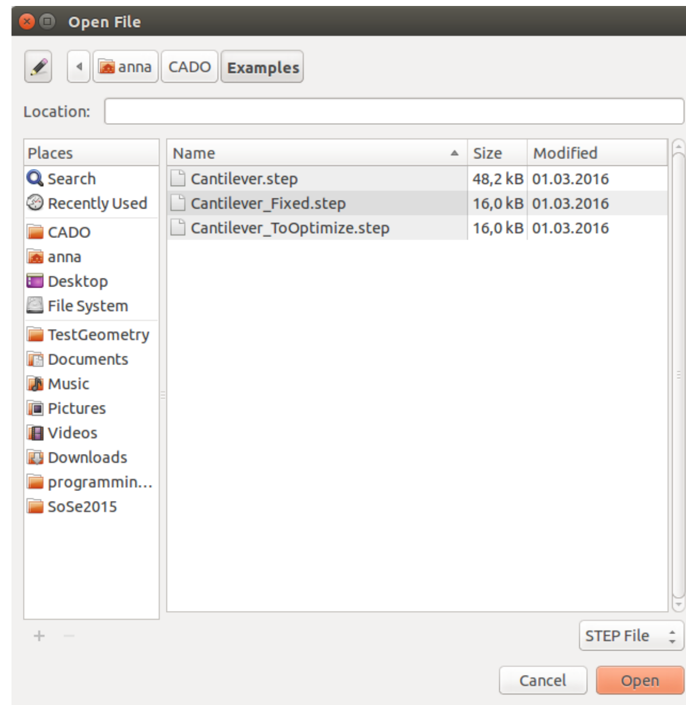


Figure 2: CADO input files. Please make sure that files follow the naming convention stated in the document.

2 STRUCTURE

Now let us take a closer look at the parameters required by CADO.

2.1 Input files

To select your input *.step* file:

- Press the **Select STEP File** button in the top left corner of CADO;
- Choose the appropriate file by navigating through the file system in the opened dialog (see fig. 2);
- Press the **Open** button.

After that, the path to the chosen file should appear near the **Select STEP file** button (see fig. 3).

Note that the same directory must also contain an *.iges* file with the information about colors (see fig. 2) with an identical name:

<StepFileName>.iges

2.1.1 *Nonchanging domain*

In case some domains need to be kept unchanged during the optimization (for example, screw threads), select the **Nonchanging domain** checkbox in the *Additional Specifications* section. Make sure that your *.step* file that specifies the nonchanging domains is located in the same directory as the original *.step* file (see fig. 2) and named as:

<StepFileName>_Fixed.step

2.1.2 *Optimization domain*

You can constrain the solution to be limited to a certain domain of optimization. In order to specify this, select the **Optimization Domain** checkbox in the *Additional Specifications* section. Make sure that your *.step* file with the optimization domain is located in the same directory as the original *.step* file (see fig. 2) and named as:

<StepFileName>_ToOptimize.step

2.2 Topology Optimization Parameters

In order to tune the topology optimization for your specific problem, please enter the following parameters:

- **Force Scaling** - scaling factor for the force from the input file. Scaling is performed from the range $-0.5 - 0.5$ (corresponding to the color in the original input file) to the desired one.
- **Resolution** - the resolution of the optimized geometry. Increase the number to increase the resolution and get a more accurate solution.
- **Volume Fraction Limit** - the fraction of the volume to be kept in the voxel after the topology optimization

To fine-tune the parameters, use the raise/lower arrows. You can also enter the numbers directly into the fields.

2.3 Surface Fitting Parameters

In order to specify the desired quality of the output surface, please enter the following parameters in the *Surface Fitting Parameters* section:

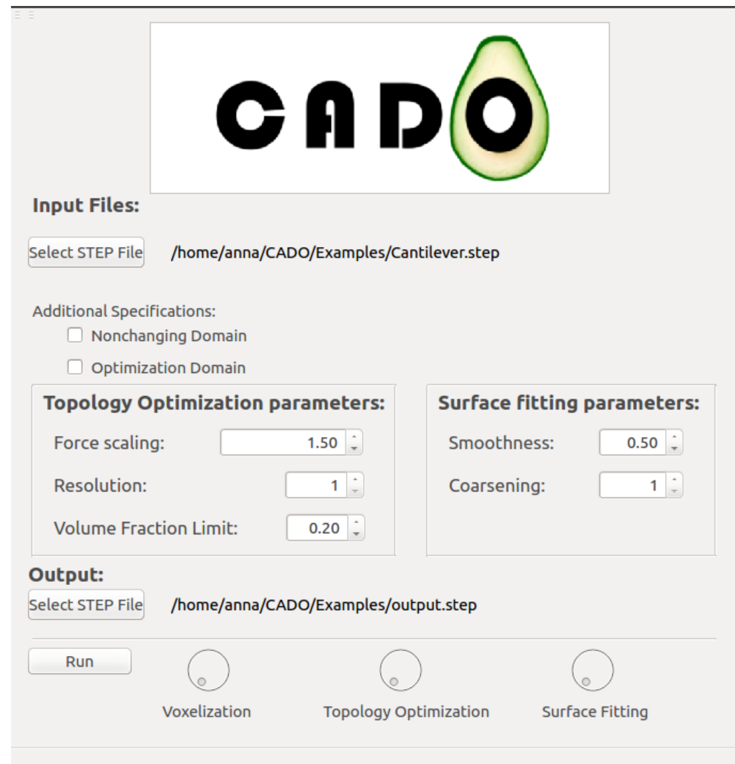


Figure 3: CADO main window with specified parameters

- **Smoothness** - increase the number to increase the smoothness. Note that some features might be lost if your desired smoothness varies from the physical result.
- **Coarsening** - the resolution of the output surface. Increase the coarsening to reduce the number of patches in the output surface. Note that increasing the coarsening factor can lead to loss of the features of the surface (such as holes, etc.).

2.4 Output

To specify the output file:

- Press the **Output** button;
- Choose the output destination by navigating through the appeared file dialog (see fig. 2);
- Enter the name of the output file;
- Press **Open**.

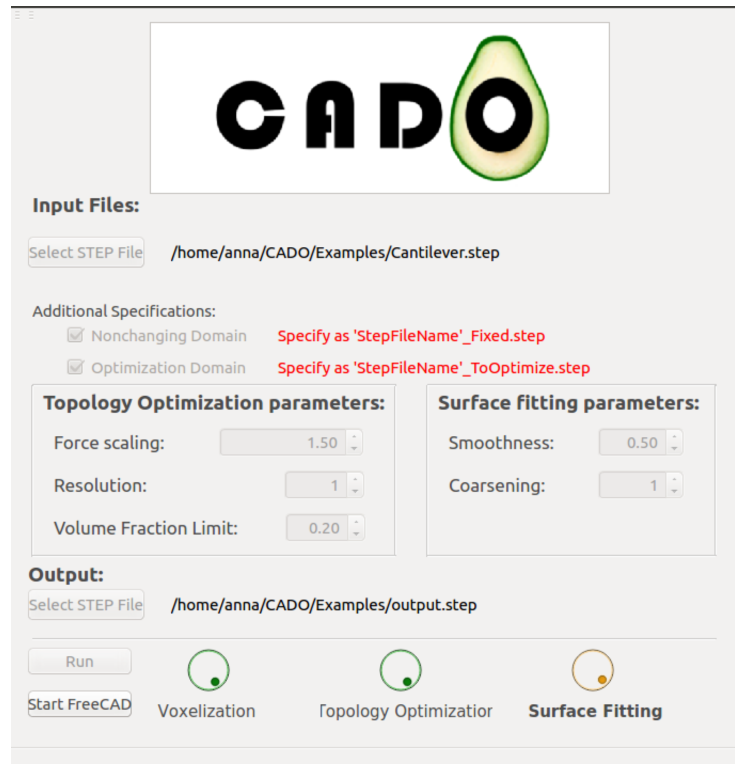


Figure 4: CADO in progress

Once the optimization process is completed (all progress bars are colored green), view your results in FreeCAD by clicking **Start FreeCAD** button.

3 EXAMPLE

To run a sample optimization, run CADO from the command line as described in 1. In the appeared window (see fig. 1) you can see the default parameters for the **Cantilever** test case, provided together with the source code distribution. To complete the necessary input for running the program, enter the following information:

- Choose the input file as described in sec. 2 from the following folder:

  Examples/Cantilever/

- Check the checkboxes **Nonchanging Domain** and **Optimization Domain**. The files with appropriate names are already placed in the folder.

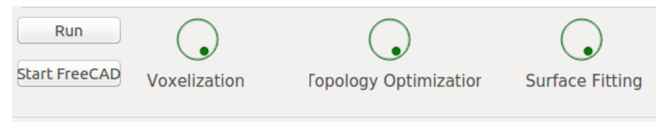


Figure 5: CADO. Optimization is finished

- Specify the output file following the instructions from the sec. 2.

To run the optimization press the **Run** button in the bottom of the window.

The progress of the Topology Optimization pipeline can be observed on the progress bars near the **Run** button (see fig. 4). Each round-shaped progress bar corresponds to one of the three main steps of the pipeline. While one of the steps is running, the corresponding progress bar is indicated yellow and moving. Once the step is finished, the progress bar is colored green (see fig. 5).

Once all progress bars are marked green the optimization process has finished. To see the optimized result in FreeCAD click on the **Start FreeCAD** button (see fig. 5).

Enjoy your topology optimized result in FreeCAD!

Bibliography

[1] Jörg Peters. Constructing C^1 Surfaces of Arbitrary Topology Using Biquadratic and Bicubic Splines. *Designing fair curves and surfaces*, pages 277–293, 1992.

[2] MMM SARCAR, K Mallikarjuna Rao, and K Lalit Narayan. *Computer Aided Design and Manufacturing*. PHI Learning Pvt. Ltd., 2008.

[3] Wikipedia. Constructive solid geometry — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Constructive_solid_geometry&oldid=674883510, 2015. Last accessed on 31/08/2015.

[4] D. Cohen-Or and S. Fleishman. An Incremental Alignment Algorithm for Parallel Volume Rendering. *Computer Graphics Forum*, 14(3):123–133, 1995.

[5] ChuaChee Kai, GanG.K. Jacob, and Tong Mei. Interface between CAD and Rapid Prototyping systems. Part 1: A study of existing interfaces. *The International Journal of Advanced Manufacturing Technology*, 13(8):566–570, 1997.

[6] MATLAB. *version 8.5.0 (R2015a)*. The MathWorks Inc., Natick, Massachusetts, 2015.

[7] William Hunter. Predominantly solid-void three-dimensional topology optimisation using open source software. Master's thesis, Stellenbosch University, 2009.

[8] RJ Yang and AI Chahande. Automotive Applications of Topology Optimization. *Structural optimization*, 9(3-4):245–249, 1995.

[9] Martin Philip Bendsøe and Ole Sigmund. *Topology Optimization: Theory, Methods and Applications*. Springer Science & Business Media, 2003.

[10] Ole Sigmund. A 99 line Topology Optimization Code written in Matlab. *Structural and Multidisciplinary Optimization*, 21(2):120–127, 2001.

[11] Timothy S Newman and Hong Yi. A survey of the marching cubes algorithm. *Computers & Graphics*, 30(5):854–879, 2006.

[12] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. Dual contouring of hermite data. *ACM Transactions on Graphics (TOG)*, 21(3):339–346, 2002.

[13] Matthias Eck and Hugues Hoppe. Automatic reconstruction of B-spline surfaces of arbitrary topological type. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 325–334. ACM, 1996.

[14] Gregory Nielson. Chord Length (Motivated) Parametrization of Marching Cubes IsoSurfaces. *Geometric Modeling and Processing*, 2004.

[15] Ulrich Schwanecke and Mario Botsch. Feature Sensitive Surface Extraction from Volume Data. *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, (121):57–66, 2001.

[16] Miguel Cepero. From voxels to polygons. http://procworld.blogspot.de/2010/11/from-voxels-to-polygons.html, 2010. Last accessed on 28/11/2015.

[17] Scott Schaefer, Tao Ju, and Joe Warren. Manifold Dual Contouring. *IEEE Transactions on Visualization and Computer Graphics*, 13(3):610–619, 2007.

[18] Marco Tarini, Nico Pietroni, Paolo Cignoni, Daniele Panozzo, and Enrico Puppo. Practical quad mesh simplification. In *Computer Graphics Forum*, volume 29, pages 407–418. Wiley Online Library, 2010.

[19] G.E. Farin, J. Hoschek, and M.S. Kim. *Handbook of Computer Aided Geometric Design*. Elsevier, 2002.

[20] G.G. Lorentz. *Bernstein Polynomials*. AMS Chelsea Publishing. American Mathematical Society, 2012.

[21] Gerald E Farin. *NURBS: from projective geometry to practical use*. AK Peters, Ltd., 1999.

[22] D. Doo and M. Sabin. Behaviour of recursive division surfaces near extraordinary points. *Computer-Aided Design*, 10(6):356–360, 1978.

[23] Gerrit Becker, Michael Schäfer, and Antony Jameson. *An advanced NURBS fitting procedure for post-processing of grid-based shape optimizations*. 2011.

[24] OPEN CASCADE S.A.S. OPEN CASCADE. http://www.opencascade.org/, 2015. Last accessed on 27/08/2015.

[25] FreeCAD. FreeCAD. http://www.freecadweb.org/, 2009. Last accessed on 27/08/2015.

[26] William Hunter. ToPy - 2D and 3D Topology Optimization using Python. https://github.com/williamhunter/topy, 2009. Last accessed on 21/08/2015.

[27] Pysparse. Pysparse. http://pysparse.sourceforge.net/, 2008-2011. Last accessed on 27/08/2015.

[28] Numpy.org. Numpy. http://www.numpy.org/. Last accessed on 27/08/2015.

[29] Python.org. Python. https://www.python.org/, 2015. Last accessed on 14/11/2015.

[30] Joe Warren Scott Schaefer. Dual marching cubes: Primal contouring of dual grids. In *Pacific Graphics*, pages 70–76, 2004.

[31] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python. http://www.scipy.org/, 2001 – ˙Last accessed on 30/03/2016.

[32] Degree Elevation of a Bézier Curve. http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/Bezier/bezier-elev.html. Last accessed on 30/03/2016.

[33] Eugene V Shikin and Alexander I Plis. *Handbook on Splines for the User*. CRC Press, 1995.

[34] The Qt Company. Qt5.4. http://www.qt.io/qt5-4/, 2016. Last accessed on 26/03/2016.

[35] BGCE@CSE 2015. Cado. https://github.com/BGCECSE2015/CADO, 2015-2016. Last accessed on 22/03/2016.

[36] General Electric. General Electric Jet Engine Bracket Challenge. https://grabcad.com/challenges/ge-jet-engine-bracket-challenge, 2013. Last accessed on 20/03/2016.

[37] Paul Tripon. Optimized design for GE jet bracket. https://grabcad.com/library/ge-jet-engine-bracket-53, 2013. Last accessed on 20/03/2016.

[38] G.M. Nielson. Dual marching cubes. *IEEE Visualization 2004*, pages 489–496, 2004.

[39] Yongjie Zhang and Jin Qian. Dual Contouring for domains with topology ambiguity. *Computer Methods in Applied Mechanics and Engineering*, 217-220:34–45, 2012.

[40] Gregory M Nielson. Dual marching tetrahedra: Contouring in the tetrahedronal environment. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5358 LNCS:183–194, 2008.

[41] Chien-chang Ho, Fu-che Wu, Bing-yu Chen, Yung-yu Chuang, and Ming Ouhyoung. Cubical Marching Squares: Adaptive Feature Preserving Surface Extraction from Volume Data. 24(3), 2005.