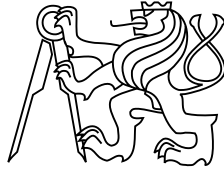


CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Civil Engineering

Department of Mechanics

Multimesh Methods for Data Visualization and Finite Element Analysis

DOCTORAL THESIS

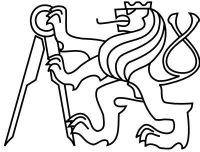
Ing. Štěpán Beneš

Doctoral study programme:
Civil Engineering

Branch of study:
Building and Structural Engineering

Supervisor:
prof. Ing. Jaroslav Kruis, Ph.D.

Prague, 2018



CZECH TECHNICAL UNIVERSITY IN PRAGUE
Faculty of Civil Engineering
Thákurova 7, 166 29 Praha 6

DECLARATION

Ph.D. student's name: Ing. Štěpán Beneš

Title of the doctoral thesis:
Multimesh Methods for Data Visualization and Finite Element Analysis

I hereby declare that this doctoral thesis is my own work and effort written under the guidance of the tutor prof. Ing. Jaroslav Kruis, Ph.D. All sources and other materials used have been quoted in the list of references.

In Prague on January 31, 2018 Signature:

Abstract

Multimesh Methods for Data Visualization and Finite Element Analysis

Štěpán Beneš

Finite element analysis is a process of modeling physical reality that consists of several phases – model generation, meshing, attribute assignment, solution, and post-processing. With the ongoing desire to solve more complex systems with better and better precision, an analysis has to process enormous amount of data in each of its phases. Traditional unstructured file-based representation of the mesh, input parameters, and the results from the solution is the bottleneck of the entire process. It lacks the scalability and complicates the development of the tools for engineers and researchers that are either preparing the input to FEM, or interpreting the output from FEM.

Limitations of the standard file-based approach are the motivation to re-think the entire process of data management in FEA. The focus of the thesis is mainly on the post-processing of the results and the way the results are stored, transfered, and visualized. However, the thesis describes the design and implementation of the complete FEA data management system that connects all the parts of the finite element analysis, providing query interface and remote access over the Internet. There is proposed the new storage format for representation of FEM results that provides the persistent representation of visual filters to simplify the implementation of a post-processor.

The main feature of the storage format is the support for compression of FEM results. The compression method based on singular value decomposition is proposed. The method is able to compress arbitrary results from FEM using low-rank approximation matrices. The compression ratio is at most 10% for all tested results. In many cases, the compression ratio is bellow 1% of the original size, while the relative approximation error is kept under 10^{-5}.

To demonstrate the proposed methods, the thesis describes the implementation of two post-processors. The desktop post-processor is a feature-rich visualization tool that allows to visualize the data in various formats including the new proposed storage format. It is able to create efficient surface representation of an arbitrary finite element mesh and it implements advanced techniques for manipulation with the mesh entities. The web-based post-processor is a simple cross-platform application that demonstrates the benefits of the proposed storage format. It is able to visualize the simulation results located in a remote storage. As the hard work connected with processing of the results is offloaded to a server, the web application is just a thin client that works even on devices with limited CPU and memory resources.

Keywords: Finite Element Method (FEM), Finite Element Analysis (FEA), Post-processing, Data visualization, Data compression, Data management, Singular Value Decomposition SVD)

Abstrakt

Vícesíťové metody pro vizualizaci dat a výpočty MKP

Štěpán Beneš

Konečně prvková analýza je proces sloužící k simulaci průběhů fyzikálních veličin, který sestává z několika fází – vytvoření geometrického modelu, generování sítě konečných prvků, přiřazení parametrů modelu, konečně prvkový výpočet a zpracování výsledků. S pokračující snahou o stále vyšší přesnost výpočtu, každá z fází analýzy musí zpracovávat obrovské množství dat. Tradiční reprezentace sítě, vstupních parametrů a výsledků založená na obyčejných nestrukturovaných souborech je úzkým hrdlem celého procesu. Tento fakt komplikuje vývoj nástrojů pro inženýry a vědce, kteří připravují vstupní data do MKP nebo interpretují výsledky z MKP.

Nevýhody a omezení tradičního přístupu založeného na souborech je motivací pro výrazné přepracování celého způsobu nakládání s daty v konečně prvkové analýze. Práce je zaměřena především na zpracování výsledků z MKP a na způsob jejich ukládání, přenos a zobrazování. Nicméně, dizertační práce popisuje také návrh a implementaci kompletního systému pro správu dat, který propojuje všechny části konečně prvkové analýzy a poskytuje rozhraní pro dotazování nad daty a vzdálený přístup přes Internet. Je zde rovněž představen nový formát pro reprezentaci výsledků z MKP, který mimo jiné podporuje uložení vizuálních filtrů aplikovaných na data, což usnadňuje implementaci post-procesoru.

Hlavní výhoda nového formátu je podpora pro kompresi dat. Kompresní metoda založená na singulárním rozkladu je představena a popsána. Metoda je schopna zkompresovat libovolnou sadu výsledků z MKP použitím aproximace maticí s nižší hodností. Kompresní poměr je nejvýše 10% pro všechny testované výsledky. V mnoha případech je kompresní poměr pod 1% původní velikosti, zatímco relativní chybu aproximace se podařilo udržet pod 10^{-5}.

Pro demostraci uvedených metod dizertační práce rovněž popisuje implementaci dvou post-procesorů. Desktopový post-procesor je vizualizační nástroj, který umožňuje zobrazovat data v různých formátech včetně nově navrženého formátu podporujícího kompresi výsledků z MKP. Post-procesor je schopen vytvořit efektivní reprezentaci konečně prvkové sítě a implementuje pokročilé techniky pro manipulaci s uzly, hranami a prvky sítě. Webový post-procesor je jednoduchá multi-platformní aplikace, která demonstruje výhody nového formátu. Umožňuje zobrazit výsledky z MKP umístěné ve vzdáleném úložišti. Díky tomu, že výpočetně náročné operace související se zpracováním výsledků jsou prováděny na vzdáleném serveru, webová aplikace je pouze tenký klient, který je schopen pracovat i na zařízeních s velmi omezeným výkonem a pamětí.

Klíčová slova: Metoda konečných prvků (MKP), Konečně prvková analýza, Vizualizace dat, Komprese dat, Správa dat, Singulární rozklad (SVD)

Acknowledgements

I would like to express my sincere gratitude to my supervisor prof. Ing. Jaroslav Kruis, Ph.D. for all his help and guidance throughout the work on this thesis, and also throughout my studies.

I also thank the teachers of my doctoral courses, especially prof. RNDr. Ivo Marek, DrSc., rest his soul, for his advice on the mathematical problems that arose during my research work. I am grateful for the discussions with prof. Dr. Ing. Bořek Patzák about the design of the database for the finite element model, which was an inspiration for a part of my research. My special thanks go to Ing. Martin Horák, Ph.D., who brought me to the idea to use singular value decomposition for the compression of FEM results. Many thanks also to the colleagues in the Department of Mechanics for a great time spent after the working hours.

I would also like to give my thanks to my parents for their patience and understanding. Last but not the least, I thank Hana for her endless support and encouragement that helped me to accomplish the goal.

This work was supported by the Czech Science Foundation under projects with numbers P105/12/G059 and 15-05935S. The financial support is gratefully acknowledged.

Contents

Declaration iii

Abstract v

Abstrakt vii

Acknowledgements ix

1 Introduction 1
- 1.1 Concepts . . . 2
- 1.2 Aims . . . 5
- 1.3 Challenges . . . 5

2 Related work 7
- 2.1 Data compression and visualization . . . 7
- 2.2 File formats . . . 8
- 2.3 Web-based data management . . . 8

3 FEA data management 11
- 3.1 System architecture . . . 11
- 3.2 Project-based data representation . . . 13
- 3.3 Storage format for results . . . 15
  - 3.3.1 Format specification . . . 17
  - 3.3.2 Compression . . . 21
  - 3.3.3 Encoding . . . 22
- 3.4 Post-processing . . . 23
- 3.5 Implementation details . . . 29
- 3.6 Results and evaluation . . . 33

4 Efficient methods to visualize finite element meshes 39
- 4.1 Theoretical background . . . 39
- 4.2 Implementation details . . . 41
  - 4.2.1 Data structures overview . . . 44
  - 4.2.2 Surface representation construction . . . 45
  - 4.2.3 Looking inside the mesh . . . 46
  - 4.2.4 Finding visible nodes . . . 47
  - 4.2.5 Selection of entities . . . 48
- 4.3 Results . . . 49

5 Approximation of FEA results by polynomial functions 53
- 5.1 Idea . . . 53
- 5.2 Implementation . . . 54
  - 5.2.1 Octree generation . . . 54
  - 5.2.2 Approximation in space . . . 56

Approximation functions 60
Results 62
5.2.3 Approximation in time 67
Difference between two functions 70
Results 71
5.3 Evaluation 71

6 SVD used for compression of FEA results 77
6.1 Mathematical background 77
6.1.1 SVD compression 77
6.1.2 Low-rank approximation matrix 78
6.1.3 Error estimation 79
6.1.4 Randomized SVD 80
6.2 Implementation 81
6.2.1 Algorithm description 82
6.2.2 Optimization 83
6.3 Results 84

7 Conclusions 93
7.1 Future work 94

A Data format for storage and transport of FEM results 95

Bibliography 99

List of Figures

1.1 Motivation example – reactor vessel model visualization. 2
1.2 Illustration of all FEA phases. 4

3.1 FEA system architecture. 12
3.2 FEA system workflow. 13
3.3 Database schema for FEA. 14
3.4 Database schema for FEA with results representation only. 15
3.5 Object representation of simulation results. 16
3.6 Element types supported in storage format. 19
3.7 Diagram of a layer tree. 23
3.8 Visualization of a master layer. 24
3.9 Visualization of a deformation layer. 24
3.10 Visualization of multiple slice layers. 25
3.11 Visualization of multiple iso-surface layers. 26
3.12 Visualization of an attribute selection layer. 26
3.13 Visualization of a surface layer. 27
3.14 Example of the vector field visualization. 28
3.15 Iso-areas visualization of the xx component of the stress tensor. 29
3.16 Desktop post-processor screenshot. List of remote solutions. 33
3.17 Desktop post-processor screenshot. Visualization of a master layer. . . 34
3.18 Desktop post-processor screenshot. Visualization of a slice layer. 35
3.19 Desktop post-processor screenshot. Visualization of a iso-surface layer. 35
3.20 Web post-processor screenshot. Visualization of a surface layer. 36
3.21 Web post-processor screenshot. Visualization of a slice layer. 36
3.22 Web post-processor screenshot taken on a mobile device. 37

4.1 Winged edge data structure. 41
4.2 Class diagram of element types. 42
4.3 Class diagram of surface representation. 43
4.4 Data structure overview. 44
4.5 Memory consumption of the surface representation. 45
4.6 Activity diagram of mesh construction. 46
4.7 Selection of element faces. 49
4.8 Meshes for benchmarks. 51
4.9 Visualization of significant edges. 51

5.1 Octree visualization. 55
5.2 Octree generation. 58
5.3 Diagram of octree generation classes. 59
5.4 Mean value approximation diagram. 61
5.5 Octree creation. 63
5.6 Nodal value computation. 63
5.7 Reactor vessel 2D. 64

5.8 Segment of reactor containment. 64
5.9 Geological layers simulation results. 65
5.10 Mean value approximation of geological layers simulation results. . . . 66
5.11 Trilinear approximation of geological layers simulation results. 67
5.12 Interpolation in time diagram. 68
5.13 Illustration of unifying time steps in octree nodes. 69
5.14 Heat transport analysis results (displacements). 72
5.15 Heat transport analysis results (displacements). 73
5.16 Exact data values of heat transport analysis results. 74
5.17 Approximation method's artifacts. 74
5.18 Glitches in approximation function. 75

6.1 Singular value decomposition illustration. 79
6.2 Dependency of SVD execution time on number of rows. 85
6.3 Dependency of SVD execution time on number of columns. 85
6.4 Results visualization: reactor containment 3D. 86
6.5 Dependence of NRMSD on compression ratio and rank (reactor containment 3D). 86
6.6 Results visualization: geological layers. 87
6.7 Dependence of NRMSD on compression ratio and rank (geological layers). 87
6.8 Dependence of NME on compression ratio and rank (geological layers). 88
6.9 Results visualization: reactor vessel 2D. 88
6.10 Dependence of NRMSD on compression ratio and rank (reactor vessel 2D). 89
6.11 Dependence of PSNR on compression ratio and rank. 89
6.12 Dependence of PSNR on compression ratio and rank (randomized SVD). 90
6.13 Execution time of standard and randomized SVD decompositions. . . 91

List of Tables

4.1 Initial loading time comparison. 50
4.2 Memory consumption comparison. 50

5.1 Approximated results of reactor vessel 2D simulation. 64
5.2 Approximated results of reactor containment simulation. 64
5.3 Approximated results of reactor vessel 2D simulation (space and time). 71
5.4 Benchmark results: heat transport analysis of Charles Bridge. 73

6.1 Memory consumption of compressed results. 3D reactor containment analysis. 91

Chapter 1

Introduction

The research work presented in this thesis has set ambitious goal to redesign the whole well established process of data management in finite element analysis software. It also proposes efficient methods to visualize finite element meshes and the results from complex finite element analyses.

Finite Element Analysis (FEA) is the term describing the entire process of modeling the physical system using the Finite Element Method (FEM). FEA consists of model generation, meshing, attribute assignment, solution, and post-processing. With the ongoing desire to solve more complex systems with better and better precision, an analysis has to process enormous amount of data in each of its phases. Traditional unstructured file-based representation of the mesh, input parameters, and the results from the solution is the bottleneck of the entire process. It lacks the scalability and complicates the development of the tools for engineers that are either preparing the input to FEM, or interpreting the output from FEM.

The solution of a large-scale finite element analysis itself can be parallelized and calculated in reasonable time on high-performance computing clusters. Nevertheless, in the end the results are transfered over a network and post-processed on an ordinary personal computer. Another concern is the colaboration and sharing of the model and the results between engineers and researchers. Limitations of the standard file-based approach lead to re-think the entire process of data management in FEA. The focus of the thesis is mainly on the post-processing of the results and the way the results are stored, transfered, and visualized. However, the storage format for the results introduced in this thesis was designed with the whole picture in mind and there is proposed a database-centric environment for complete FEA data management.

The analysis of reactor vessels in nuclear power plants can serve as an motivation example (see Figure 1.1). The analysis is used in the process of prolongation of the service life. The vessels are approximately 40 years old and detailed thermo-hydro-mechanical analysis has to be performed. Usually, two-dimensional axisymmetric or fully three-dimensional models are considered and it means hundreds of thousands degrees of freedom are used. The number of time steps is between 10,000 and 15,000. The output files contain displacements, strain and stress components, temperature, relative humidity (or moisture content) and several internal parameters (e.g., creep strains, damage parameter, etc.) in all time steps. The output files with size in the order of gigabytes are generated. More details can be found in [1, 2].

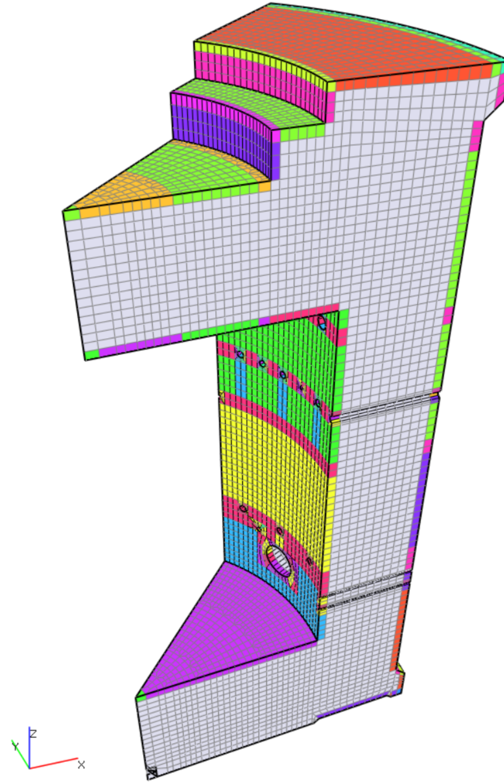


FIGURE 1.1: Motivation example. Visualization of a nuclear power plant reactor vessel model, which is used to perform complex thermo-hydro-mechanical analysis.

The thesis is structured in the following manner. Chapter 2 gives a brief revision of related work performed in FEA data management, file formats, data compression, and post-processing of the results from FEM. Chapter 3 describes an alternative to file-based data management, proposes the new storage format for FEM results, and presents tools that are based on this new storage format. Sections 3.1 and 3.2 contain a proposal of relational database model that connects all parts of the finite element analysis including geometry, model attributes, and simulation results, providing query interface and remote access over the Internet. Section 3.3 contains detailed specification of the new data format. Section 3.4 presents the design of the post-processor that is built on top of the data management system. Section 3.5 summarizes technical details related to the implementation of the data management system and the post-processor.

Chapter 4 describes the implementation of efficient methods to visualize finite element meshes. Chapter 5 presents a method for approximation of results from the finite element method using polynomial functions. Chapter 6 proposes different approach to compress FEM results that is based on singular value decomposition. Each individual chapter contains its own section that evaluates the results of the proposed methods therein. Chapter 7 concludes the thesis with final remarks, benefits and weaknesses of the proposed solutions, and possible future work.

1.1 Concepts

Here follows a summary of basic terms and concepts the thesis is based on.

Finite Element Method (FEM). FEM is a numerical method for solving problems of engineering and physics. Typical areas include structural analysis, heat transfer, fluid flow, and electromagnetics. The analytical solution of these problems generally require the solution to boundary value problems for partial differential equations. The finite element method formulation of the problem results in a system of algebraic equations. The method yields approximate values of the unknowns at discrete number of points over the domain [3]. To solve the problem, it subdivides a large domain into smaller, simpler parts that are called finite elements. This process is called the mesh generation [4, 5]. The simple equations that model these finite elements are then assembled into a larger system of equations that models the entire problem. FEM then uses variational methods from the calculus of variations to approximate a solution by minimizing an associated error function. The focus of this thesis is exclusively on the data management problems in the context of FEM-based simulations, not the simulations themselves.

Finite Element Analysis (FEA). Various data creation and modification tasks precede and follow the actual numerical solution of the boundary value problem using FEM. This whole process is called **Finite element analysis** and consists of several distinct phases. The basic phases of a FEA depicted in Figure 1.2 are[1]:

1. **Model creation** phase describes geometry of the domain, typically by defining boundaries of the domain using parametric surfaces like Bézier patches or Non-Uniform Rational Bézier-Splines (NURBS) [9] in a CAD (Computer Aided Design) tool.
2. **Attribute definition and assignment** specifies properties of the model, i.e., material properties of volumes, initial and boundary conditions for the solution.
3. **Mesh generation** decomposes the geometry of the model into simple shapes (triangles or quadrilaterals) or voxels like tetrahedra or bricks that fill the volume. This is often only an approximation of the orignal domain, because it is not possible for these simple shapes to fill the complex domain completely without gaps.
4. **FEM solution** uses the equations describing the problem, model discretization, and attributes to simulate the system's behavior. Often, the process is parametric either in geometry, or in assigned attributes. Simulation then produces multiple sets of output data, each for different configuration.
5. **Post-processing** of results is examination of the output by engineer or scientist, who is seeking the features and trends in data using the visualization tool.

[1]This is a quite simplified description of the FEA process as the results from one simulation are then sometimes used as an input for other simulations. Preliminary results of the simulation can also be post-processed continually during the calculation phase to monitor the convergence of the iterative methods. Another case represent the iso-parametric representation of finite elements [6], isogeometric finite elements [7], and NURBS-enhanced FEM [8]. These new approches are somewhat bridging the mesh generation phase as the curves describing the geometry are used also as the base functions of the finite elements.

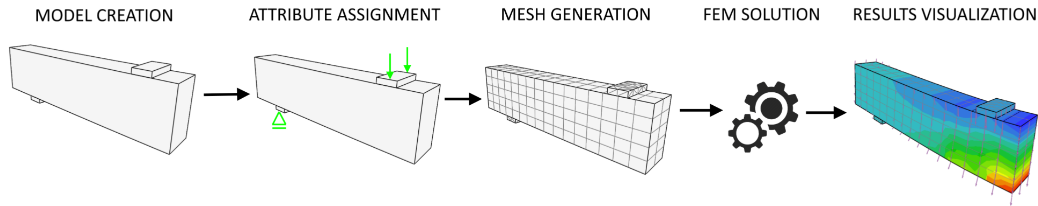


FIGURE 1.2: Illustration of pre-processing, discretization, solution, and post-processing phases of FEA.

Visualization tools are used to explore and analyse the data during all the phases. However, the vast majority of discussion about FEA focuses on the solution phase only. This makes sense as the solution phase consumes the largest portion of the computer time. But the solution phase itself consumes the insignificant amount of people-time. The majority of people's time is spent in pre-processing and post-processing of complex models. This fact seems to be overlooked and is one of the motivations for research work presented in this thesis.

FEM results. Results from FEM are scalar, vector, or tensor fields represented by discrete values. Some results are stored in nodes of the mesh, such as vectors of nodal displacements. Other results are stored typically in Gauss points (i.e., integration points) on finite elements. There are two similar sets of results. One is generated by a non-linear algorithms, where several incremental steps are stored and the other is generated by time integration, where results in particular time steps are stored. These results are represented as dense tabular values of basic types, usually double-precision floating-point numbers (8 bytes each). For example, a 3D material stress calculation of the domain discretized by tetrahedral elements with quadratic approximation yields 12 values for the stress and strain tensors in each Gauss point. There are 11 Gauss points in each element. If 100 time samples are taken, the size of the solution output is $100 \times 12 \times 11 \times 8 \approx 100$ kilobytes per single (!) element. The number of finite elements depends on (1) the resolution of the discretization, (2) the geometric complexity of the model, and (3) the desired accuracy of the output. In practice, fine discretizations of the problem domain contain millions of elements.

Current FEA software packages store the results either sequentially in formatted *ASCII or binary files* ordered by time steps, or use more sophisticated *database*[2] systems to preserve the links between the input model and the simulation results. Either way, the size of output data is very large for non-trivial analyses, which puts pressure on storage capacity, transfer times over the network, and memory consumption of post-processing tools.

Data compression. A compression method can be lossy or lossless. Lossless methods reduce information by identifying and eliminating statistical redundancy in data and are therefore able to fully reconstruct original data from its compressed form. Lossy methods, on the other hand, reduce the size by removing less important information in data and they are thus producing only an approximation of original data.

[2]The term database is used here to describe any structured storage beyond a simple file store.

Approximation error. Compression methods usually yield approximated data. In the following text, the term *approximation error* denotes an error resulted from compression, i.e., difference between original results of FEM analysis and their compressed form. It should not be confused with the error of the finite element method itself that yields approximate solution to mathematical problems used to model physical reality. To quantify the approximation error, several error metrics were investigated. In [10], there are some of them used in similar area of research. The ability to control the quality of compression was a key requirement for the implementation of the compression algorithm. There are defined several error metrics in the text (Section 6.1.3) that are used to measure the approximation error.

1.2 Aims

Here follows the list of aims that were set for the research work described in this thesis.

- The main goal of the research work is to design a new storage format, that will support compression of results, and outline the transition to a new post-processor that can read and visualize the compressed data in this new storage format. There is understandable resistance against invention of new data formats in the area of information technology. A new format leads to fragmentation of user base and compatibility issues. Conversion tools need to be created and maintained. There should be a strong motivation for introduction of a new format. However, there is no standard format for representation of results from FEM. Each software package uses proprietary format with syntax suitable for its internal implementation. There is also lack of support for compression methods that fit the character of FEM results. Standard file-based format does not allow querying of specific information without the need to parse through the complete set of results. Chapter 2 contains discussion about the existing formats in more detail and Chapter 3 describes the proposed format.
- In addition, a suitable compression method need to be developed. Singular Value Decomposition (SVD) (Chapter 6) is the most promising method used for compression of FEM results in this research. Other methods, that are investigated, include Wavelet transform [11] and approximation of discrete values by continuous polynomial functions (Chapter 5).
- Finally, the product of this research should be the implementation of two post-processors. The first is the standard desktop post-processor that will demonstrate the way of transition from the convential file-based formats to the proposed structured database format utilizing compression. The second post-processor is the web-based thin client intended to demonstrate the advantages of the proposed format when incorporated into a complex FEA running on a remote server.

1.3 Challenges

The main challenge is the design of universal format that can hold the results from any FEM analysis. Results are composed of scalars, vectors, or tensors. Each field has different number of components. The results can be located in nodes or integration points. There may be a requirement to extrapolate the results from integration

points to element nodes. There are various extrapolation strategies. The mesh can be different for each time step (e.g., in case of simulating the construction stages). The mesh can contain 1D, 2D, or 3D elements – each of different type and approximation. The results from 3D simulations can be visualized on the surface of the mesh, in form of cross-sections or iso-areas, or as a vector field. The storage format should support efficient representation of all these forms of results.

Finite element solution and post-processing of results can be sometimes done on different computers. Complex FEA solution phase runs on a supercomputer or a performant cluster of workstations, but the results are post-processed on a common personal computer that has significantly less memory available. Typical personal computer has 8 to 32 GB of RAM, while the size of results can be in order of tens to hunderds of gigabytes. Also, the data to post-process have to be first transfered over the corporate network or the Internet. These conditions indicate the need for partitioning of data into smaller chunks and/or compression of the data.

The goal of compression methods is the significant reduction in size while preserving the quality (keeping the approximation error low). Unlike with image compression methods, where the main aspect is the human perception of the reconstructed image, the compression of FEM results should be able to guarantee the matematical acuracy of the approximations and the user should be able to specify a desired value of the approximation error. Another concern is the computational complexity of the compression algorithm. The compression will be performed only once after the solution phase is complete. The computational time should be an order of magnitude shorter compared to the solution phase. Decompression (reconstruction of the original data) should be very fast as it is supposed to be performed every time the data are post-processed on the end device, which can be ordinary PC or even mobile device. The ability to create animations should also be taken into account.

Other kind of challenge is to provide the data management system that will connect all the FEA phases, i.e., to provide links between the geometric model, the mesh entities, and the output values. A FEA project typically encompasses multiple simulations, each with different input or solver parameters. Multiple users are usually involved in the project and the system should help them to cooperate during the preparation of the input and allow to share the output of the analysis. All these aspects influence the design of the data management system.

Chapter 2

Related work

This chapter gives a brief revision of related research work that deals with visualization of finite element meshes and results from FEM, file formats used for representation of FEM data, compression methods, and web-based FEA data management.

2.1 Data compression and visualization

The visualization of data produced by the finite element method is of two kinds. The discretization of the domain, called the finite element mesh, and the result fields that are mapped on the points inside the domain (either nodes or integration points). The general methods for visualization of 2D or 3D polygon (usually triangle) meshes have been researched in great depth. These data, when describing an object in great detail, can be relatively large in size if unoptimised. Therefore, many methods for reducing the size of data have been developed [12]. In [13], there is a wide survey of methods for data visualization and compression, especially with the focus on the web environment.

Progressive meshes represent one category of methods aiming for size reduction. They allow continuous, progressive coarsening or refinement of a polygonal mesh using a sequence of vertex-split operations. Progressive meshes were introduced originally in 1996 by Hoppe [14] and their efficient implementation was presented in 1998 [15]. Since then, there were various attempts to use Progressive meshes for compression and decompression of 3D meshes [16, 17, 18, 19] and also for streaming of geometrical information from web server to client [20, 21].

However, basic implementations of Progressive meshes and similar methods do not take into account vertex properties other than position. These methods are not designed to satisfactorily handle attributes assigned to mesh entities and their reconstruction. This is a major obstacle when applying the method to the data produced by FEM. Also, despite a considerable progress in the area of performance of Progressive meshes, decompression time is still a significant issue [22]. In order to keep compression and decompression time within reasonable limits, an untolerable compromise must be made in compactness of the compressed representation.

To avoid the problems with mapping FEM results on a coarsened mesh, the volumetric visualization can be used as shown in [23]. Volumetric modeling and rendering approaches are a common choice for visualization of large data from numerical simulations. It is a useful technique for visualization of FEA data either by generating semi-transparent 3D cloud images, or by extracting iso-surfaces. However, volumetric rendering can lead to the loss of details in regions where singularities or discontinuities in the data occur. Although this is partially solved in [24], volumetric rendering is computationally intensive task and it is best suited for uniformly sampled data sets, which need not be the case for a general finite element mesh.

A new way to visualize FEM data brings the isogeometric analysis introduced by Hughes [7], which reuses the matematical representation of the input geometry created in CAD tools during the entire engineering process, including FEM analysis itself. In [25], there is proposed an extension of this concept also for post-processing and visualization purposes.

Different approach for post-processing of FEM data can be to avoid trying to compress geometrical part (finite element mesh) and compress only result fields instead. The finite element mesh itself can be visualized using well-known methods of computer graphics and the result fields, as they can be viewed as a series of arbitrary rectangular matrices, can be handled separately by methods used for image compression. There are many image compression methods. The most commonly used are the discrete cosine transform [26] used in JPEG standard and the wavelet transform used in JPEG 2000 standard [27].

OpenCTM [28] is a 3D geometry technology for storing triangle-based meshes in a compact binary format. Its compression method is based on lossless entropy reduction. The downside of using this format is the associated decompression cost.

2.2 File formats

There are many file formats that can be used for representation of input geometry of finite element software. In fact, McHenry in [29] presents about 140 file formats for representation of 3D models. To give an example, the commonly used universal standards for representation of geometry in CAD systems are IGES[1] [30] and STEP[2] [31].

On the other hand, there are very few open formats for representation of finite element meshes and results from the finite element method. Commercial software packages, such as Abaqus [32], use proprietary formats that are intended for internal use only or their documentation is not available. The available open formats are provided by open-source projects, e.g., Gmsh [33] or ParaView [34], which are mainly used in academia. ParaView is based on the VTK file format [35]. VTK can be considered as the only universal format for the representation of results from FEM that also supports data compression, eventhough the compression is based on ZIP method, which is not very suitable for FEM results. There is also not so widely used Gambit file format [36], which supports representation of solution results.

GiD [37] is a pre and post processor for numerical simulations in science and engineering. Although it is a commercial software, its file format is documented [38] and accessible as the GiD is often connected to finite element software provided by other companies or organizations.

In [39], there is presented a possibility to convert different types of finite element mesh files to one universal format. The paper considers only ASCII file representations and the conversion method is based on regular expressions.

2.3 Web-based data management

Peng et al. in [40] propose an implementation of an engineering data access system for a finite element analysis, which utilizes the Internet as the communication channel to access the analysis results. Besides the description of the overall architecture

[1] Initial Graphics Exchange Specification
[2] STandard for the Exchange of Product model data

of the data management system and the communication between its components, the paper addresses three important aspects of any data management system: data storage scheme, data representation, and data retrieval. Although the exact type of database is not mentioned there, the relational type of database is expected to be used. Peng and Law in [41] further build on the idea and present an Internet-enabled framework that allows users easy access to the FEA core service and the analysis results by using a web browser or other application programs.

The authors in [42] and [43] advocate for the use of a relational database management system in support of finite element analysis. They also propose a new way of thinking about data management in FEA as neither extreme data sizes nor integration (with other applications over the Web) was a design concern 40 years ago when the paradigm for FEA implementation first was formed. The papers also discuss how to make the transition from a file-based to a database-centric environment in support of large scale FEA.

Chen and Lin in [44] present an Internet-based finite-element analysis framework, named Web-FEM, which allows users to access existing finite-element analysis service running on their machines from remote sites over the Internet. Weng in [45] focuses on post-processing of FEA data using web technologies. The paper points out the fact that the Web is actually the only truly cross-platform environment. The implementation of the finite element simulation as cloud computing service is discussed in [46]. The authors deal with the technical issues related to the design and implementation, as well as the issues related to the data privacy and security of the cloud services.

The benefits of web-based applications with respect to native desktop (or mobile) applications are summarized and explained in detail in [47] and [48]. An example of remote rendering service for visualization of scientific data is ParaViewWeb [49]. It is a JavaScript library that can be used to build applications with interactive 3D visualization inside the web browser. SimScale [50] is a commercial computer-aided engineering software product based on cloud computing that utilizes ParaViewWeb for the data visualization.

However, Marion and Jomier point out [51] a downside of this and similar frameworks. They require customized client setup either with plugins or external applications. Therefore, the authors present a system which uses Three.JS and WebSockets to counter these issues. The WebSockets specification [52] introduces a full-duplex connection between server and client to allow fast data transmission with low latency. Behr et al. [53] point out the draw-backs of formats used to transfer geometrical data from server to browser and propose to separate vertex coordinates from other vertex attributes. Their approach takes advantage of JavaScript typed arrays to allow the downloaded data to be passed directly to the GPU memory, which eliminates relatively slow parsing and interpretation of data stored in complex structured formats.

Chapter 3

FEA data management

Together with growing complexity of finite element calculations, the importance of management of data produced by the calculations is increasingly emphasized in both industrial and research communities. Information is often shared between multiple users, moved from one computer to another, and further transformed to enable different views over the data. Some definition of persistent and standard representation of the data is therefore required as well as the corresponding data access system architecture that allows to query the data.

3.1 System architecture

The prototype implementation of the FEA data management system is designed as a collaborative framework that can be accessed by users from different client devices. Figure 3.1 depicts the schema of the system architecture. The system consists of several independent modules. The FEM calculation itself runs on a remote server as one of micro-services[1] along with a mesh generation service, a results processing service, etc. These services are controlled by an application service that provides interface to client applications in form of REST API[2].

The architecture design relies on an abstraction provided by Platform-as-a-Service[3] computing model. No service component is tied directly to a specific machine. Hardware resources are allocated when they are needed by a service infrastructure controller. This makes the scaling and deployment of components easier and allows to focus on the problem domain instead of solving server configuration and networking issues.

[1]Microservices is a service-oriented architectural style that structures an application as a collection of loosely coupled services. The benefit of decomposing an application into different smaller services is that it improves modularity and makes the application easier to understand, develop, test, and deploy.

[2]Representational state transfer (REST) is a way of providing interoperability between computer systems on the Internet. API stands for Application Programming Interface and describes a proper way for a developer to write a program requesting services from the software exposing the API. RESTful API takes advantage of verbs provided by HTTP protocol, i.e., GET, POST, PUT, DELETE, etc.

[3]Platform as a Service (PaaS) is a category of cloud computing services that provides a platform allowing customers to develop, run, and manage applications without the complexity of building and maintaining the complex infrastructure. PaaS can be delivered in two ways: as a public cloud service from a provider, where the consumer controls software deployment with minimal configuration options, and the provider provides the networks, servers, storage, operating system (OS), middleware (e.g., .NET runtime, Node.js, etc.), database, and other services to host the consumer's application; or as a private service inside the firewall in an organisation's on-premises data center.

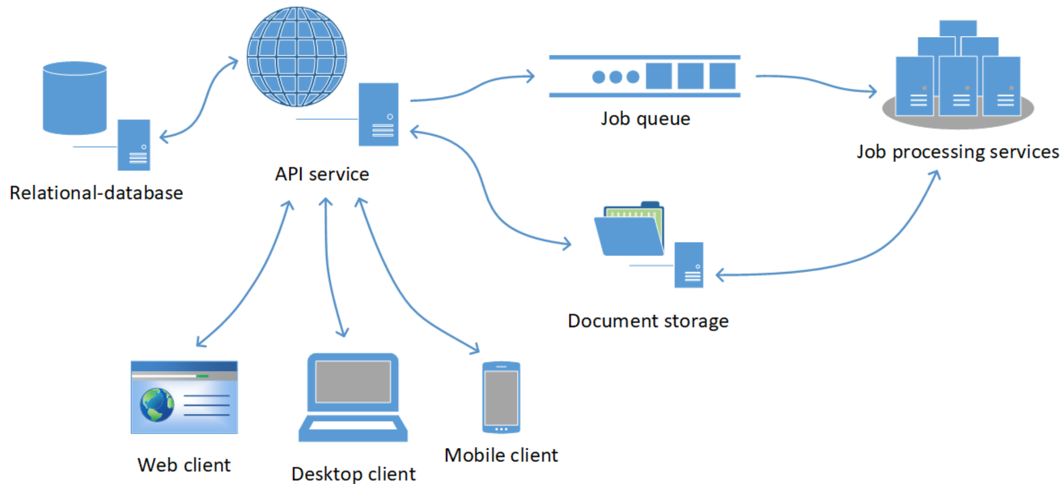


FIGURE 3.1: FEA system architecture.

The system contains two types of data storage. The relational-database type of storage is intended to store basic project-related data such as description of simulations, links to the simulation resources, information about the owner and other colaborating users, etc. The input to FEA – geometrical model, attribute assignments, and analysis parameters – can also be stored in a relational database, eventhough, storing this complex type of information in the SQL database is questionable and has its drawbacks[4].

The second type of storage is the blob storage[5] used to hold temporary files being the input or the output of particular components, especially the mesh generator and the FEM solver. The system is designed to be independent of the solver and mesh generator components, therefore this intermediate step of converting the input to proprietary file format that the components understand is necessary. In the future, it is possible to expect a gradual transition from the file-based approach to the direct connection to the database and query the input model directly. Also, the output of the calculation could be saved directly in the proposed format to represent the results in post-process-ready form.

Workflow diagram in Figure 3.2 helps to visualize the sequence of FEA steps and the transfer of data between the service components. It also reveals the basic design principle behind the microservice architecture – Separation of concerns[6]. The vertical bars denote computational intensive tasks performed by the service components. The client side in the diagram represents the presentation layer of the FEA system that the user directly interacts with. In the presentation layer, also called *frontend*,

[4]Relational databases are primarily designed to support mutations of the data, while the input model to FEA, after it is created, does not need to be changed. Relational databases are not easily scalable. They have fixed schema and tables often do not map to objects well and some object-relational mapper must be used. Also, the nature of relational databases often leads to data for different tenants, users, and projects being mixed together in the same tables, which complicates the management of data and could cause security issues. The use of a NoSQL database or a simple blob storage should be considered as an alternative if the character of the application data does not fit well into the relational database model.

[5]Blob storage is also referred to as Object storage. It is a service for storing large amounts of unstructured object data, that can be accessed over the network via HTTP or HTTPS. A blob can be any type of text or binary data, such as document or media files. Blob is an acronym for Binary Large OBject.

[6]Separation of concerns – design principle for separating system into distinct sections, such that each section addresses a separate concern.

there is spent the vast majority of time by users doing pre- and post-processing of data (which is not depicted in the diagram). The web API service, also called *backend*, is the key component that assigns work to other components. It serves as an controller for a running analysis and as an interface between the data stored in databases and the client applications.

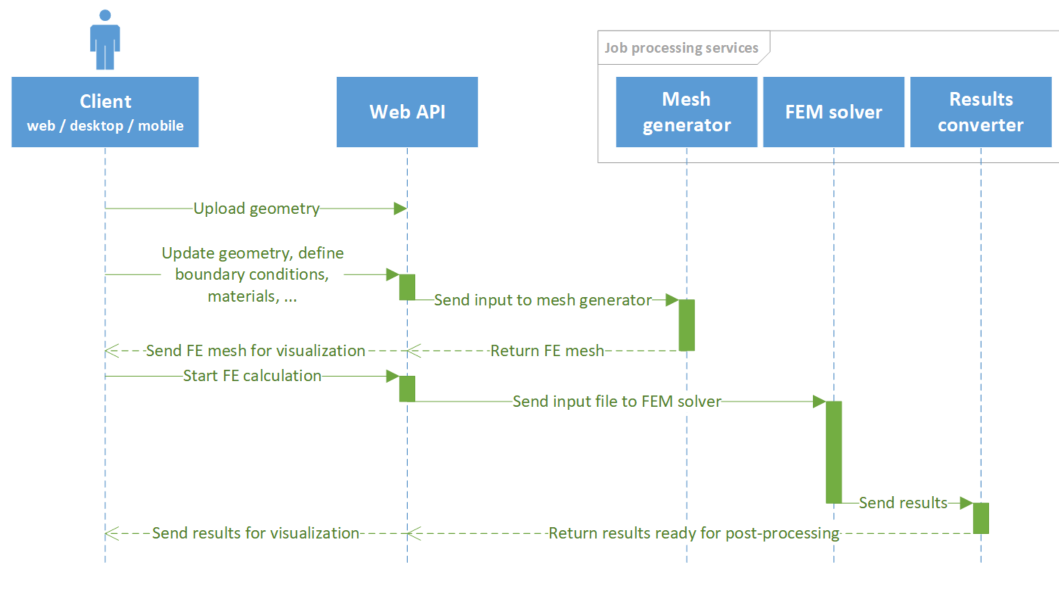


FIGURE 3.2: FEA system workflow.

The prototype implementation of the data management system follows the schema and the workflow depicted in Figures 3.1 and 3.2. The difference is that the pre-processing phase is currently excluded. The focus of the work presented in this thesis is primarily on the post-processing features and the representation of results. Therefore, the results from the existing FEM solver are uploaded into the system and the system converts them to the internal representation suitable for post-processing. To test the prototype implementation of the data management system, two client applications are created. The first is the feature-rich desktop post-processor with the support for Microsoft Windows and Linux operating systems.

The second is the simple web application that provides basic control over an analysis and basic post-processing capabilities. Its purpose is mainly to demonstrate the benefits of proposed format for storage of results when post-processing complex FEA. Its web-based implementation allows for truly cross-platform experience without the need for installation and it allows to access the analysis data even from low-end mobile devices.

3.2 Project-based data representation

Most researchers and engineers typically work independently using their own workstations while sharing the hardware infrastructure for intensive FEM calculations. The output from complex analyses are also shared as it would be costly and ineffective if each collaborator had performed her/his own calculations. Since potentially many users can access the server to oversee the analysis and to query the analysis results, a project management scheme is needed. Such data management scheme is proposed in this thesis. The overall database schema is depicted in Figure 3.3. It is

an entity-relationship diagram representing conceptual model that can be mapped on the SQL database model.

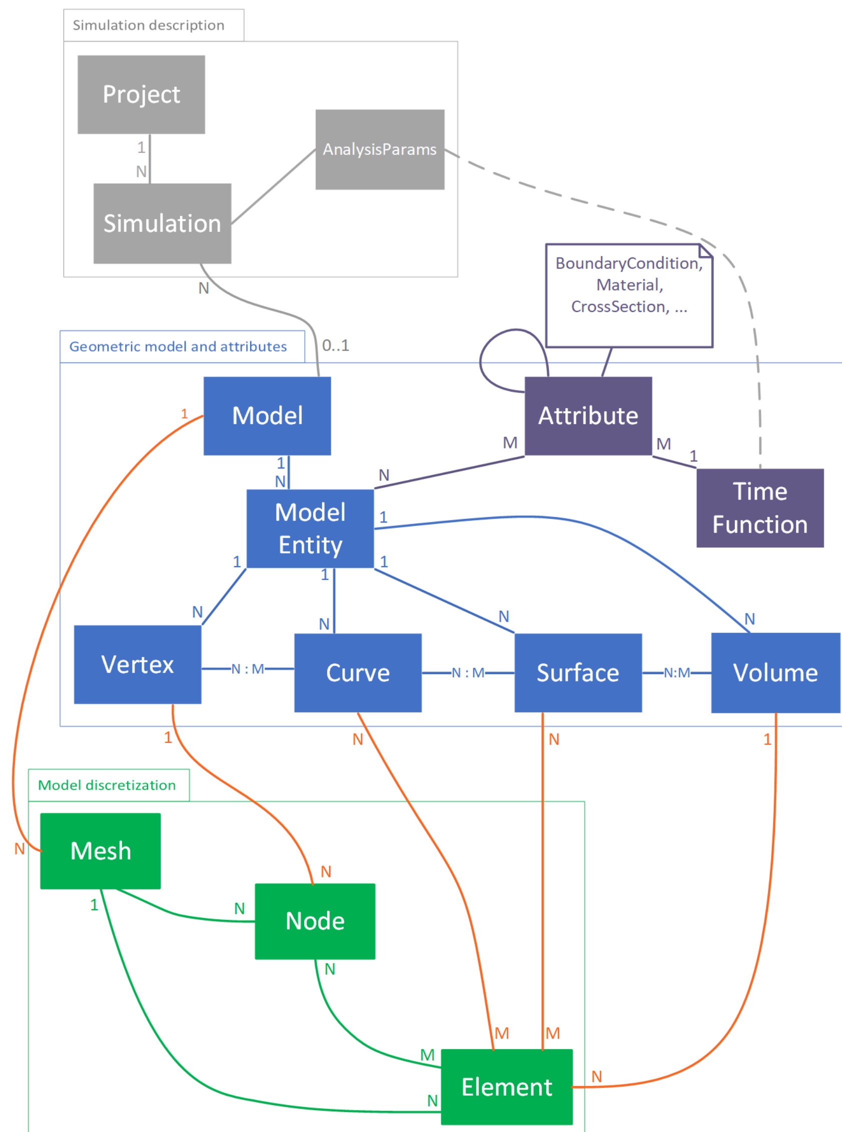


FIGURE 3.3: Database schema for FEA.

The Project entity holds the basic information about a set of related analyses. It has the name, the owner, and the list of other users that have access permissions. A project has also relation to a list of simulations. The Simulation entity encapsulates the information about a single finite element analysis. Each simulation can have different input – geometrical model, attributes (properties of model entities, i.e., material properties of volumes, initial and boundary conditions, ...), and/or parameters of analysis (e.g., number of time steps). Geometrical model and its discretizations (finite element meshes) can either be stored directly in a relational database or as a custom file in a blob storage.

3.3 Storage format for results

The conceptual model presented in Figure 3.3 can be naturally extended with the entities representing the results of the simulations. The project-based data model enhanced with the representation of simulation results is depicted in Figure 3.4 as ER diagram. The corresponding class diagram is depicted in Figure 3.5, which describes the object representation of the Solution entity. It is physically represented either by a JSON file in a local file system or by several tables in a relational database.

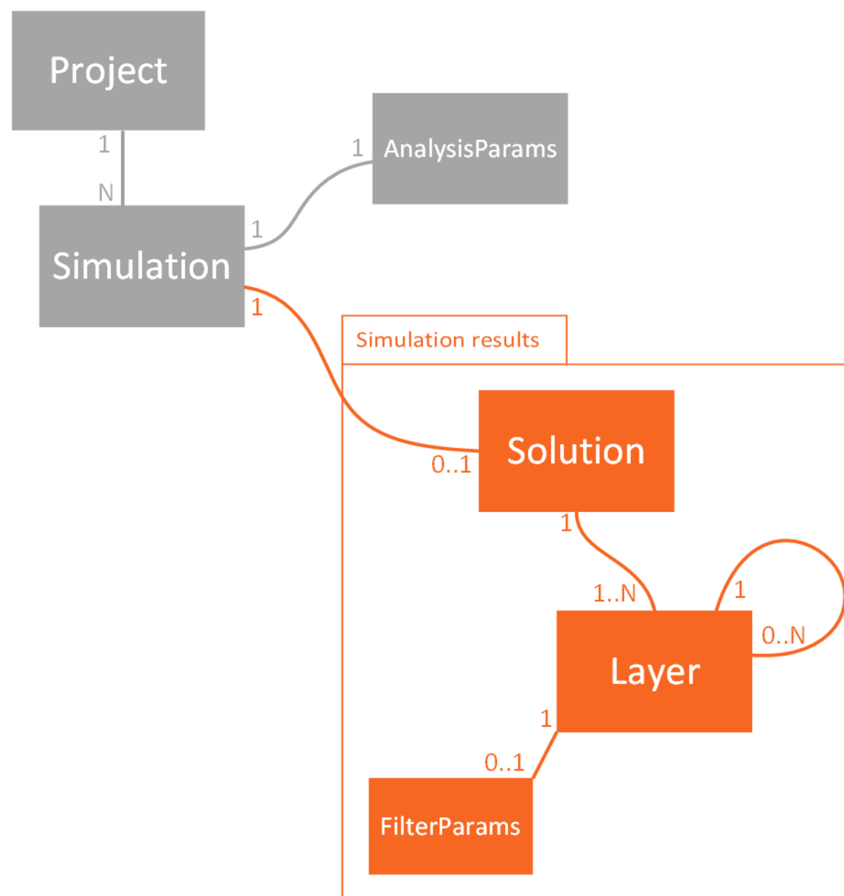


FIGURE 3.4: Database schema for FEA with results representation only (without the input model).

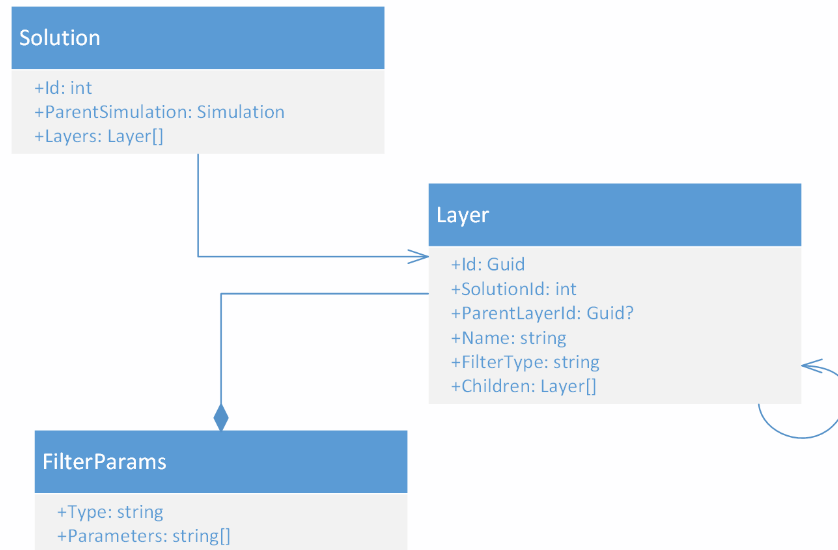


FIGURE 3.5: Object representation of simulation results.

The simulation results are represented by the Solution entity, which holds a structure consisting of the results from the FEM solver that are converted to the form suitable for post-processing. The structure has the form of a tree. A node of the tree is represented by the Layer entity. A layer is an association of a mesh and corresponding result fields and other attributes. The use of a tree structure allows to preserve the relations between the parent layers and the layers that are derived from them. The mesh that is referenced by a layer need not be the same as the mesh used as the input mesh to the FEM solver. It could be modified by the solver during the calculation, e.g., due to the use of adaptive finite element techniques [7]. Also, the resulting mesh can be further modified to facilitate the post-processor implementation, e.g., the surface representation of a 3D mesh can be generated or a number of cross-sections can be created at uniform intervals to provide a look inside the mesh. Generally, multiple views on the results can be prepared in advance before the user even starts to investigate the results.

The concept of layer is not new. It is used in existing implementations of the post-processors in FEA software packages (often named *filter* instead of layer), to represent a view on the results. However, the layers (visual filters) are usually generated on demand by specific user request and they are not kept in a persistent storage. The proposed storage format for results is built on top of the layer tree structure directly, which allows the layers to be accessed later on, even on a different device, or shared by multiple users working on the same project.

In the prototype implementation of the FEA data management system, it is supposed that the FEM solver uses its own proprietary format to represent the results from the simulation. As the system is designed to be independent of individual components, the Results converter component converts the results from the FEA solver into the standardized layer-based format suitable for post-processing. The specification of the format follows.

[7] Adaptive finite element methods can automatically refine, coarsen, or relocate a mesh to achieve a solution having a specified accuracy.

3.3.1 Format specification

All the documents that participate in the storage format are presented in JSON format. JSON is the today's standard for data transport in web technologies and also as a storage format in growingly popular NoSQL databases[8]. It is a consise textual format easily readable both for human and for computer. However, the storage format is not tightly coupled with JSON, data can be encoded in another structured format, such as XML. As mentioned above in the text, there are two types of locations the data can be stored in – **remote** and **local**. Both locations are supported in the prototype implementation. In either case, the key part of the format is the solution document. An example of the solution document is given in Listing A.1. In case of remote storage, the solution document is stored in a relational database as part of the project-based data representation shown in Figure 3.4, while the rest of documents is stored in a blob storage as group of JSON documents. In case of local storage, all documents, including the solution document, are stored in a folder as JSON files in local file system on a personal computer on which the results are post-processed.

As can be seen in the example of the solution document in Listing A.1, each layer is assigned a unique identifier, specifically a GUID[9] number. A GUID in its textual representation is also used to unambigously determine the location of layer documents. In case of a solution stored locally, a GUID is used as the name of the folder the related layer documents are stored in. For remote solutions, a GUID is used as a part of a URL when requesting resources from a database or a blob storage.

Data in each layer are stored in four types of documents. All documents are identified by the layer id and by its index within the layer (except Summary document that does not need an index as it exist in one instance per layer).

- **Summary document** contains all the descriptive information about a layer. An example of a summary document in JSON format is presented in Listing A.2. Besides a unique id, a layer has its name, which the user can choose arbitrarily, otherwise it is automatically derived from the `Filter` property. `Filter` property describes the parameters of the transformation applied to the parent layer to obtain the current layer. `ParentId` property contains the GUID assigned to the parent layer. The topmost root of the layer tree has no parent. Therefore, its `ParentId` property is null as well as `Filter` property. Default name for the root layer is "master"[10].

  `Meshes` property holds the collection of mesh descriptors. A mesh descriptor serves as an reference to an actual mesh document related to a layer. Each mesh in a layer is assigned an index. It is a number that uniquely identifies the mesh within the layer. `TimeSteps` property of each mesh contains the list of time steps for which the mesh is defined. In usual case, there is only one mesh for all time steps. In some cases, however, there can be different mesh

[8] NoSQL databases (NoSQL originally stands for non-SQL, today often for Not-only SQL) provide a way to store and query the data that is modeled in means other than the tabular relations used in relational (SQL) databases, e.g., key-value pairs, object graphs, or documents. The purpose is to offer better scalability, availability, and flexibility of the database schema. Examples of NoSQL databases: MongoDB, Redis, or Azure Cosmos DB.

[9] Globally unique identifier (GUID) or Universally unique identifier (UUID) is a 128-bit number used often as an identifier of information. GUIDs are for practical purposes unique, without depending for their uniqueness on a central registration authority or coordination between the parties generating them. While the probability that a GUID will be duplicated is not zero, it is close enough to zero to be negligible. GUID is therefore often used as the type of a primary key in databases.

[10] The format allows the existence of multiple root layers, but it turns out that it is not necessary in practice. In the vast majority of cases, there is a single root layer and it is called "master".

for each time step. Application of deformation filter to a layer yields a derived layer that contains the same number of meshes as is the number of time steps, because the deformed meshes are generated by translating the nodes of the parent mesh by displacements calculated for each time step. Even the master layer can be composed of multiple meshes, e.g., when representing the results from a simulation of construction stages. The time step number must be a unique decimal number as it serves as an identifier.

Each mesh has optional set of attributes. An attribute is an umbrella term for additional information assigned to mesh entities. It is usually a property of the input model that is propagated to the results as it can be interesting during post-processing. The most common attribute is the material number that is assigned to each element of the mesh.

`Fields` property contains a dictionary of result descriptors. Each field descriptor is introduced by the name of the field imported from the FEM results. Field can be composed of one or more components. Similarly, each component descriptor is identified by its name and enumerates the time steps in which the component is defined. Each time step descriptor contains the index of the corresponding result document as well as the index of the mesh document. Each result document always contains data for only one data component. However, the format allows that the data from multiple time steps can be gathered and stored in a single result document. The compression can be then applied on the range of time steps as a whole to achive better compression ratio. Therefore, to recover an arbitrary data component located either in a local or in a remote storage, the post-processor needs just a triplet of identifiers – the layer id, the index of a result document, and the time step.

- **Mesh document** contains the geometric representation of a finite element mesh. An example of a mesh document in JSON format is presented in Listing A.3. Positions of the nodes are encoded in `PointCoordinates` property[11]. Position of each node is a vector, whose components are floating-point numbers. Number of vector components is equal to the number of dimensions of the mesh. Vector components are flattened into a one dimensional array and converted to text using binary-to-text encoding (more about encoding in Section 3.3.3). `CellConnectivity` property[11] contains encoded list of indices of element nodes that point to the coordinate array. The number of nodes per element can be retrieved by decoding `CellTypes` property, in which the types of all elements are stored. The supported element types along with their identifiers are depicted in Figure 3.6. The identifiers are 8-bit unsigned integers (bytes) and the values match the definition of cell types in VTK file format [35] to provide basic compatibility and to ease the conversion from the VTK format.

[11] Note that the terms *node* and *element* are internally referred to by more general terms *point* and *cell*.

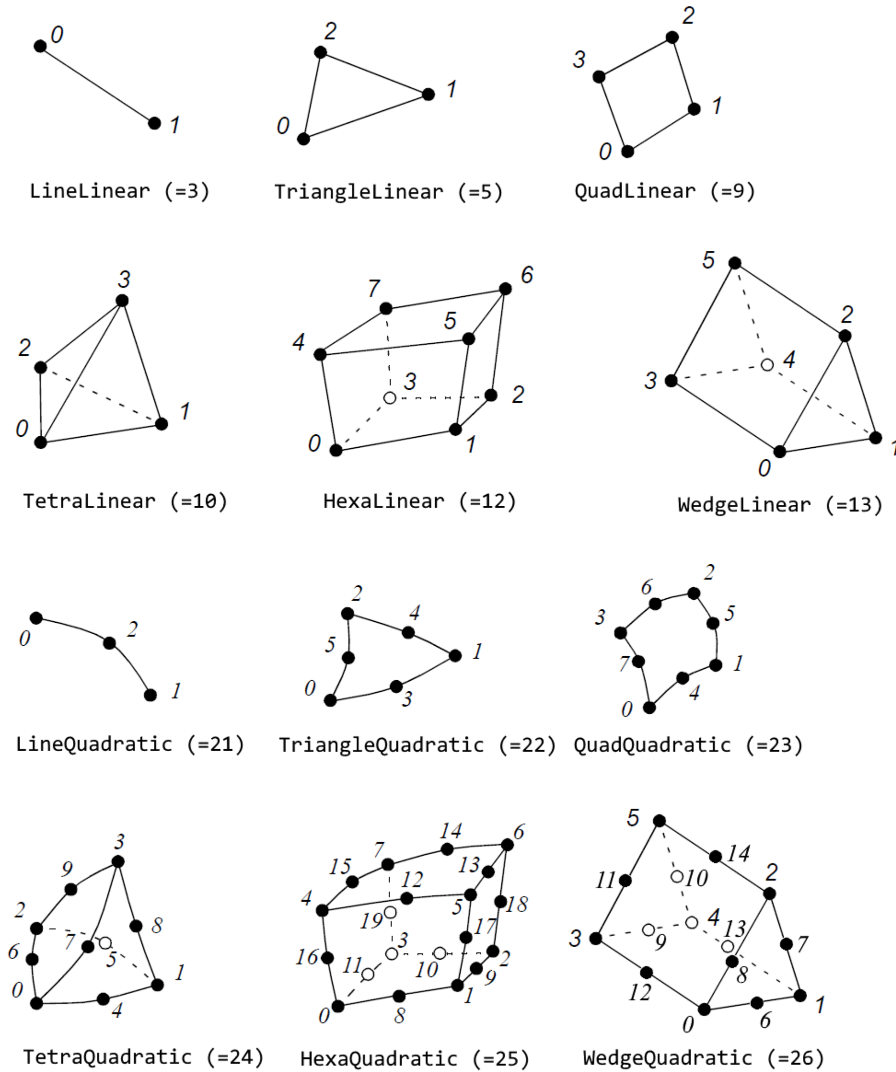


FIGURE 3.6: Element types supported in storage format. The values used for encoding element types in CellTypes array are shown in parentheses. The illustrations of elements are taken from [35].

Properties `Center` and `Radius`, eventhough their values can be calculated from the coordinate array, were added as an performance optimization for a post-processor since it is usually needed to normalize the mesh dimensions before the first rendering.

- **Attribute document** serves as a container for additional information assigned to mesh entities, e.g., material id of each finite element. An example of an attribute document in JSON format is presented in Listing A.4. `FieldName` property contains the name of the attribute. The schema of Attribute document is similar to Result document. The only difference between the two is that an attribute document is not tied to any time step, but directly to a mesh. Therefore, Attribute document does not need `TimeSteps` property. An attribute is defined in all time steps in which the referenced mesh is defined.
- **Result document** is the document that contains the result fields from FEM. An example of a result document in JSON format is presented in Listing A.5.

Result document is usually the most memory-consuming component of the storage format and it is therefore designed to support compression of data to reduce its size. There are also many options to partition the data into smaller groups. The largest granularity is achieved when the data corresponding to a single time step is stored in a single document, which allows for the lowest latency when transfering the document from the storage to the post-processor, in particular when the data are stored on a remote machine. On the other hand, the better reduction of overall size can be achieved when the data for all time steps of a field component are stored in a single document, which usually leads to better compression ratio as the compression method can find more redundancies in the data. There is also a compromise between the two approaches – to divide the time steps into multiple groups of equal or distinct sizes. This is usually connected with the input of the user who specifies the collection of key time steps of the analysis. If the key time steps are carefully chosen, the compression ratio and/or error can be even better compared to the previous case. The selected time steps are listed in `TimeSteps` property.

The value of `FieldName` property identifies the name of a physical field in the results from FEM, such as "Displacements", "Stress", "Strain", or "Temperature". The components of vector or tensor fields are identified by `ComponentName` property. Scalar fields, which consist of a single component, have its `ComponentName` property left empty or set to some generic name, such as "value". The value of `Location` property determines which type of mesh entity the data relates to. Its value can be either "Points", "CellPoints", or "Cells", which relates to nodes, element nodes, and elements, respectively. The storage format does not support the data located in integration points (Gauss points) as the storage format is intended to simplify the work for post-processor as much as possible and the extrapolation from integration points is non-trivial task. The data must be therefore transformed to one of the supported locations before converting to the storage format. Various extrapolation strategies exist and it is up to the user to select the one that is appropriate for the task. The default extrapolation strategy in the implementation of the results converter component is the projection of a value in a Gauss point to the nearest element node.

The data of the field component can be optionally compressed. Then it is converted to text using a binary-to-text encoding method and stored as the value of `Data` property. The parameters of the compression method are contained in `Compression` property and the parameters of the encoding used are in `Encoding` property (for details see sections 3.3.2 and 3.3.3).

At first glance the format seams to be overcomplicated. The data are scattered throughout a large number of documents. There are various types of documents with different schema, some information is duplicated, etc. However, the format is carefully designed to allow efficient and simple implementation of a post-processor while enabling the use of compression to significantly reduce the storage size of the results if needed. A post-processor should be able to easily decode the data and display it immediately, without the need for additional transformation, sorting, or caching. E.g., there is one-to-one mapping of the data in a result document to a mesh to be able to use the decoded data array directly as an OpenGL buffer in the graphics card. Also, the fact that the data are stored in a small structured documents of a known schema enables to index the data by a database management system and also to create efficient queries over the data. As a side effect of the data fragmentation, the resulting latency is being very low, i.e., the delay between the user request

and the data being rendered on a screen is small (which allows real-time creation of animations).

3.3.2 Compression

The storage format supports compression of results. In order to not be dependent on a particular compression method, the format is designed to be extensible. Existing compression methods can be refined and new types of compression methods can be added in the future. The only requirement is that the compression method must comply to the standard interface. The input to the compression method is required to be a matrix of numeric values. The number of rows of the matrix is equal to the number of time steps, while the number of columns is equal to the number of points in which the results are stored (nodes, element nodes, or elements, depending on the location of the data). Such auxiliary matrix is assembled for each scalar field and for each component of the vector and tensor field, and then passed to the compression method. The output from the compression method is expected to be a contiguous array of numeric values that is then encoded and stored in `Data` property. Dimensions of the original matrix are saved as parameters `Rows` and `Columns` in `Compression` property and are used by post-processor when decompressing the data.

The prototype implementation supports three compression methods. Each method is identified by a token that is assigned to `Method` property. A method is allowed to store additional values inside `Compression` property if needed.

- **SVD**. The compression method based on singular value decomposition is a very promising method as it can be applied on an arbitrary rectangular matrix and the compression error is controllable. The result of the compression method is a low-rank approximation matrix in form of its decomposition. The decomposition is then serialized into a contiguous array in column-major order. The rank of approximation matrix is stored in `Rank` property. The post-processor must perform matrix multiplication to get the original results. However, in a usual case the data for one analysis step are needed at a time, i.e., multiplication of a single row is enough. The detailed description of SVD used for compression of FEM results is in Chapter 6.

- **Wavelet transform**. The compression method using wavelet transform is inspired by the JPEG 2000 image compression standard [27]. However, unlike with the compression of images, which are uniform two-dimensional matrices, the data locations in FEM results are in a general case scattered throughout 3D space without predictable order. Therefore, some traversing path that preserves the locality of neighboring data points has to be found. Space-filling curves, such as Hilbert curve or Z-order curve, can be used for this purpose.

- **Transparent**. Transparent compression is a name for a helper method that is used primarily for testing purposes. It does not perform any type of compression of the data, it is just a convenient way of grouping the data from multiple time steps into a single document.

For the sake of completeness of this section it is necessary to mention another type of compression method that was developed – the approximation of the results from FEM, which are discrete functions, by a series of continuous polynomial functions. This method was introduced before the work on the storage format and it is not compatible with the storage format as it is based on the geometric division of the

problem domain. However, the method could be used in combination with the existing solution as an optimization of the memory consumption of the post-processor. The detailed description is contained in Chapter 5.

3.3.3 Encoding

The term encoding is used to denote the process of converting the data into a format required for storage, transmission, or further processing. In this context the term encoding denotes a sequence of steps that transforms the binary data produced by the compression method into the textual transport and storage format. The parameters of the encoding are stored in `Encoding` property of Result document.

1. The first step is an analysis of the data array produced by the compression method. The goal is to trim a repeating value at the beginning and at the end of the data array. It is a simple and fast method to further reduce the size if no compression method is used, or if the compression method fails to handle the degenerated cases, such as the data array full of zeroes or the array containing the `NaN` values[12]. `NaN` represents a missing value in a particular location and can be contained for instance in the results from a construction stages analysis.

   The value that is trimmed out of the data array is stored in property called `DefaultValue`, the index of the first significant value in the original array is represented by `Offset` property, the length of the trimmed array is in `Length` property, and the length of the original array is in `OriginalLength` property. To reconstruct the original array, the trimmed array is simply extended by the value of `DefaultValue` property.

2. The next step is the serialization of the data array from its binary representation to text. For this purpose, the base64 encoding scheme is used. The general strategy of the base64 encoding is to choose 64 characters that are members of a subset common to most text encodings, and they are also printable and human-readable. Groups of 6 bits ($2^6 = 64$ different binary values) are then converted into a sequence of characters using the encoding scheme. Considering the fact that each character is represented by 8 bits in common text encodings (ASCII, UTF-8), the ratio of output bytes to input bytes is $8 : 6$, i.e., there is 33% overhead when using base64 encoding.

3. The final step is to serialize a result document into JSON format. JSON was chosen over XML as it is more concise and it is often used as an transport format in HTTP protocol and related web technologies. It can also be used as the file format when storing in either local file system or in blob storage. JSON is also native document format in NoSQL databases. An example of a result document in JSON format is in Listing A.5.

Parameter `DataType` of `Encoding` property indicates the type of data values encoded in the data array. There are four possible values of this parameter, each representing a numeric data type that is supported by the storage format:

- **"Float32"** – single-precission floating point number.
- **"Float64"** – double-precission floating point number.

[12]NaN stands for Not-a-Number. It is a numeric data type value representing an undefined or unrepresentable value. It was introduced as a part of the IEEE 754 standard for floating-point arithmetics.

- **“Int32”** – signed 32-bit integer.
- **“UInt8”** – unsigned 8-bit integer (byte).

3.4 Post-processing

The storage format is carefully designed with respect to the implementation of a post-processor. A post-processor ought to be as lightweight as possible, with low memory and CPU requirements and low latency when loading the data. It should preserve the state of data visualization between sessions and allow to share the state among users or devices. To allow this experience, the storage format must support the representation of the visual filters and a post-processor itself should be decoupled from the filter generation. There must therefore exist an infrastructure that generates the filters and serves the desired data on demand. This infrastructure should support deployment either on a local or a remote machine. As these requirements are satisfied, the post-processor is just a thin client that communicates with a server and visualizes the responses.

The data in the proposed storage format are structured into the form of a tree. Each node of the tree is called “layer” and the whole tree is called “layer tree”. Typical layer tree consist of the master layer and zero or more derived layers. The layer tree corresponding to the example of the solution document in Listing A.1 is depicted in Figure 3.7. The master layer is the direct product of the conversion from the results produced by the FEM solver. Derived layers (listed in `Children` collection) are the products of visual filters applied on the master layer. Most filters deal with the geometry of the parent layer. The mesh of the parent layer can be cut or deformed to a new shape. The corresponding attributes and data fields are filtered or interpolated to match the underlying geometry.

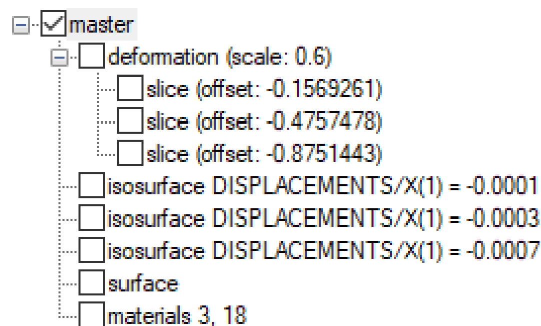


FIGURE 3.7: Diagram of a layer tree.

The visualization of the master layer is in Figure 3.8.

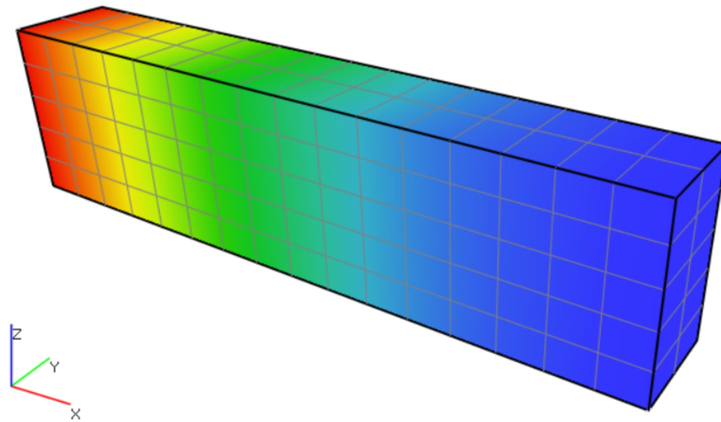


FIGURE 3.8: Visualization of a master layer with displacements in Z axis.

The prototype implementation contains the following types of filters:

- **Deformation** – Deformation filter performs the translation of the nodes of the parent mesh by the values of a vector field, usually the displacements field. The product is a layer that contains a deformed mesh as the one shown in Figure 3.9.

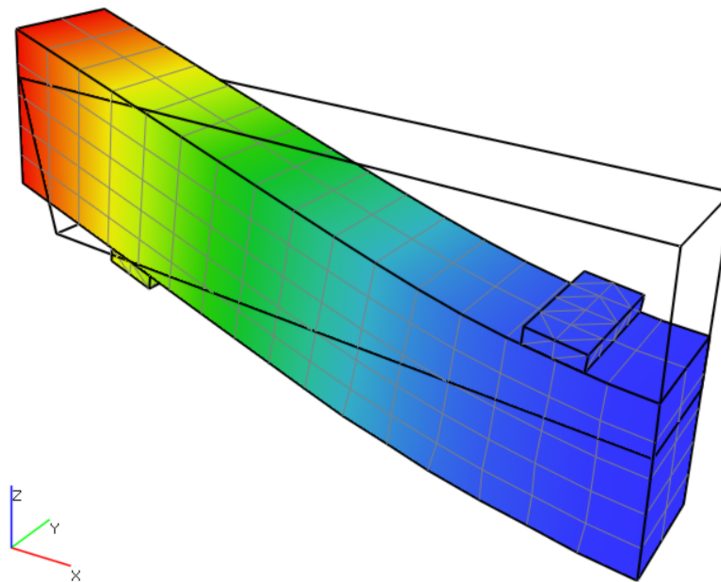


FIGURE 3.9: Visualization of a deformation layer with displacements in in Z axis.

As there is one-to-one mapping of the data values between the data locations in the parent mesh and the deformed mesh, there is no need for any transformation of the data or attributes. Also, to avoid the unnecessary copy of the data, the storage format supports a mechanism to share the data between a parent layer and its child layers. For this purpose, optional properties `DataFallbackLayerId`, `AttributeFallbackLayerId`, and `MeshFallbackLayerId` are available in Summary document. A post-processor checks the value of the

fall-back properties before each request for a result, an attribute, or a mesh document to determine the layer id that should be used for the lookup.

The geometry of the deformation layer is created once the deformation filter is applied on the parent layer. Therefore, the scale of the deformation has to be carefully selected as it cannot be modified after-the-fact during the post-processing. This is considered as a disadvantage of the chosen approach. The system helps to mitigate this problem by calculating a reasonable default value for the deformation scale (it is set to 10% relatively to the dimensions of the original mesh).

- **Slice** – Slice filter generates a two-dimensional cross section through a three-dimensional mesh. A cross section is defined by a plane using a normal vector and an offset. The offset is a distance between the plane and the origin of the coordinate system. An example of a solution with multiple slice layers is depicted in Figure 3.10.

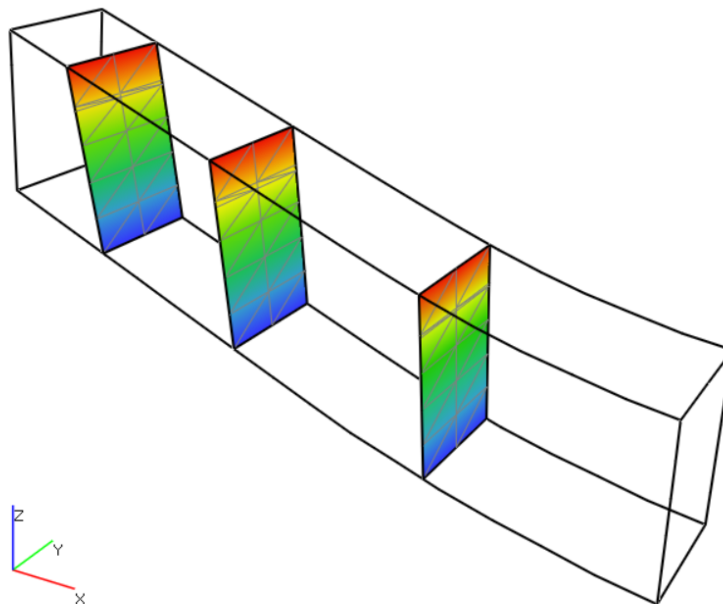


FIGURE 3.10: Visualization of multiple slice layers with displacements in X axis.

Natural improvement of the slice filter could be the ability to automatically generate a sequence of parallel slices with a uniform distance between them. This will be a subject of the future work.

- **Iso-surface** – Iso-surfaces are two-dimensional surfaces following a constant value of a data component. They help to visualize the gradient of a field inside the volume of a domain. Examples of three iso-surface filters are shown in Figure 3.11.

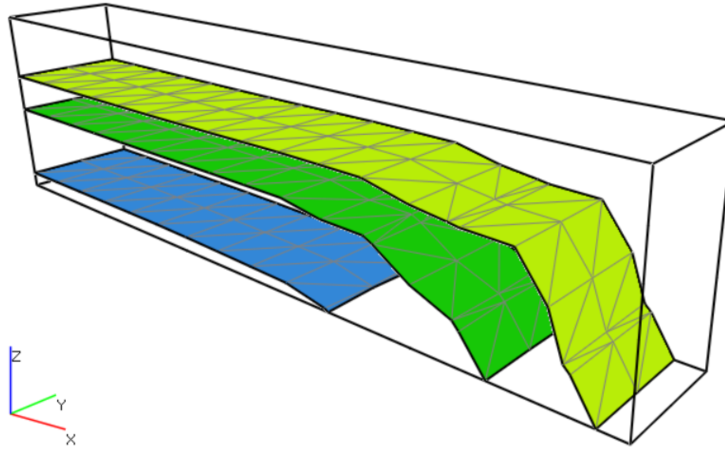


FIGURE 3.11: Visualization of multiple iso-surface layers with displacements in X axis.

- **Attribute selection** – Attribute selection filter allows to choose mesh entities (usually elements) having a particular attribute assigned (e.g., material property) and propagate them into a new layer, filtering out the other entities that does not have the attribute assigned. In Figure 3.12, there is shown how the attribute selection filter can be used to create a layer that contains only the one-dimensional elements of the original mesh.

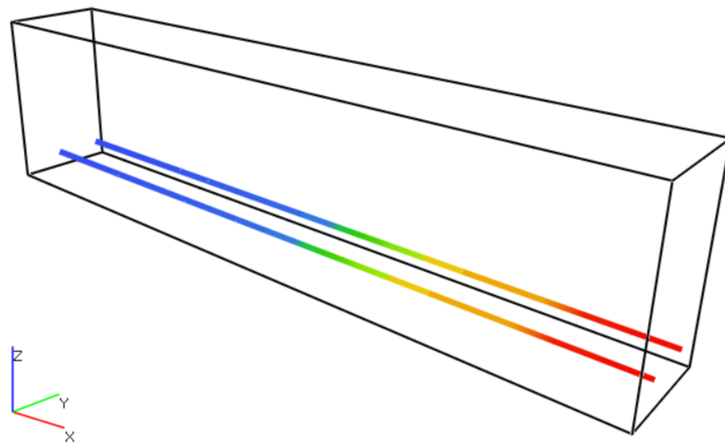


FIGURE 3.12: Visualization of an attribute selection layer.

- **Surface** – Surface filter is a helper transform that produces a surface representation of a given mesh and maps the corresponding data and attributes onto the surface. It allows to significantly simplify the implementation of a post-processor. The prototype implementation of the web-based post-processor is able to visualize only the mesh that consists of simple triangles. To be able to visualize all types of elements (as shown in Figure 3.6) the post-processor requests a creation of a surface layer for each layer containing 3D elements.

The hard work is offloaded to the server and the implementation of the post-processor is kept as simple as possible with low memory and CPU requirements. Figure 3.13 shows the visualization of a surface layer. Notice the triangles instead of quads on the surface. To preserve the identity of the original finite elements, the ids of elements can be tracked using a dedicated attribute document.

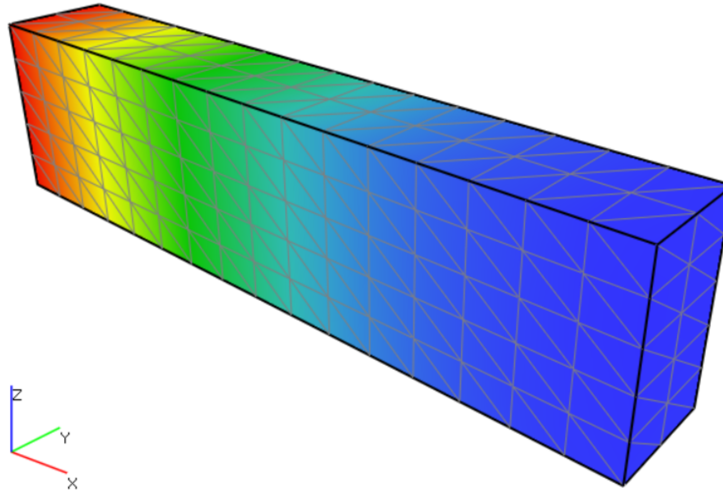


FIGURE 3.13: Visualization of a surface layer.

There are also ideas for other filters that are not yet implemented. E.g., Clip filter that creates a section through a mesh, but preserves (unlike Slice filter) the geometry behind the cut plane, or Stream-lines filter that enables to visualize a flow field by creating the tangent lines to the velocity vectors of the field.

As stated, the post-processor does not need to implement the visual filters by itself. The filters are the integral part of the storage format and there is an dedicated service component that does all the manipulations with the format. The post-processor provides just the user interface for the filter definition and for the layer tree management. It provides a tree view that displays a current layer tree, similar to the one in Figure 3.7, where the user can turn on or off a particular layer or change the display mode of the current layer. When a filter layer is selected, the post-processor also displays the outline of its parent layer to be able to see the relationship of both layers. E.g., a deformation layer can be compared to the undeformed geometry, or the position of a slice layer can be put into context of its parent mesh.

The post-processing involves a number of features that are not handled externally as they cannot be represented persistently by the storage format. An example of such feature is the visualization of vector fields using arrows as shown in Figure 3.14. It is not a good candidate for a layer filter as vector arrows does not have static geometry. The user may want to change the scale of the arrows or move the origin of the arrows. The downside is the requirement to load all the components of a vector field into the memory at once to be able to build the visual representation. Therefore, this is only implemented in the feature-rich desktop post-processor for the time being.

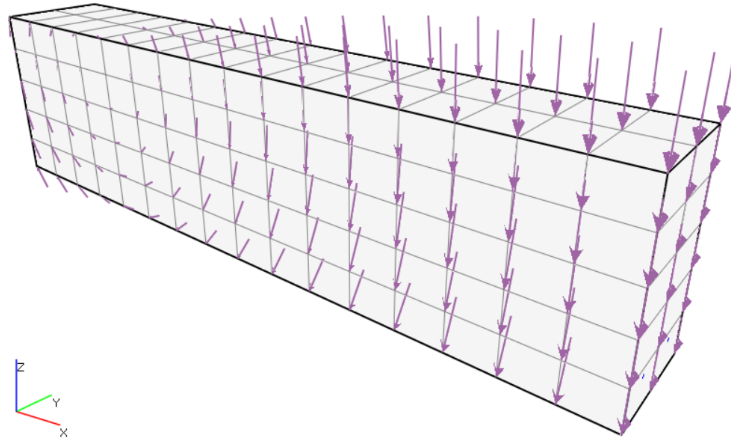


FIGURE 3.14: Example of the vector field visualization.

One of the key features of a post-processor is the ability to map a data value, which is a floating-point number, to a color in the color scale that is used to render the mesh surface. The mapping of the color can be configured. The implementation contains several predefined types of color scales including gray-scale, separated-by-zero[13], or user-defined color scale type. There are two modes for calculating the color between the data locations – standard interpolation and quantization, i.e., dividing the whole color scale into a few buckets, each corresponding to a different range of data values, and assigning a constant color for each value in the range. In the implemented post-processor, this technique is called the iso-areas color scale and an example is shown in Figure 3.15. The iso-areas feature cannot be represented using the storage format. To avoid an ad-hoc generation of the corresponding geometry, which would be difficult to implement and would significantly degrade the performance, iso-areas feature is implemented solely inside the OpenGL pipeline using the special pixel shader[14].

[13]Separated-by-zero color scale type has three control points. The middle point is assigned the value zero and mapped to white color. All the negative values are mapped to a shade of blue color and the positive values are mapped to a shade of red color.

[14]A shader is a type of computer program that calculates rendering effects on graphics hardware. Shading languages are usually used to program the programmable GPU rendering pipeline. There are many types of shaders. Pixel shaders, also known as fragment shaders, compute color and other attributes of each "fragment" – a technical term usually meaning a single pixel.

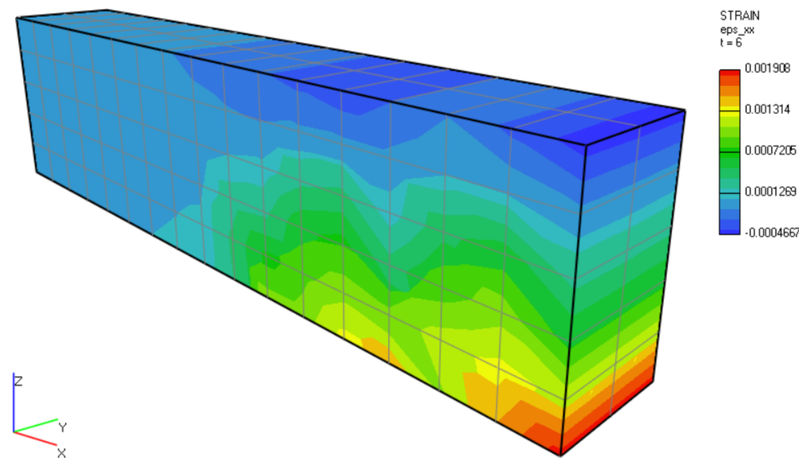


FIGURE 3.15: Iso-areas visualization of the xx component of the stress tensor.

3.5 Implementation details

The components of the data management system and their relations were outlined in Figure 3.1 and Figure 3.2. This section discusses the implementation details of the application components based on the system design set out in the previous sections. The implementation of each component is discussed, including the programming languages, technologies, external providers, or libraries used in the prototype implementation of the application.

The architecture of the data management system is designed as modular and it consists of several isolated services. The full list of services needed either for remote or local post-processing follows below. If there is no need or desire to run the system in the distributed environment, one can simply use only the components necessary to post-process the simulation results locally. These are two components – the FEM results converter and the desktop post-processor.

It is worth noting that the prototype implementation uses the Microsoft Azure public cloud as an enviroment to test and demonstrate the whole system. However, any other cloud provider can be used in production. The system can also be run on-premises if there is any concern about data privacy.

Project data storage

Project-related data are stored in the SQL database. The ER model depicted in Figure 3.4 is mapped on the database schema consisting of tables Projects, Simulations, Solutions, and Layers. There is also the table Users that holds the information about the owners and collaborators connected with a particular project.

Microsoft SQL Server is used as the underlying relational database management system – specifically, its cloud-based version that runs as a platform-as-a-service in Microsoft Azure cloud.

FEM results storage

FEM results converted to the proposed format are represented by a series of JSON documents. In case of the remote storage, documents are stored in the blob storage.

The Microsoft Azure storage account is used as a provider for the blob storage service. The name of each document follows predefined convention so it can be easily constructed from the document index retrieved from the solution object stored in the relational database. The name of each blob follows the pattern:

`{Layer-id-GUID}/{document-index}.{document-type}.json`

In case of the local storage, the documents are stored in the local file system. The documents corresponding to a single layer are stored in a single folder and the paths and the names of the files are in the following form:

`{path-to-folder}/{Layer-id-GUID}/{document-index}.{document-type}.json` [15]

FEM results converter

FEM results converter is the core component of the data management system. It does implement all the necessary operations that query or manipulate the data in the proposed layer-based storage format. It is designed as an independent service that can run either in a remote or a local environment. It supports three deployment options:

1. It works as a stand-alone console application that exposes command-line interface.
2. It can run as a background task on a remote server, communicating via HTTP requests.
3. It can be directly referenced as a library by a post-processor, which eliminates the overhead of the communication interface.

The FEM results converter component supports the following commands. (The commands can either be called directly by the post-processor, triggered by the message sent over the network, or even entered manually to the terminal as command-line arguments.)

- **import** – converts the output from the FEM solver to the universal storage format. (GiD and VTK file formats are currently supported.) The command creates a new solution with a single layer in it – the master layer.
- **filter** – creates a new layer by applying the requested filter on the specified parent layer.
- **list** – enumerates all the layers in the solution. It is used to show the current layer tree to the user.
- **delete** – deletes a layer.
- **help** – displays more information on a specific command.

The filter command at first downloads the solution object from its storage location. Then it retrieves the documents corresponding to a specific layer from either the remote blob storage, or the local file system. Then it applies the requested filter on the layer's documents and generates the new filter layer consisting of transformed data. Finally, the layer tree is updated to contain this new layer. As soon as

[15] The example of the path to a document in the local file system:
`C:/Projects/Project1/3fe47370-fb09-4c17-9bb7-0e75807d531f/4.result.json`

the new layer is generated, it is considered as immutable and cannot be modified by any command. This simple principle prevents the conflicts when accessing the layer documents simultaneously from multiple locations, eliminates data consistency issues, and helps to avoid any need for sychronization mechanisms.

Both the import and the filter commands support the compression of the layer data. There is an optional argument for specifying the name of the compression method and the additional list of parameters that varies with the chosen compression method. E.g., the SVD compression method accepts either "error" or "size" parameter (see Chapter 6 for details). To support the SVD compression, the converter component references RedSVD library [54] that provides the compression method with the optimized algorithms for matrix decompositions.

The FEM results converter is implemented in C# language and targets .NET Core framework, which ensures cross-platform availability. The component can be either hosted as a webjob inside the Azure Web app service (on a dedicated virtual server), or run on demand as a serverless[16] application using the Azure Functions infrastructure.

Web API service

The web API is a simple service that provides the access to the project-related data to all client applications in form of REST API. It also serves as the backend for the web post-processor.

ASP.NET Core [55] was chosen for the implementation as it is cross-platform, high-performance, and open-source framework for building web applications. As an object-relational mapper that enables to simplify the writing of data-access code, the Entity Framework Core [56] is used.

If there will be a need in the future for the system to be extended to handle the pre-processing of the input to the FEA, the web API will be the service that will need to be extended to provide all the operations handling the input model.

Web post-processor

The web post-processor is a simple frontend application that directly communicates with the web API service to provide the user with the remote access to the projects he is involved in, the state of simulations running on the remote machine, and also the basic visualization of simulation results. Its implementation highly depends on the storage format design. In fact, the storage format itself was designed to allow efficient implementation of a thin presentation client suitable for resource-limited devices. As an example can serve the requirement that the surface representation of a 3D mesh has to be generated in advance on the server and represented as a special layer in the storage format. The reason is to avoid the expensive ad-hoc generation of the surface on the client.

[16]Serverless computing is a cloud computing execution model in which the cloud provider dynamically manages the allocation of machine resources. Pricing is based on the actual amount of resources consumed by an application, rather than on pre-purchased units of capacity. Serverless computing still requires servers, hence the name is misleading. The name "serverless" is used because the server management and capacity planning decisions are completely hidden from the developer.

It is implemented as a single-page application (SPA)[17] using Aurelia client framework [57] and it is written in Typescript language. It uses Bootstrap library [58] to enable resposive layout of HTML components. THREE.js library [59] is used for the 3D visualization. THREE.js is a high-level JavaScript library built on top of the low-level WebGL[18] standard.

Desktop post-processor

The desktop post-processor is the feature-rich visualization tool that allows to visualize the data stored either in a local or in a remote location. Besides the basic support for the layer-based storage format offered by the web post-processor, the desktop version implements more advanced operations, such as generation of the surface representation of 3D meshes, automatic detection of significant edges, free movement of the camera, manual hiding of arbitrary elements, etc.

The application is written in C# language and targets full .NET framework. It supports both Microsoft Windows OS and Linux OS (tested on Debian-based Linux distributions; Mono framework [60] is required). It uses OpenTK library [61] for 3D visualizations, which is a low-level wrapper for OpenGL.

The design and implementation of the efficient representation of mesh surface and its construction algorithm is described in detail in Chapter 4. Chapter 5 discusses an alternative method to SVD compression – the approximation of FEM results by polynomial functions, which is implemented only in the desktop post-processor and it is not related to the proposed storage format.

[17] A single-page application (SPA) is a web application that interacts with the user by dynamically rewriting the current page rather than loading entire new pages from a server. This approach avoids interruption of the user experience between successive pages, making the application behave more like a desktop application. In an SPA, either all necessary code – HTML, JavaScript, and CSS – is retrieved with a single page load at the beginning, or the appropriate resources are dynamically loaded, usually in response to user actions.

[18] WebGL is a cross-platform web standard for a low-level 3D graphics API based on OpenGL ES, exposed to ECMAScript (JavaScript) via the HTML5 Canvas element.

3.6 Results and evaluation

This section contains screenshots and performance evaluation of both the desktop post-processor and the web post-processor. In Figure 3.16, there is a screenshot of the desktop post-processor that demonstrates the ability to connect to the remote web API and retrieve the list of solutions stored in the database.

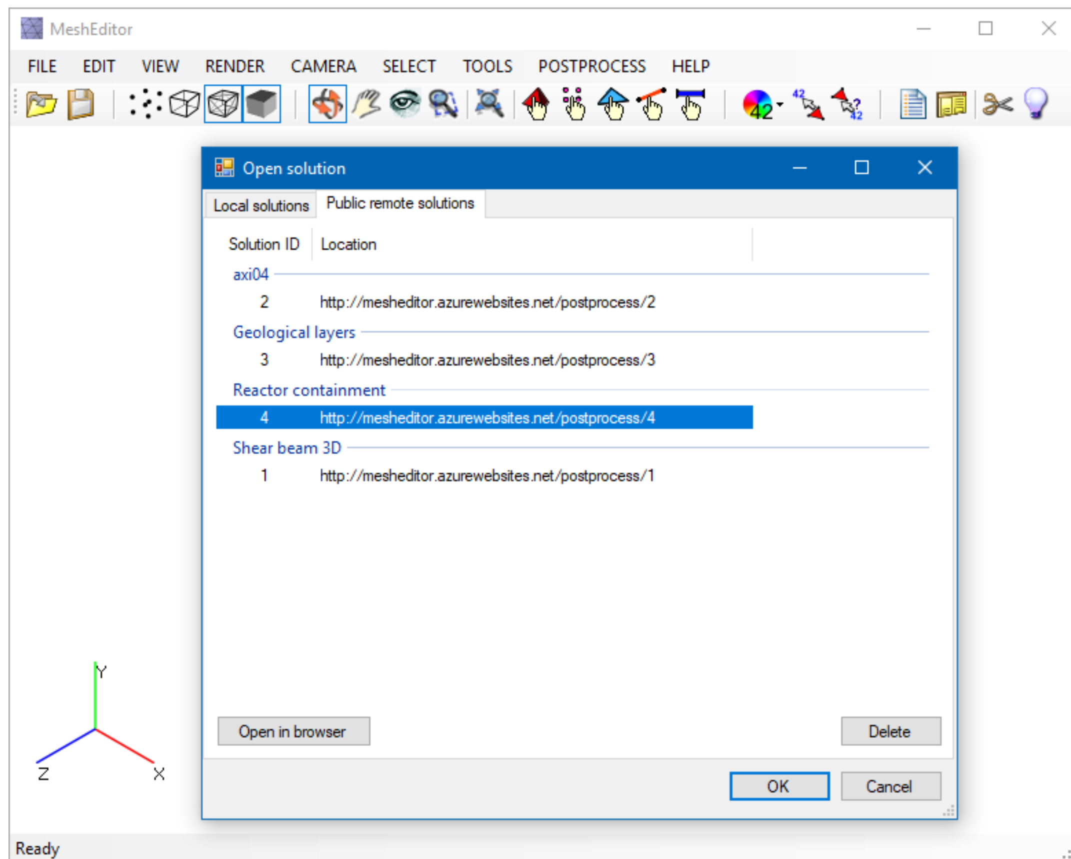


FIGURE 3.16: Desktop post-processor screenshot. Demonstration of connection to the web API (listing of remote solutions).

Figure 3.17 shows the visualization of the master layer of the selected solution in the desktop post-processor window. The left panel contains the layer tree view that allows to choose the layers to show in the main window. There are also the options to visualize components of scalar, vector, or tensor data using the color scale on the mesh surface. Vector fields can be also visualized using arrows in the data locations pointing in the direction corresponding to the vector field. The top panel contains the tools for manipulation with the mesh, changing the camera view, selecting mesh entities, and assigning attributes to them.

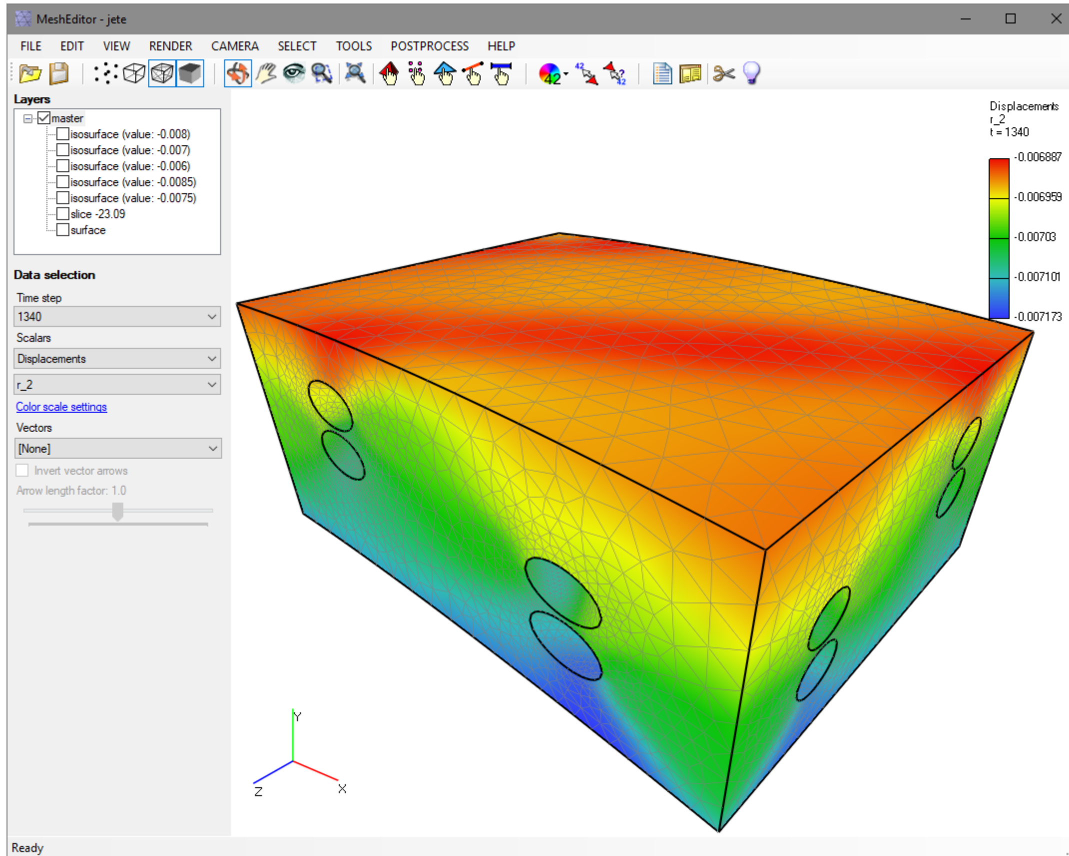


FIGURE 3.17: Desktop post-processor screenshot. Visualization of a master layer.

The post-processor also provides the user interface for definition of the parameters of a visual filter that the user wants to be applied on the selected layer. The parameters are sent as a part of the filter command that is sent to the FEM format converter component that generates the new layer. The layer tree is then updated and the new layer is displayed. The examples of slice and iso-surface layers are shown in Figure 3.18 and Figure 3.19, respectively.

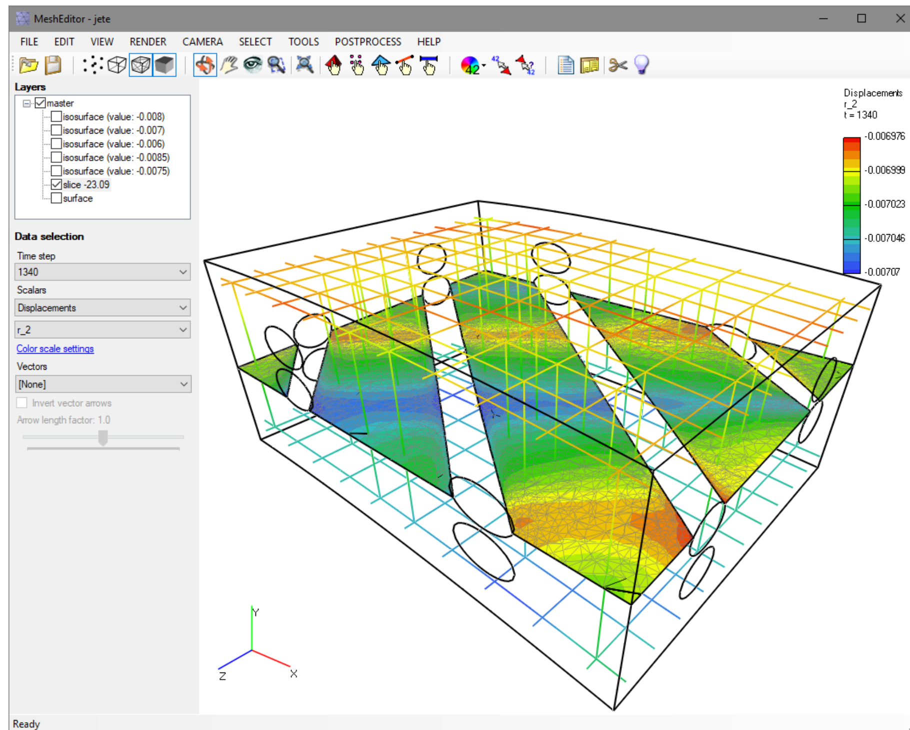


FIGURE 3.18: Desktop post-processor screenshot. Visualization of a slice layer.

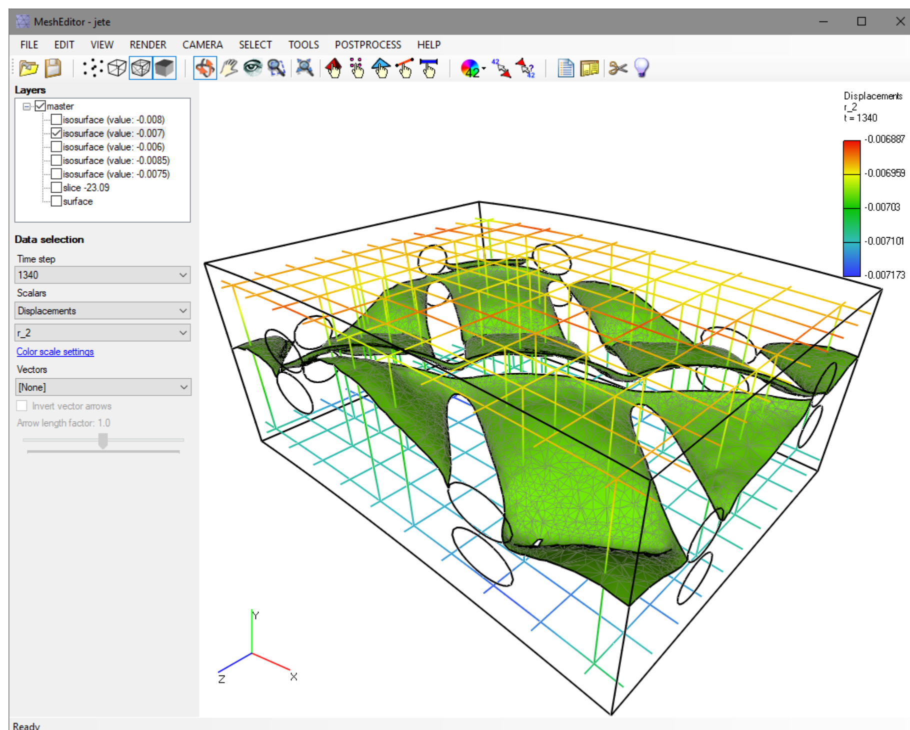


FIGURE 3.19: Desktop post-processor screenshot. Visualization of a iso-surface layer.

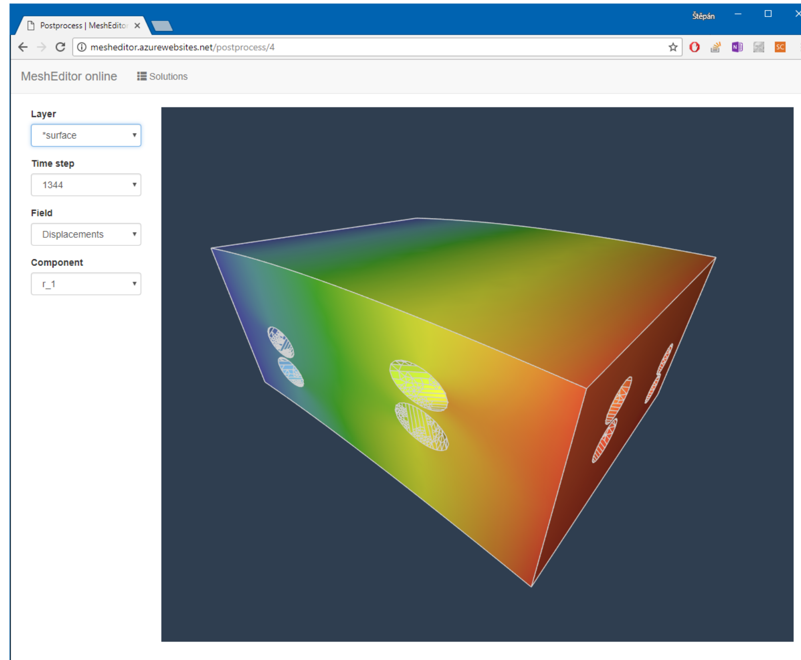


FIGURE 3.20: Web post-processor screenshot. Visualization of a surface layer.

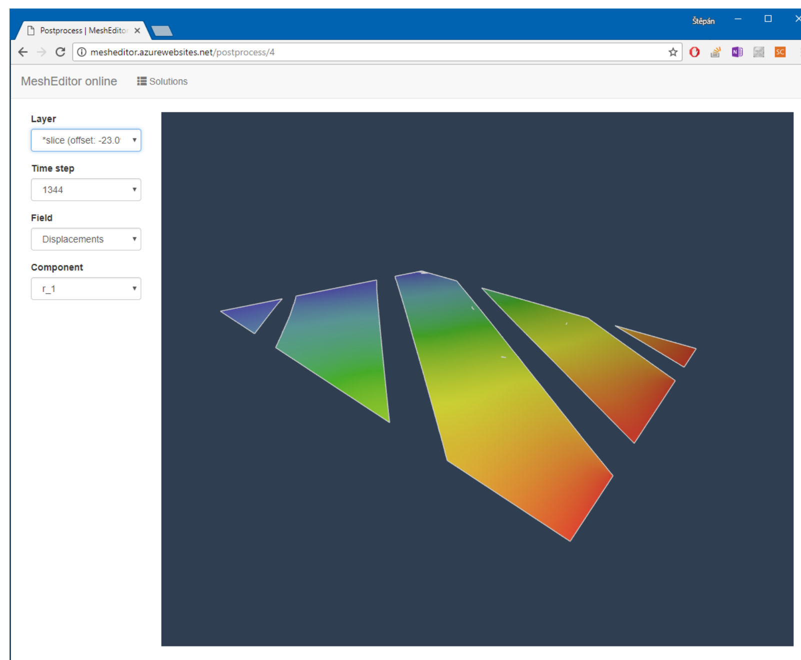


FIGURE 3.21: Web post-processor screenshot. Visualization of a slice layer.

Figure 3.20 shows the web post-processor that is connected to the same database as the desktop post-processor. The user interface also allows to select the layer of the current solution and the data component that should be visualized on the mesh surface. Figure 3.21 contains the visualization of a slice layer.

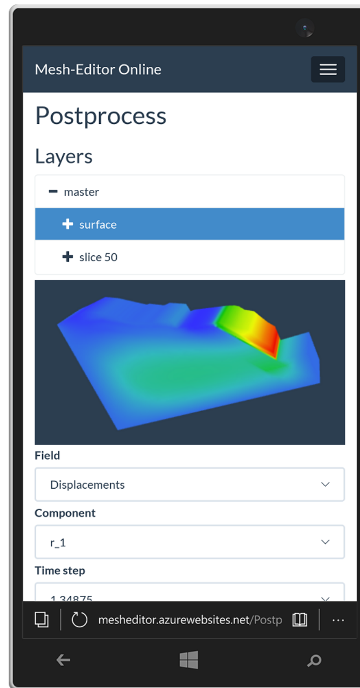


FIGURE 3.22: Web post-processor screenshot taken on a mobile device.

Figure 3.22 demonstrates the fact that the web post-processor is fully capable of running on a mobile device with limited CPU and memory resources. The web application is designed to support multiple form factors and can be used comfortably even on a small device with touch screen.

Generation of a new filter layer is offloaded to the server and can took tens of seconds for large solutions. However, it is a one-time operation because the result is stored in a persistent storage. The subsequent requests for documents containing layer data are very fast. For a common mesh having around 300000 elements the web post-processor needs only tens of milliseconds to render layer geometry and selected data component on the mesh surface, even though it downloads all the necessary data from a remote server over the Internet. This speed allows to avoid caching of the data on the client device and helps to keep the memory consumption of the proposed post-processors very low (tens of megabytes for common mesh sizes) compared to traditional post-processors that need to load and process all the results from a file in order to display selected information.

Chapter 4

Efficient methods to visualize finite element meshes

This chapter contains the description of the methods to process and visualize large finite element meshes and presents the design and implementation details of the desktop post-processor. The objectives for the implementation are high responsivness, comfort of use, and low memory consumption of the final application. The text in this chapter is also published in [62].

4.1 Theoretical background

The geometric model that serves as the input to finite element analysis is usually described by its boundary. It has to be discretized for the further use. The process of discretization is called the mesh generation [4, 5]. The output of a mesh generator is a finite element mesh corresponding to the input domain. The elements are the basic components of a mesh and there are several types of them. The frequently used tetrahedral elements consist of four triangular faces described by three edges while an edge is defined by two nodes.

It is sufficient to have only the list of nodes with their coordinates and the list of elements with references to their respective nodes for the mesh representation. Other entities (e.g., faces, edges) are usually omitted from the mesh generator output. However, the mesh editor has to handle them all, therefore, they must be created while loading the mesh. It is necessary to know all the kinds of the meshes that can be used as an input for the mesh editor implementation and especially for the design of the internal mesh representation. Meshes can be divided into several groups according to different criteria. The mesh classification is described in [14]. The most basic form of mesh classification is based upon the connectivity of the mesh: structured or unstructured.

A structured mesh, also known as a grid, has a regular internal structure. Elements in the mesh are simply addressable due to the uniform distances between nodes. It restricts the element choices to quadrilateral in 2D or hexahedra in 3D. The regularity of the connectivity allows us to conserve space since neighborhood relationships are defined by the storage arrangement.

An unstructured mesh is characterized by irregular connectivity. It allows for any possible element that a solver might be able to use. When compared to the structured meshes, the storage requirements for an unstructured mesh can be substantially larger since the neighborhood connectivity must be explicitly stored.

Other mesh classification is based upon the **dimension** and the type of elements present. Depending upon the analysis type and solver requirements, meshes can be composed of one-, two- or three-dimensional elements. **Homogenous** meshes

contain elements of the same type and dimension. **Hybrid** meshes are composed of elements of different type and/or dimension, e.g., tetrahedral mesh with 1D bars.

Additional classification can be made upon whether the mesh is **conformal** or not. An intersection of any two elements is either by a face, an edge or a node in conformal mesh. A non-conformal mesh contains for instance two quadrilaterals sharing two edges or two quadrilaterals sharing only a half of one edge. Non-conformal meshes are usually created during distributed generation of meshes from sub-domains and can cause issues during creation of the surface representation of non-conformal meshes. This issue is described in detail in section 4.2.

Three-dimensional meshes can be replaced for the purpose of visualization with its surface representation. The elements (or parts of elements) that are hidden inside the volume of the mesh can be omitted. Visualization is much more efficient then. Most of the operations on entities, such as selection or setting of properties, can also be made on the mesh surface. The implementation of cuts through the volume is a problem. In order to show the entities on the cross-section, the surface representation must be regenerated each time. Making a cut is therefore a little more computationally intensive but it is outweighed by the fact that 2D surface representation is sufficient to handle any finite element mesh.

Because of the fact that the surface of both two-dimensional and three-dimensional elements is formed by either a triangle or a quadrilateral to represent the whole mesh, it is sufficient to use these two shapes. The surface representation must also include edges and vertices which the faces are formed of. The one-dimensional elements will be dealt with separately. When considering the internal representation of a mesh it is necessary to take into account the memory requirements. It should be noted that closed 3D mesh (not counting boundary elements) with homogenous structure and with n elements has approximately $6n$ edges, $10n$ faces and $5n$ tetrahedral elements.

Most of the triangles and the edges will be inside the volume and can be therefore discarded after surface generation. The number of the surface entities is closely related to the geometrical shape of the domain. However, the number is significantly lower than the number of all entities for most meshes. The common operations on the mesh, e.g., to find neighboring faces, need complete topological data about the original mesh. The input file usually does not contain information about connections between elements. Therefore it must be determined while loading the mesh from the input file.

The data structure called **Winged edge** is used to store this kind of information. It is a widely used data structure in computer graphics especially for modeling practice [63, 64, 65]. It describes explicitly the geometry and topology of faces and allows fast traversing between faces, edges and vertices on the surface through a structure similar to the linked list. Traditional winged edge data structure is represented by edge table. Each entry in the edge table contains these references: start vertex and end vertex, left face and right face, the predecessor and successor edges when traversing its left face, and the predecessor and successor edges when traversing its right face. Clockwise ordering (viewing from outside of the polyhedron) is used for traverse. Note that if the direction of the edge is changed, all entries in the table must be changed accordingly. Also, if some faces of a solid have holes, the above form of winged edge data structure does not work. To make it work, ordering of the edges must be changed or some auxiliary edges must be added to surface representation. All these changes are difficult to implement efficiently. And this type of winged edge data structure cannot be used for non-conformal finite element meshes. The basic composition of the winged edge data structure that describes polygon meshes

is widely used in computer graphics. However, the increased memory requirements compared to representations like the simple list of vertices and elements is the disadvantage. Moreover, the winged edge structure is based on dynamically created objects and therefore fragmentation of the memory can occur.

Figure 4.1 shows the adjusted data structure describing mesh surface based on traditional winged edge schema, but eliminating some of its deficits. This structure is more suitable to use in finite element mesh scenario.

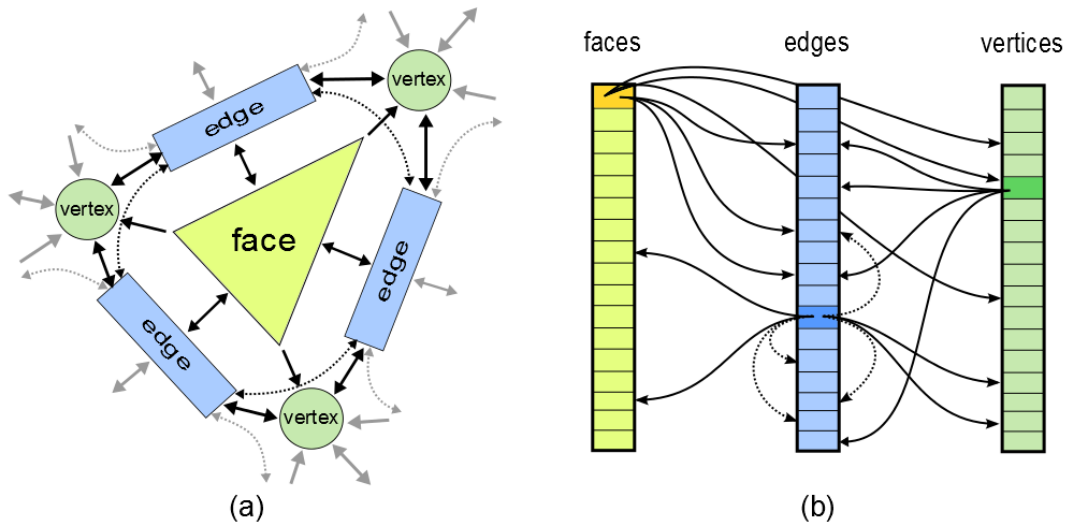


FIGURE 4.1: Winged edge data structure. (a) Entity dependencies. (b) Storage in lists with references.

4.2 Implementation details

All types of elements that are handled by the program are represented by the class hierarchy depicted in Figure 4.2. The common properties of all elements are accommodated in an abstract base class Element. The next level of the abstraction classifies the elements according to their spatial dimension. The particular class Beam stands for the one-dimensional element with linear approximation. The class inheriting from the Beam gains quadratic approximation by adding extra node in the middle of the line. The approximation type of other elements is distinguished by the type of their edges (a data structure describing the edge is similar to the one for 1D-elements).

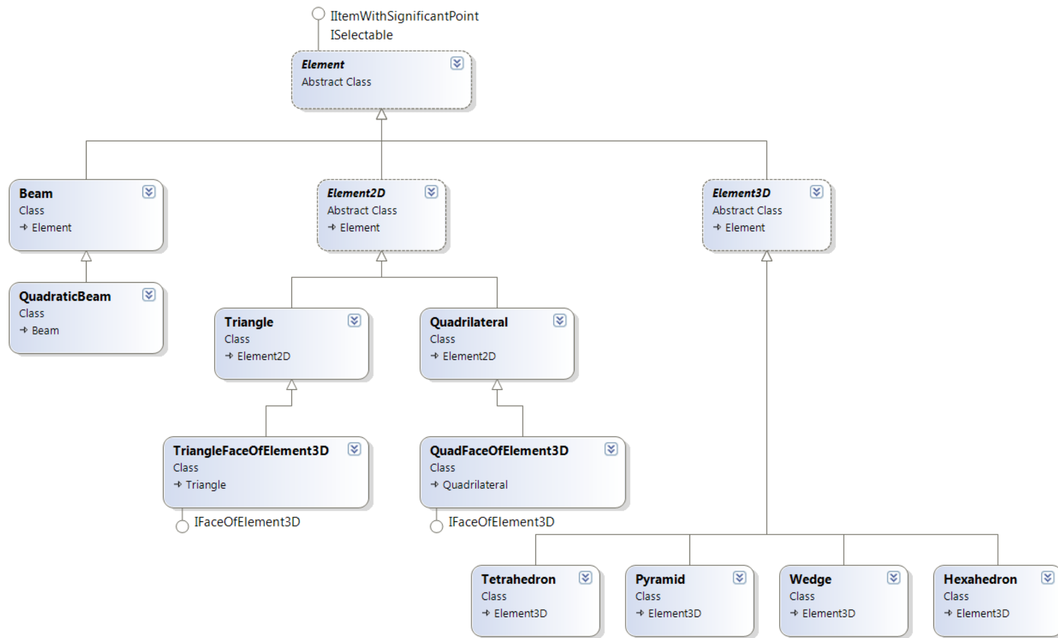


FIGURE 4.2: Class diagram with hierarchy of all supported element types.

The abstract base class Element2D is common for both the two-dimensional elements (triangle and quadrilateral) and faces of the three-dimensional elements (also triangles or quadrilaterals for all the widely used element types). The fact that 2D elements and faces of 3D elements can be handled in the same way allows us to implement a single generic algorithm for generation of the surface of the mesh so that the problem with hybrid meshes is hereby elegantly solved.

The 3D element face classes differ only by implementation of the interface `IFaceOfElement3D`.

```
interface IFaceOfElement3D
{
        Element3D ParentElement { get; }
}
```

The interface helps to include reference to the parent 3D element at each face. This link is important for selection of elements on the mesh surface, because surface is composed, among other things, of external faces of those 3D elements that lie on the domain boundary. The adapted winged edge pattern is applied for the surface representation. All classes participating in this data structure are summarized in the class diagram in Figure 4.3.

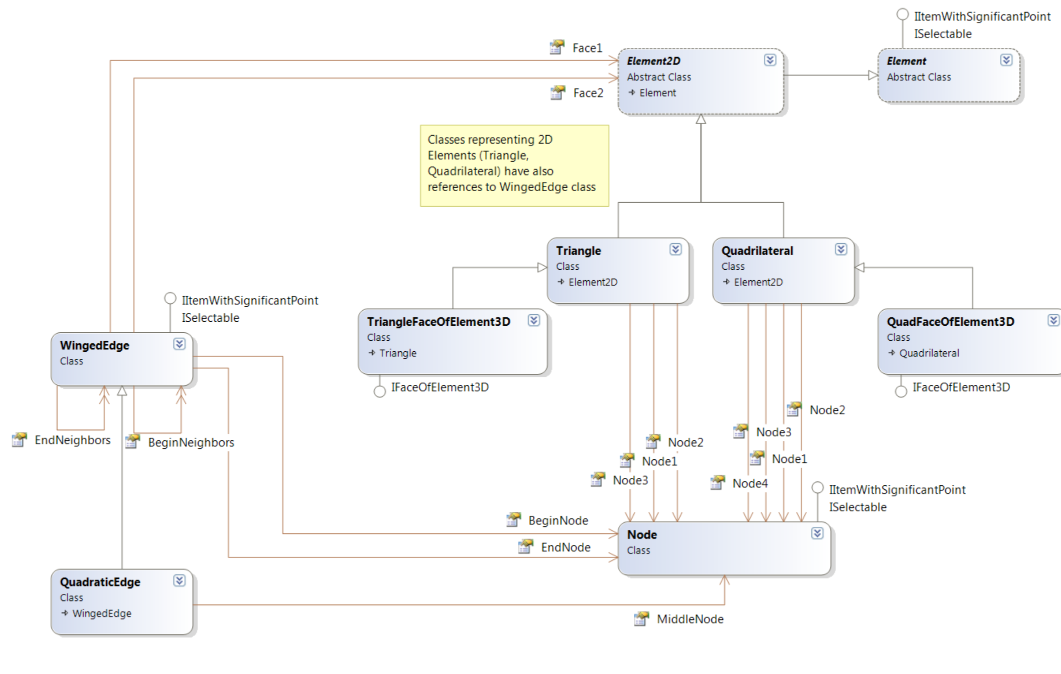


FIGURE 4.3: Class diagram with mesh surface representation.

Unlike traditional winged edge data structure used in computer graphics in which each edge has references only to two neighboring edges, in the proposed data structure each edge knows all its incident edges. The list of adjacent edges is tracked by every node and is shared between the node and all its neighboring edges. The ordering of edges in the list is arbitrary because in some cases there can be multiple candidates for its predecessors and successors. In this case no right ordering exists and traditional winged edge data structure used in computer graphics cannot be used. The surface representation is constructed on the fly using a single-pass algorithm. During the construction it cannot be known which edges are on the surface and what are its predecessors and successors. Therefore all adjacent edges for each node are kept in single unsorted list. To determine ordering of edges there would have to be second pass which would have significant negative impact on performance and therefore it is desired to avoid it. Moreover, for non-conform meshes there can be found no right ordering even after the surface construction.

Additionally, the proposed approach has better memory footprint due to sharing adjacency list between node and all its neighboring edges as oppose to traditional winged edge structure. Another advantage is better performance in most common use cases of the mesh editor. Every user-triggered operation with mesh starts with selection of node or face on the mesh surface. Having direct references between each node and all its incident edges enables us quickly traverse the whole mesh surface.

Another adjustment of the data representation that differs from the traditional winged edge structure used in computer graphics is calculating and storing angle between each two neighboring faces. This enables us to implement advanced features such as finding significant edges or selection of logically related elements (e.g., on the same flat surface) as described below.

Besides the references to the begin- and end- nodes, every edge also has references to the lists of neighboring edges for each of both nodes. The lists are shared between the adjacent edges to achieve lower memory consumption. The edge also contains references to the faces on the left and on the right. This kind of linking is

suitable for the majority of meshes. A problem can occur only in the non-conformal mesh processing when elements can share only a part of its surface. Visualization of this mesh can show up some artifacts caused by the fact that some internal faces do not have the adjacent counterparts and thus cannot be paired off. Another problem shows up when some elements have only one edge in common. In that case, the edge can be shared by more than two faces and the winged edge data structure cannot capture properly this situation. However, these are rare cases and do not render the program unusable.

4.2.1 Data structures overview

The modified winged edge data structure was used to describe the internal surface representation of a mesh. Instead of references to the left and the right adjacent edges, each node has reference to the list of all edges that begin or end at this node. The references to these lists are also contained in each winged edge object. This approach allows representing the meshes with some abnormalities or non-conformal meshes in where it is impossible to determine which edge is the left and which is the right. Other characteristics of the winged edge data structure remain the same. Each edge has references to the left and the right adjacent faces. Each face has references to its nodes, edges and the parent 3D element. The elements need to have references only to its nodes, because not every element is on the mesh boundary and so it does not need to have reference to surface objects. Every operation on the mesh, such as selection, begins with either a face or a node on the surface. Other entities are found by searching through the links in the winged edge data structure. Figure 4.4 shows data lists used in the mesh editor.

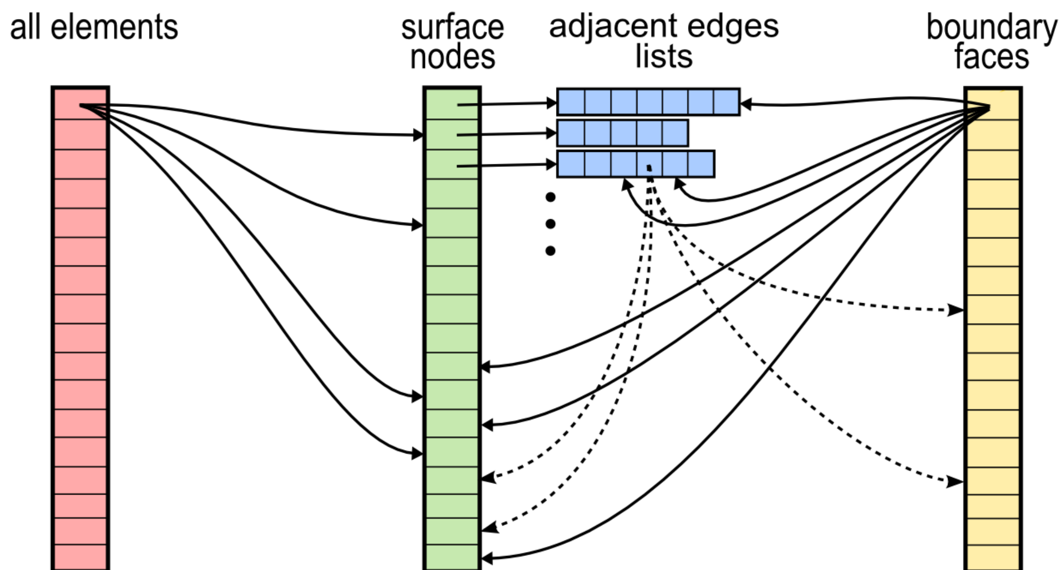


FIGURE 4.4: Diagram with data structure overview.

Memory requirements of each entity object are summarized in Figure 4.5. The overall memory consumption depends highly on the mesh topology and on the ratio of the number of surface elements to the total number of elements. This ratio decreases with the growing size of the mesh (or the element density) because the total number of elements increases with cube of the size of mesh, unlike the number of surface elements which increases with square of the mesh size.

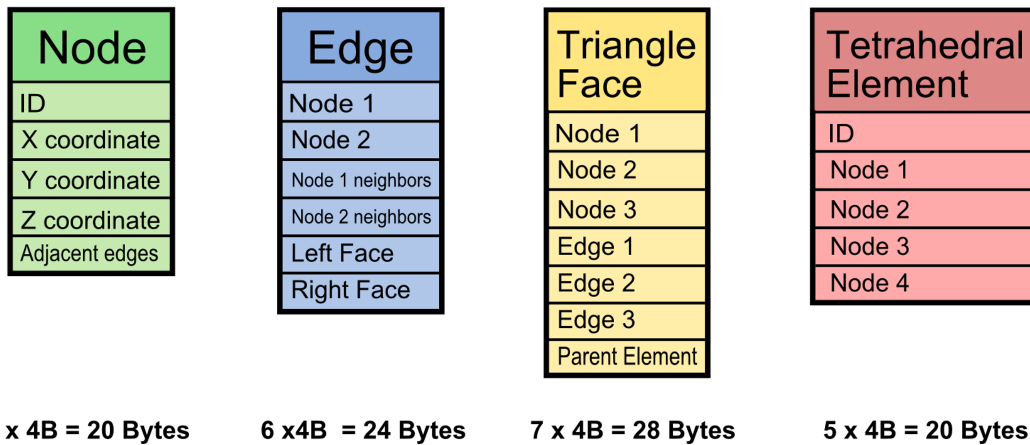


FIGURE 4.5: Memory consumption of the surface representation.

4.2.2 Surface representation construction

The process of creating the mesh surface representation is the key feature. It was designed as a one-pass algorithm which finds surface entities and creates the winged edge data structure on the fly during loading an input file. It should be noted that the input file does not contain any relationships between the mesh entities besides the coordinates of each node and a simple list of elements with links to relevant nodes.

The main problem is to find those faces of 3D elements that belong to the mesh surface. The key to resolving this issue is the consideration that each internal face has its twin in the adjacent element. While loading each element from the file, all its faces are generated. Then, the faces are included to a global hash table [66]. The key to this table is a special object consisting of the face node IDs. If the face under the same key already exists, it means that the face is an internal face that is not in the interest. Hence, the face is thrown away and the twin face is removed from the hash table. The procedure is repeated until all faces of all elements are processed. At the end (for conformal meshes) it is guaranteed that all remaining faces in the table are on the mesh boundary. For non-conformal meshes the algorithm can produce also some internal faces. Additional check for these situations is possible but it would break the simplicity and performance of the algorithm. Figure 4.6 shows a diagram with the progress of loading an input file and constructing the mesh internal representation.

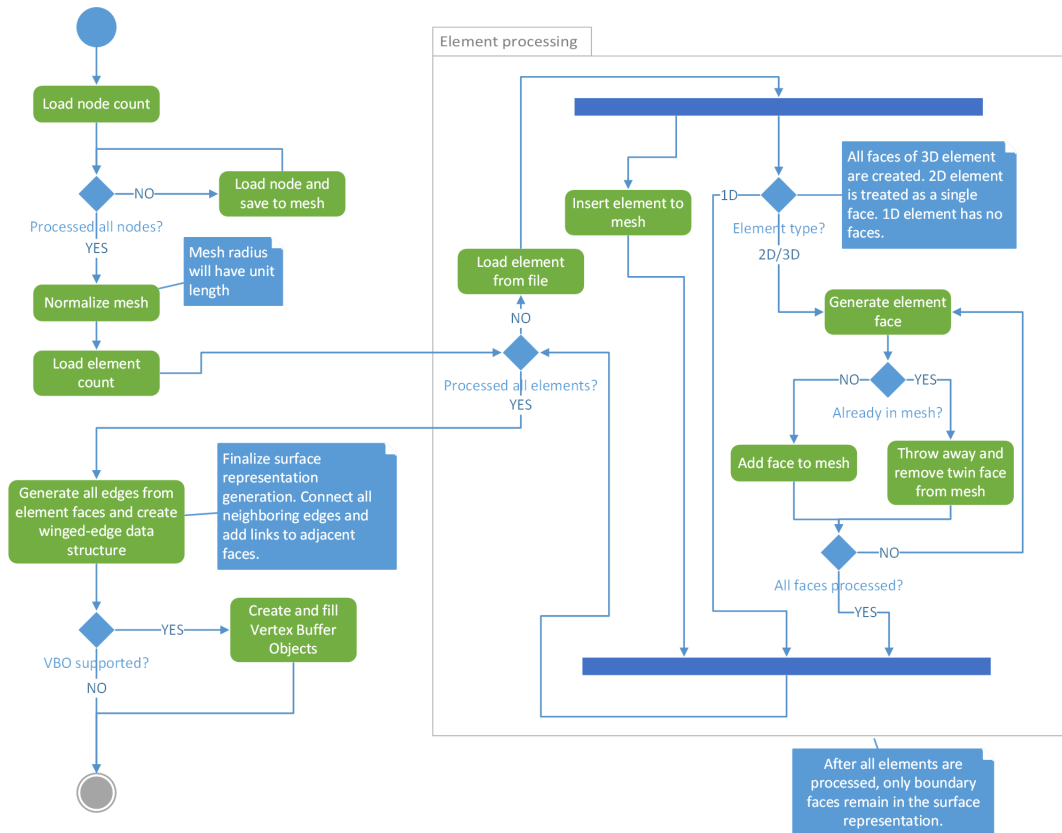


FIGURE 4.6: Activity diagram of loading and construction mesh surface representation.

In order to complete the surface representation, the edges must be generated from their faces. The analogous procedure that works with the hash table of edges is used here. Each edge has its twin in the adjacent face on the surface. But only one representative is needed. If a new edge is included into the table and under the same key the object is already contained, the new edge is thrown away. But compared to the previous case the edge in the table is not removed. The byproduct of this algorithm is finding the edges on the surface boundary. These edges have no twins as they have only one neighboring face. This however applies only to two-dimensional meshes, since the 3D meshes have a closed boundary.

Finally, when all surface entities are found, the Vertex Buffer Object (VBO) is created. It is an OpenGL feature that provides methods for uploading vertex data to the video device for rendering. VBO offers substantial performance gains over the immediate mode rendering because the data resides in the video device memory rather than the system memory and so it can be rendered directly by the video device. However, the buffer of rendered data must be updated each time the surface geometry changes. All up-to-date video cards support the VBO feature. If the card still does not support this feature, the application is smart enough to use the immediate-mode for rendering.

4.2.3 Looking inside the mesh

The editor allows to hide (or delete) some group of elements to enable the user to look inside the mesh. This makes sense especially for the 3D models when the user

wants to see and manipulate the internal entities. Let us suppose that the set of elements to be hidden is known. Firstly, the program disposes of the current mesh surface representation and after that it basically creates the entire surface again with the use of the above mentioned effective surface representation construction algorithm. However, in this case the algorithm does not take all the elements from the input file as an input parameter, it take only those that are **not** contained in the list of elements to be removed. This solution offers the possibility to reuse the existing code that is already optimized and universally applicable to all types of meshes.

Moreover, this approach simplifies the implementation of other useful features summarized in the following list (all the features use the same surface construction algorithm).

- **Hide selected elements** – remove a set of arbitrary elements selected by the mouse pointer.
- **Show or hide elements with specific property** – e.g., turn on or off layers containing elements with the same material id.
- **Cut through the mesh defined by plane** – hide all the elements that are located behind the plane specified by three points lying on the plane or by a point on the plane and a normal vector. In this case, only the elements that fall entirely behind the cut plane are displayed. Then the view is not perfect, but this approach is needed in the editor, where the properties need to be assigned to the individual entities, such as nodes and faces of elements. If the cut portion of the elements intersecting the cut plane would be displayed, editing of entities on the cut will not be possible. Also the implementation would be more complicated and the winged edge data structure could not be used.
- **Cut through the mesh defined by general algebraic equation** – previous operation (cut defined by plane) uses internal testing function that takes the point coordinates as an input parameter and returns a boolean value saying whether the point lies inside the area to be removed or not. The testing function is called for all currently visible elements. The testing function is in this case generalized to test the point against the algebraic equation specified by the user.

Each time the cut is performed the surface representation of the mesh is regenerated. Therefore turning on the cuts has no effect on the speed of manipulating views. All calculations are performed only during applying the cut through the mesh.

4.2.4 Finding visible nodes

For various operations with a mesh, it is useful to have some method that finds the set of nodes that are visible from the current view-point, i.e., nodes that are not hidden behind a part of the mesh. Apart from the rendering of nodes and labels on the mesh surface, information about visible nodes can be used for implementation of the method for selection of entities on the surface using the mouse pointer.

The principle of the method is simple. Firstly, the method performs the perspective projection of all nodes to the screen and their actual depth is determined. Then the whole model is rendered into depth-buffer to produce a depth-map of the mesh. After that, the value in the depth-buffer corresponding to each node is compared to the Z-component of the screen coordinates of this node. If the value in the depth-buffer is greater than the projected distance of the node, the node is hidden behind

some face. Otherwise, the node is added to the visible nodes set. The best precision in the depth-buffer is for the vertices that are in a small distance from the front clipping plane of the viewing frustum. The first step of the algorithm is therefore to move the near clipping plane from the observer to the mesh surface as close as possible. The pseudo-code of the described algorithm follows.

```
Set<Node> FindVisibleNodes(Rectangle area)
{
        GL.MatrixMode(MatrixMode.Projection);
        GL.LoadIdentity();
        // move the Near plane closer to model for better precision
        double newZ_NEAR_PARAM = ComputeMeshMinVisibleDistance();
        // set perspective projection with updated parameters
        Glu.Perspective(FOVY_PARAM, aspect_ratio, newZ_NEAR_PARAM, Z_FAR_PARAM);
        // -------------------------------------------------------------------
        // compute projections of all nodes to screen coordinates
        Dictionary<Node, Vector3> projections = new Dictionary<Node, Vector3>();
        Vector3 winPos;
        foreach (Node node in allSurfaceNodes)
        {
                Glu.Project(node.Position, modelview, projection, viewport, out winPos);
                if (area.Contains(winPos.X, winPos.Y)) // if the point is inside area,
                        projections[node] = winPos;  // save projection to dictionary
        }
        // -------------------------------------------------------------------
        // draw faces to depth buffer
        GL.Clear(ClearBufferMask.DepthBufferBit);
        GL.PolygonOffset(1f, 1f); // move little bit to enable testing
        GL.Enable(EnableCap.PolygonOffsetFill);
        DrawFaces();
        GL.Disable(EnableCap.PolygonOffsetFill);
        // -------------------------------------------------------------------
        Set<Node> result = new Set<Node>();
        foreach (KeyValuePair<Node, Vector3> pair in projections)
        {
                Node node = pair.Key; winPos = pair.Value; float pixelDepth;
                GL.ReadPixels(winPos.X, winPos.Y, 1, 1, Format.Depth, Type.Float, out pixelDepth);
                // !! key depth test; if passes, node is visible
                if (winPos.Z <= pixelDepth)
                        result.Add(node);
        }
        return result; // return list of visible nodes
}
```

4.2.5 Selection of entities

The mesh editor implements a tool for selection of entities in the mesh. The tool has two modes. The first mode allows user to select entities contained in the rectangular area specified by dragging the mouse pointer. Implementation of this tool uses the above mentioned method `FindVisibleNodes`. The set of nodes returned by the method is then transformed to the set of entities that the user wants to select by looking to the winged edge data structure that takes advantage of its rich interconnections. For example, if the user wants to select all the edges in the rectangular area, the algorithm finds all edges that have one of theirs node in the set of nodes returned by the method `FindVisibleNodes` that takes the selection rectangle as a parameter.

The second mode of the tool is selection of some geometrically related entities, e.g., all nodes on the top of the mesh. The input file that is used does not contain any information about neither the model from which the mesh was generated nor the curves and planes that form the surface of the model. However, the user usually requests to be able to select such groups of entities.

The core of the implementation of this functionality is the method `SelectSurface`. The method returns the set of element faces on the mesh surface that have prespecified maximum angle between them. The angle (passed as a function parameter) defines the degree of smoothness of the resulting surface.

The limit angle is determined by the count of the mouse button clicks on some face. The target face is also passed as an input parameter of the function. The limit angle increases with a growing number of clicks. The program operates with four levels of tolerance in the surface selection. The default values of the limit angles for

each level are carefully chosen to suit the largest number of the mesh. For atypical-formed meshes, the user can change these default parameters.

The following pseudo-code shows the implementation of the `SelectSurface` method. Figure 4.7 illustrates the result of the double- and triple-click on the mesh surface. Double-click selects all faces in the area that is delimited by the first limit angle. Default value of this parameter is one degree. Therefore double-click selects the elements or faces of elements which have a face on the same flat surface as the mouse cursor, whereas the triple-click selects the elements or faces on the smooth (possibly curved) surface as the mouse cursor. This surface is delimited by edges whose adjacent faces include angle lower than or equal to the second limit angle. Fourth click selects all the remaining neighboring elements or faces.

```
Set<Element2D> SelectSurface(Element2D startFace, float borderAngleLimit)
{
	Set<Element2D> selectionSet = new Set<Element2D>(); // faces to select
	Stack<Element2D> faces = new Stack<Element2D>(); // visited faces
	faces.Push(startFace); // push starting face
	selectionSet.Add(startFace);
	while (faces.Count > 0) // while the stack is not empty
	{
		Element2D face = faces.Pop();
		// add to the stack all neighboring faces
		// that form with the face an angle of value at most borderAngleLimit
		foreach (Element2D neighbor in face.GetNeighbors(borderAngleLimit))
		{
			if (!selectionSet.Contains(neighbor))
			{
				faces.Push(neighbor);
				selectionSet.Add(neighbor);
			}
		}
	}
	return selectionSet;
}
```

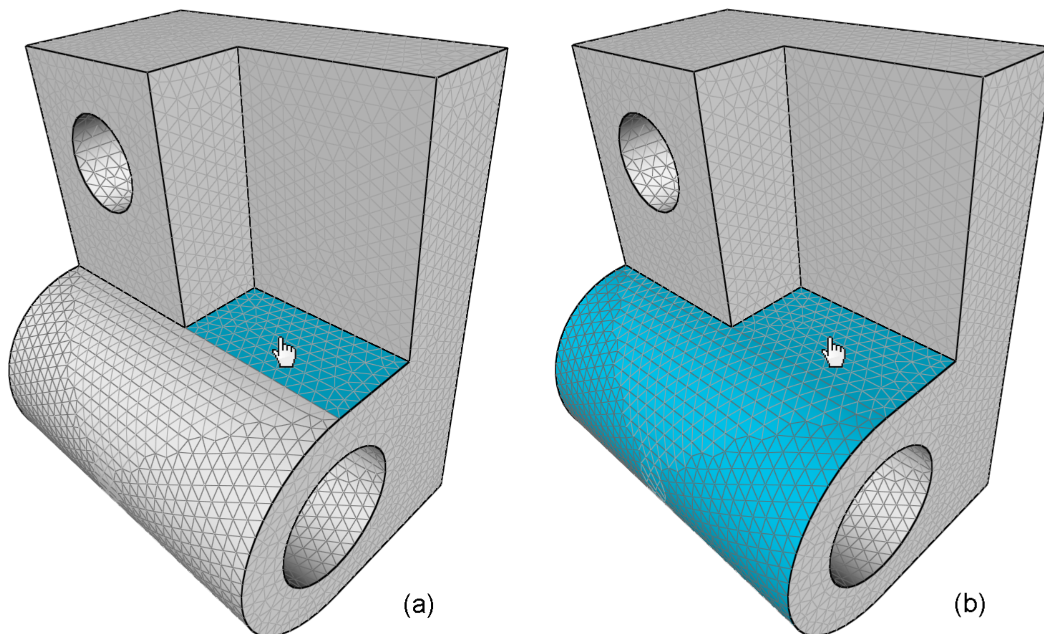


FIGURE 4.7: Selection of element faces on the mesh surface. (a) Result after mouse double-click. (b) Result after mouse triple-click.

4.3 Results

The program was benchmarked against other commonly used editors that have similar capabilities. The **GiD** post-processor [67], **ParaView** [34], and **VisIt** [68] were

TABLE 4.1: Measured results: Initial loading time in seconds (CPU time).

mesh (size)	MeshEditor	GiD	ParaView	VisIt
Beam (651873 elements)	36	28	16	24
Beam (47680 elements)	4	5	3	2
Beam (3222 elements)	1	4	2	1
Sphere (348014 elements)	4	3	7	2
Robo (244188 elements)	7	7	7	7

TABLE 4.2: Measured results: Memory consumption in megabytes (Private working set) [MB]

mesh (size)	MeshEditor	GiD	ParaView	VisIt
Beam (651873 elements)	824.616	620.956	339.436	526.144
Beam (47680 elements)	104.124	92.772	128.888	123.244
Beam (3222 elements)	42.800	52.480	114.380	89.116
Sphere (348014 elements)	108.324	106.428	133.652	128.808
Robo (244188 elements)	291.836	200.484	543.316	355.268

chosen. The initial loading time (see Table 4.1) and memory consumption were measured (see Table 4.2). The memory requirements of the mesh editor grow slightly faster than requirements of other tools for meshes with high number of finite elements. It is caused by auxiliary data structures assembled for future fast manipulation with meshes. The mesh Beam with 651.873 elements represents a limit task which is solvable on a single-processor computer. For smaller problems, memory requirements of the mesh editor are equal or even smaller in comparison with GiD, ParaView, or VisIt. Figure 4.8 contains visualizations of the meshes that were used in the benchmark.

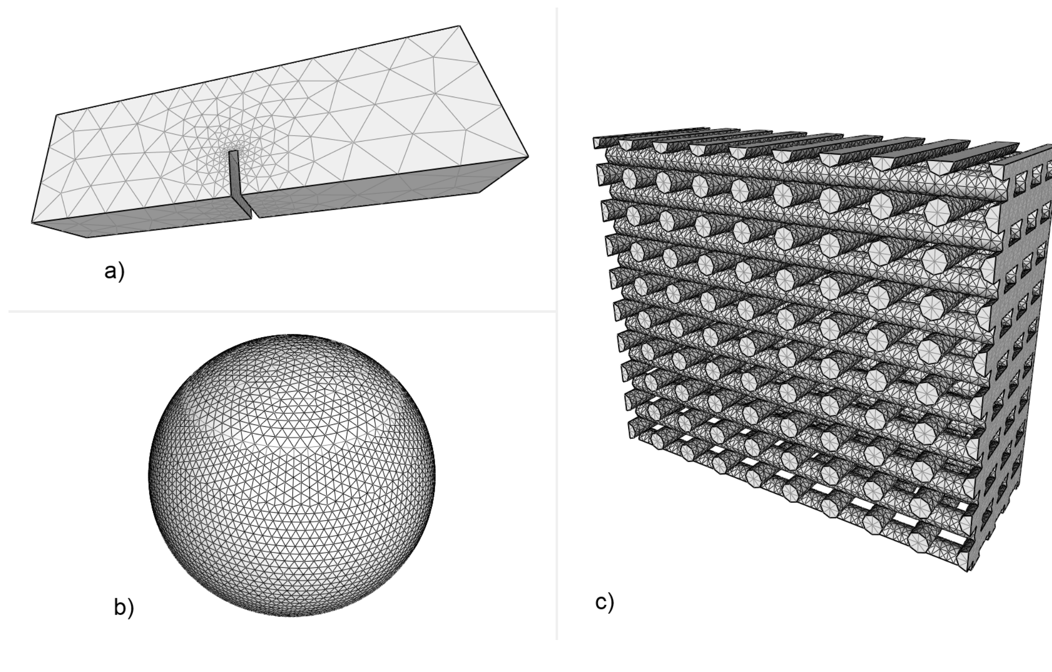


FIGURE 4.8: Visualizations of meshes used in benchmark. Three different densities of the Beam mesh were used to demonstrate the effects of mesh size. Meshes Sphere and Robo show effect of surface area/volume ratio. a) Beam. b) Sphere. c) Robo.

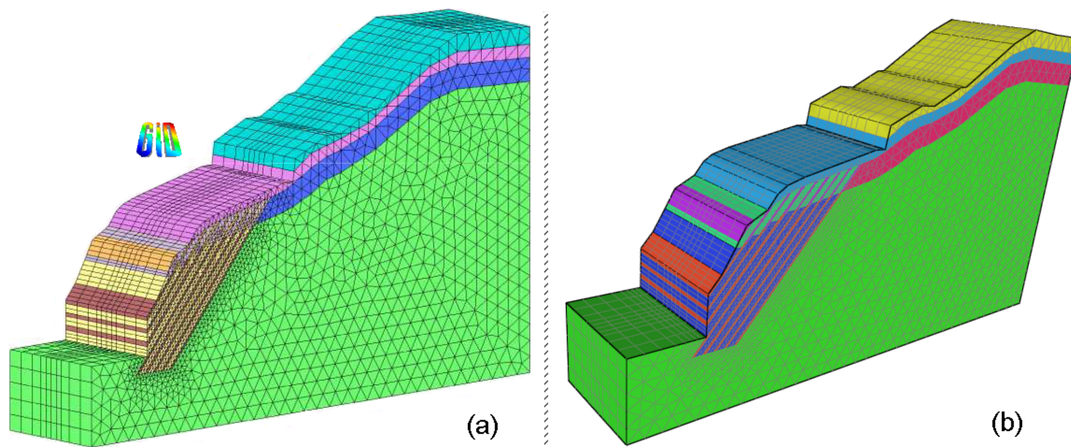


FIGURE 4.9: Finding and visualization of significant edges. a) GiD post-processor. b) The MeshEditor.

For benchmarking a PC with Intel Core i7-2600 processor, 16 GB memory, and NVidia GeForce GT 545 graphics card was used. The results reflect the fact that the more complex data structure is constructed to represent the mesh. The manipulation with the model is comfortable even for a very large data input. The program takes benefit of creating the sophisticated internal data structure that, on the contrary, causes longer initial loading time. However, the presented smart internal model representation allows selection of logically related groups of entities on the mesh surface without knowing the geometrical and topological connections from the modeller. The program is able to find automatically the significant edges where the geometry changes (shown in Figure 4.9) and therefore simplify the process of

selecting the entities. Moreover, individual elements can be easily removed so it is allowed to dig into the mesh and view the internal elements. The other editors, which were tested against the proposed editor, do not support those features. Besides that, the capability of unlimited view-point movement allows to fly through holes in the mesh and investigate the mesh from the inside.

Chapter 5

Approximation of FEA results by polynomial functions

This chapter contains the description of the proposed method for approximation of results from the finite element method, which are discrete values, by polynomial functions. It is an alternative to the data representation and the compression method introduced in Chapter 3 and Chapter 6, respectively. It is worth noting that this method was excluded from the final implementation of the desktop post-processor because of the hard-to-control error of approximation and unfavorable performance characteristics. The content of this chapter is also published in [69] and [70].

5.1 Idea

The multigrid method [71, 72, 73, 74] was the inspiration for this work. Multigrid method allows to solve partial differential equations using the hierarchy of domain discretizations. The main idea of multigrid method is to make the convergence of iterative method faster due to global corrections of error that is made from time to time on the coarser mesh. There are many variations of multigrid method. However, all of them need existence of mesh hierarchy that represents domain discretizations of different mesh sizes.

Basic steps of multigrid method are:

- **Smoothing** — The main goal of the smoothing phase is the high-frequency error reduction. It can be done e.g., by few iterations of the Gauss-Seidel method.
- **Restriction** – Restriction of the residual from the finer to the coarser mesh.
- Solution of the coarse problem.
- **Prolongation** – Interpolation and projection of the correction computed on the coarser mesh to the finer mesh.

The main problem with a mesh hierarchy is that often none is available. Only the finest mesh exists. The coarser meshes must be either generated directly by a mesh generator [4, 5] in the pre-processing phase or it must be created from the finer mesh. But generating coarser mesh from the finer one is very problematic or even impossible, because corresponding nodes between different levels should be preserved to be sure that multigrid method will work correctly without special modifications.

Therefore, it was decided to do visualization of the results from the finite element analysis on the fine mesh that is used for solution of FEM. Different methods of simplification and compression of the resulting data in space and time were developed and data were projected back to fine mesh. Results of the projection and comparison of methods are presented below in this chapter.

5.2 Implementation

Even if the mesh hierarchy is generated, one of the obstacles for using the same multigrid techniques as are often used in the finite element analysis is that only one mesh hierarchy is available. That is sufficient for finite element solver, because this hierarchy is used to solve only one set of equations. However, in the post-processor it is necessary to display various kinds of data, such as temperature, displacements, stress, strain, etc. These quantities are scalars, vectors or tensors of second order. Components of vectors and tensors could be considered in post-processing as a scalar and therefore scalars will be dealt in the following text. Every scalar is represented in the finite element analysis by a set of discrete values computed in nodes or Gauss points, but in the strong formulation of a problem, it is a function. For graphical purposes, it is possible and often suitable to replace the set of discrete values by a continuous function. In the following text, the set of discrete values describing a scalar will be denoted as the discrete function or original function, but approximation of the discrete values for graphical purposes by continuous function will be called approximation function (shape functions used in FEM are not used here).

Approximation functions should be as simple as possible to be representable by a small set of parameters. Therefore, the domain of approximation function should respect the character of the discrete function. It can't be the whole mesh, because one part of mesh could contain data replaceable by a simple linear function and other part could have much wilder character. It is therefore necessary to find alternative division of problem domain that will respect the shape of function in space and time better than the mesh hierarchy used in the multigrid method. Moreover, each quantity component must have its separately generated mesh hierarchy.

5.2.1 Octree generation

Domain space has to be divided into subdomains of the size which allows to replace discrete function with continuous, simpler, e.g., linear function that is easy to describe by fewer parameters. The goal is to automatically recognize areas in mesh, where the nature of function is smooth (the function is continuous together with the first derivatives and very coarse mesh can be used) and areas in which function rapidly changes its character (the first derivatives are large or the function is even discontinuous). These areas of interest become object of further subdivision, because for visualization purposes they need finer underlying mesh. For recursive division of 3D space the octree data structure is suitable (see Figure 5.1). The other spatial dividing data structures used in computer graphics were investigated, but octree seemed to be the best choice due to its hierarchical form and low average depth.

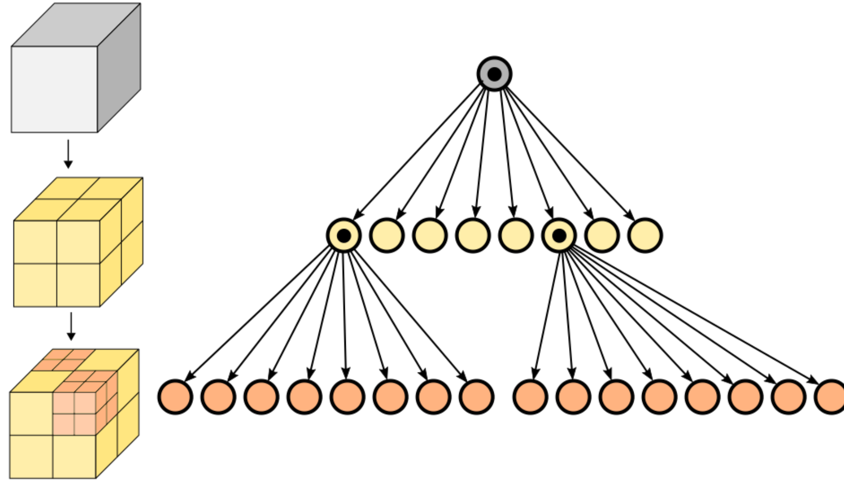


FIGURE 5.1: Octree visualization. Left: Recursive subdivision of a cube into octants. Right: The corresponding octree.

Basic overview of the decomposition and approximation procedure is in Listing 5.1. At first whole mesh is inserted into one big cube – octree root node. Then a condition that tells whether to divide current domain into eight subdomains is needed. Three conditions were designed. All have to be satisfied to proceed with decomposition.

1. First condition specifies minimum number of finite element nodes to be represented by single octree cell. There are two reasons. Firstly, some specification of minimum number is necessary to compute approximation function, e.g., least square trilinear algorithm needs at least 8 values. Secondly, if the number is too small and the approximation function does not fit ideally, subdivision of the octree will be too subtle and memory consumption will easily exceed the case with no approximation applied at all.

2. Second condition is based on maximal allowed relative error of chosen approximation function. Algorithm that replaces discrete data points with continuous approximation function also calculates relative error of the method. This number is then compared with some preset fixed value. According to the experiment results the most appropriate value of 1% was chosen.

3. Third condition describes maximum depth of octal tree. Each level of the tree exponentially increases memory consumption of data values stored in octree. Maximum depth is therefore artificially set to some acceptable value, e.g., 9. However, this depth should not be reached in common cases, it is ensured by condition 1.

If all conditions are met, domain represented by the current octree node is divided into eight subdomains. Differences between original value of discrete function and value of approximation function in the same point are transferred to corresponding sub-nodes based on their location and whole process is recursively repeated in all octree sub-nodes. The difference D_i of the i-th data value is defined by

$$D_i = V_i - \bar{V}_i, \quad (5.1)$$

where V_i is the original function value in the i-th node and $\bar{V}_i$ is the value of approximation function in the same location as the i-th node.

Similar procedure – passing residuals between mesh levels – is applied in the multigrid method. Due to this approach the top levels of octree filter out main character of function (lower frequencies), bottom levels and leaves of the octree catch higher frequencies of function values.

5.2.2 Approximation in space

Discrete values within an octree cell are replaced by a continuous function which is as simple as possible and can be represented in memory by a few parameters. It is therefore necessary to find suitable type of function and in the case of polynomial functions also the order of the function. Compression algorithm has to be very fast. Compromise between low error and memory consumption must be found. For the sake of simplicity at the beginning of the work the relations between neighboring octree nodes were neglected. Nodal values in each octree cell are approximated separately.

The compression procedure requires the surface representation of the mesh to be already created. The element connectivity and nodal coordinates has to be present in memory and accessible in constant $O(1)$ time. Efficient methods to create this surface representation are described in Chapter 4 and also in [62].

The procedure is reading the FEM results divided to data sets from the external file and then processing the data sets one by one. By a data set is meant primarily the array of floating-point numbers corresponding to one component of one quantity, e.g., x component of displacement vector, temperature (which is scalar) or one component of stress tensor. Each floating-point number is the value of the quantity in single time step corresponding to one node or Gauss point. Whole array is loaded from file into computer memory, its values are distributed into growing octree and the approximation is calculated and saved in corresponding octree cells. After that the original data are deleted and algorithm continues with the next data component.

Compression has to be made on-the-fly during loading of data from the file to the memory between each data component. Starting compression after all results has been loaded to the memory would not make sense, because the main purpose of the compression step is to save overall memory consumption of the post-processor. Therefore, the format of data should ideally be designed in the way that each data component is separated in single data file or at least in one isolated data block in the file not mixed with other data. No particular order of data components is required. However, the data formats used by common finite element software packages are usually designed in the way that the components of each quantity are grouped together. E.g., the data in GiD postprocess file format (**.res**), which is described in detail in [38], are divided into blocks according to physical quantity and time step. Each block is the list of value tuples introduced by a node number (or element number in case of values in Gauss points), e.g., x, y and z component triplet for displacement vector in node 42 in time step 3.0. To avoid multiple passes through the data file the value tuples are cached and processed right after reading the whole data block.

LISTING 5.1: Core of aproximation procedure.

```
function OctreeInternalNode.InsertDataValues(dataValues, dataComponentId, depth, approximationMethod)
{
  // find approximation function and save it in current octree node
  approximation = ComputeApproximation(dataValues, dataComponentId, approximationMethod)
  DataCatalog[dataComponentId] = approximation
  // if condition #2 is met, propagate values to child nodes
```

```
  if (approximation.MaxRelativeError > MIN_RELATIVE_ERROR_TO_EXPAND)
  {
    // split residual values to octants based on data point positions
    foreach (dataValue in dataValues)
    {
      // calculate residual and distribute to octants according to position
      position = mesh.Nodes[dataValue.NodeId].Position
      residual = dataValue.Value - approximation.GetValueAt(position)
      octantIndex = getIndexOfOctantOnPosition(position)
      residuals[octantIndex].Add(new DataValue(residual, dataValue.NodeId))
    }
    foreach (octantIndex in range 1..8)
    {
      // if condition #1 is met, algorithm is recursively called on child octree node
      if (residuals[octantIndex].Count > max(MIN_LEAF_DATA_POINTS_COUNT,
           approximationMethod.MinNumberOfDataPoints))
      {
        // recursive call to InsertDataValues, if children[octantIndex] is LeafNode, than recursion is stopped
        children[octantIndex].InsertDataValues(residuals[octantIndex], dataComponentId, depth + 1,
             approximationMethod)
        // if child node is leaf, find out if expansion is needed
        if (children[octantIndex] is LeafNode)
        {
          // if condition #3 is met, algorithm can continue with octree node expansion
          if (depth < MAX_OCTREE_DEPTH - 1)
          {
            // if condition #2 is met, expand leaf node
            if (children[octantIndex].DataCatalog[dataComponentId].MaxRelativeError >
                 MIN_RELATIVE_ERROR_TO_EXPAND)
            {
              children[octantIndex] = new OctreeInternalNode(children[octantIndex].LowerBounds,
                   children[octantIndex].UpperBounds)
              children[octantIndex].InsertDataValues(residuals[octantIndex], dataComponentId, depth + 1,
                   approximationMethod)
            }
          }
        }
      }
    }
  }
}

function OctreeLeafNode.InsertDataValues(dataValues, dataComponentId, depth, approximationMethod)
{
  // find approximation function and save it in current octree node
  DataCatalog[dataComponentId] = ComputeApproximation(dataValues, dataComponentId, approximationMethod)
}

function ComputeApproximation(dataValues, dataComponentId, approximationMethod)
{
  switch (approximationMethod)
  {
    case TrilinearInterpolation:
      approximation = DoLeastSquaresTrilinearInterpolation(dataValues)
    ...
  }
  // compute absolute error
  maxError = 0;
  foreach (dataValue in dataValues)
  {
    position = mesh.Nodes[dataValue.NodeId].Position
    error = dataValue.Value - approximation.GetValueAt(position)
    maxError = Max(maxError, Math.Abs(error));
  }
  approximation.MaxError = maxError;
  approximation.MaxRelativeError = maxError / GlobalDataRange[dataComponentId]
  return approximation
}
```

After loading a data block, compression is started. Listing 5.1 contains pseudocode of recursive procedure that is the core of the compression algorithm. Figure 5.2 contains overview of this algorithm in form of UML Activity diagram. The input is an array of nodal values in one component of one field defined on the mesh in single time step. In case of data stored in Gauss points the values has to be at first extrapolated to nodes using natural coordinates supplied in the data file. Nodal data are then passed as an input parameter `dataValues` to the function `InsertDataValues` that is called upon the octree root node which represents one big cube that surrounds the whole mesh. Each `dataValue` object is a structure consisting of floating-point number Value and integer `NodeId` that represents a key to the table of nodes in global mesh object. `InsertDataValues` is pure virtual function declared in abstract base class `OctreeNode` and its implementation differs in derived classes. Its implementation in `OctreeLeafNode` is straightforward and so follows description of its implementation in class `OctreeInternalNode`. Simplified class diagram of the key types involved in Listing 5.1 is shown in Figure 5.3.

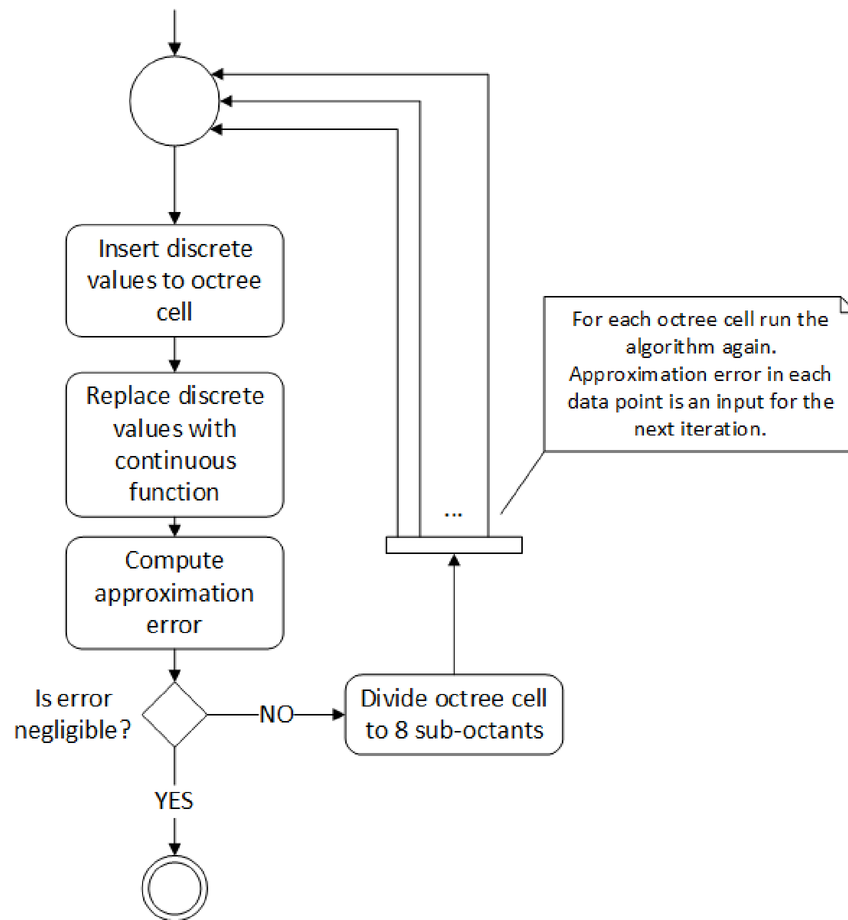


FIGURE 5.2: Octree generation algorithm in form of activity diagram.

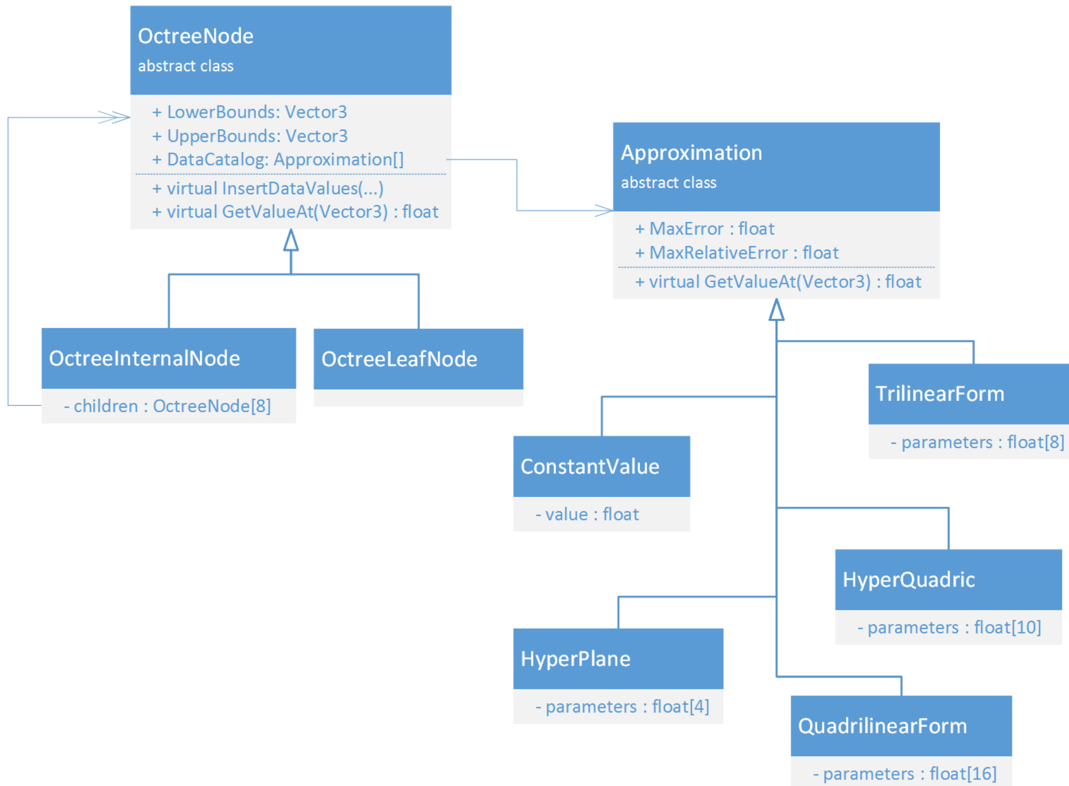


FIGURE 5.3: Class diagram of types involved in octree generation algorithm.

At first the algorithm computes parameters of continuous approximation function of discrete data values provided (variable `approximation`). In case of trilinear interpolation it uses least square method to calculate 8 parameters of a polynomial regression model and stores them in `DataCatalog` table which is a property of each octree node. Then the maximal relative approximation error is computed.

Approximation function for data values in the octree cell is assessed with the help of several metrics. First, absolute approximation error, which is the difference between the original nodal value and the value of approximation function, is evaluated in all nodes. Then, relative approximation error is obtained by dividing the absolute approximation error by global range of values (difference between global maximum and global minimum of original nodal values). The maximum relative approximation error in an octree cell is one of three parameters (see condition 2 above) that are used by the algorithm to decide whether to proceed with further subdivision of the octree cell. These three parameters control the overall quality of approximation, memory consumption and performance and need to be fine-tuned during testing on real-world data.

If the maximal approximation error is too high, the algorithm continues, calculates approximation error in each node and stores it in table residuals in relevant octant according to its position in current octree cell. `residuals` is a table of data value arrays and it is a local variable that will be disposed after each call to function `InsertDataValues` finishes. Algorithm then iterates over all child octants and checks for number of residuals assigned to them. If condition 1 stated above holds, algorithm is recursively called upon each child octree cell. If the octree cell is a leaf node, recursion is stopped and algorithm determines whether it should split current leaf octree cell into 8 sub-segments by checking conditions 2 and 3. In other words,

if the current octree branch is not deep enough and maximal residual belonging to current child octree cell is higher than designated epsilon value, then this leaf cell is replaced with internal cell and function `InsertDataValues` is called upon this new `OctreeInternalNode` object. Residuals located in current octant are passed as an input to this function.

This continues until approximation of current data component is good enough in all octree cells. When the algorithm finishes, original discrete data can be deleted, because the created octree structure with approximation functions in its nodes is all that is needed to reconstruct the original data. Then the algorithm can proceed with reading and processing next data set. Note that these operations are to a considerable extent independent and processing of data sets can be parallelized. However, if the same octree data structure is reused for multiple data sets, then the access to `DataCatalog` and octree node expansion has to be synchronized using standard locking mechanisms.

Also note that passing residuals of approximation instead of original data between octree levels is important, because it allows to describe the main character of function (lower frequencies) on top levels and details (high-frequency changes) on bottom levels of the octree. This design is inspired by the Multigrid method basic principles.

Approximation functions

Various approximation functions were investigated and tested. Besides polynomial functions also Discrete cosine transform [75] and Wavelet transform [11] were considered. Since the data compression algorithm has to be very fast, simple polynomial approximation functions were preferred. They are summarized below:

- **Mean value** – A single value (average value) replaces the set of discrete values. Arithmetic mean was used during testing because it is fast to compute unlike the median. Also, it takes into account whole spectrum of values in contrast of the mode value that is the value that occurs most often in the collection. It is suitable in statistics where the measurement errors have to be excluded. However, in the case of the results from FEM the user wants to see extremes in data and these outlying results should be rather highlighted instead of truncated. That is the reason why is neither the arithmetic mean nor other statistically estimated mean value suitable for this purpose. Mean value approximation diagram is depicted in Figure 5.4.

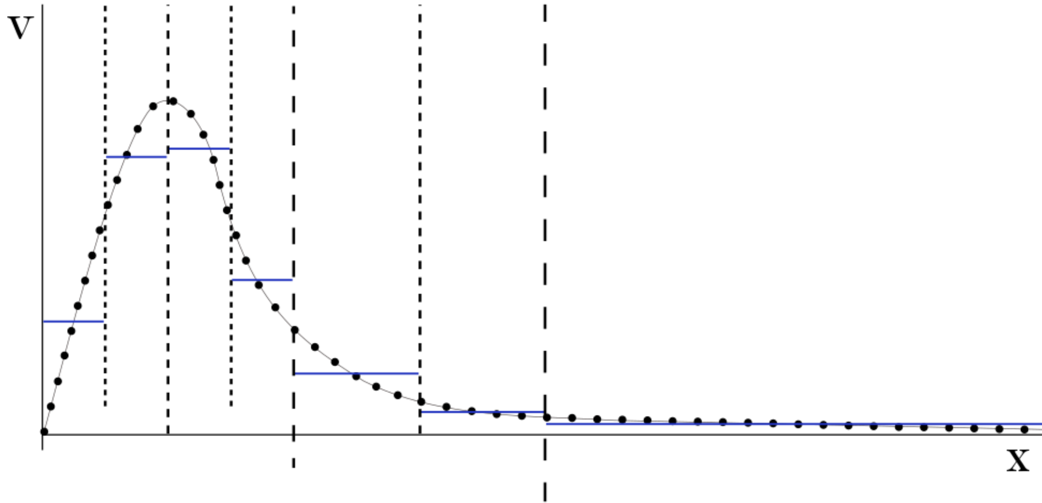


FIGURE 5.4: Mean value approximation diagram. Dashed lines denote octree division process. Shorter the line the deeper subdivision represents. Mean value is not suitable because of high errors and discontinuities between approximation cells. v stands for function value and x for spatial dimension.

- **Regression** – Finds a polynomial that models relationship between a scalar dependent variable and one or more explanatory (independent) variables. In three-dimensional problem there are three independent variables. Polynomial regression models are often fitted using the least squares method. The implementation used in this work is based on solving linear system of equations using LU decomposition. Key thing is that the size of the system does not depend on the number of data values, but on the number of approximation function parameters. Therefore, the algorithm has linear computational complexity and scales well. Several polynomials were tested. In the functions below x, y, z are spatial coordinates, c_i are parameters which determine the shape of the polynomial and v is the function value.

  - **Linear** – Hyperplane, only four parameters per octree cell. Value v in the point with coordinates x, y, z is computed using linear interpolation function in the form

    $$v = c_1x + c_2y + c_3z + c_4. \tag{5.2}$$

    Figure 5.5 contains example of creation of octree node hierarchy driven by this function.
  - **Quadratic** – Parametric shape models known as Hyperquadrics. They have too many parameters per cell (10) and are not suitable to capture continuity between octree cells. Value v in the point with coordinates x, y, z is computed using quadratic interpolation function in the form

    $$v = c_1x^2 + c_2y^2 + c_3z^2 + c_4xy + c_5xz + c_6yz + c_7x + c_8y + c_9z + c_{10}. \tag{5.3}$$

  - **Trilinear** – 8 parameters, the best compromise, consistent with neighboring octree cells, almost “seamless” transitions between octree cells. Also

used in FEM. Value v in the point with coordinate x, y, z is computed using trilinear interpolation function in the form

$$v = c_1xyz + c_2xy + c_3xz + c_4yz + c_5x + c_6y + c_7z + c_8. \tag{5.4}$$

The least squares method is applied to find parameters $c_1, ..., c_8$. The problem is solved by minimizing the sum of squared residuals G of the linear regression model

$$G = \sum_{i=1}^{N} (v_i - (c_1x_iy_iz_i + c_2x_iy_i + c_3x_iz_i + c_4y_iz_i + c_5x_i + c_6y_i + c_7z_i + c_8))^2, \tag{5.5}$$

where N is number of values which are interpolated. When the parameters of interpolation are known, value in any point of the approximated volume can be found simply by providing x, y, and z coordinates of the point in the equation.

– **Tri-quadratic** – Too many describing parameters with no significant benefit over trilinear form.
– **Quadrilinear form** – Generalized trilinear form, extended by temporal dimension, only theoretical option, not implemented.

Figure 5.6 depicts computation of nodal value via traversing octree from the root to the leaves and simultaneously summing up approximation errors. Root node of the octree contains all data components in all time steps where data value of an arbitrary element node can be computed. If an approximation is not sufficiently accurate in current octree node, correction can be made with the help of data stored in lower levels of the octree and summing up corrections together with initial value to compute final value for the node.

Results

The benchmark is designed to compare maximal relative approximation error, average error and compression ratio when using different approximation methods. Two test data sets were chosen (Figure 5.7 and Figure 5.8), both contain displacement vector values with three components (u, v and w) and about 30 time steps. Maximal relative approximation error is the highest relative error of an approximation method in single element node across all data components and time steps. Average error is a weighted sum of approximation errors in all nodes and data components divided by the number of these approximations. Compression ratio is memory consumption of the proposed data representation divided by memory consumption of original post-processor that does not use any data approximation techniques. Results are summarized in Table 5.1 and Table 5.2.

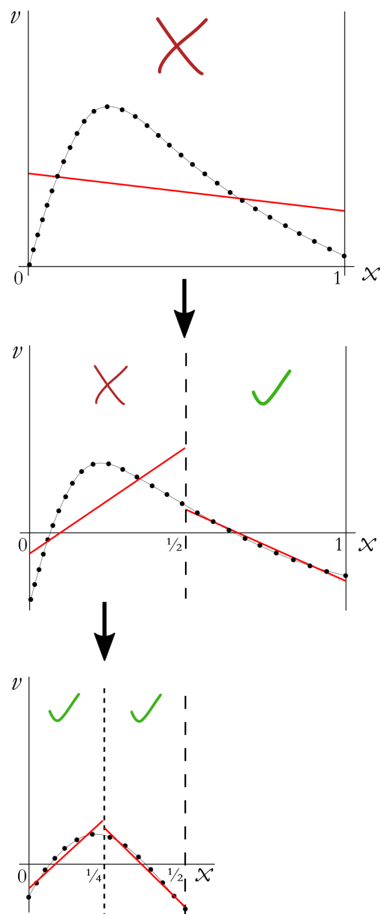


FIGURE 5.5: Octree creation example based on linear approximation error.

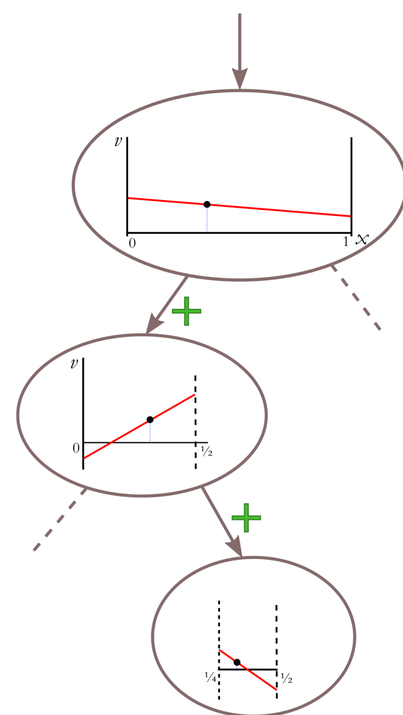


FIGURE 5.6: Nodal value computation diagram.

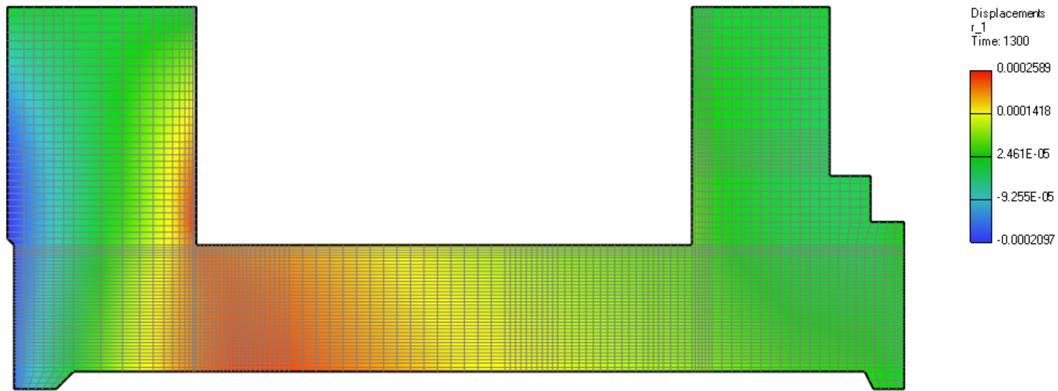


FIGURE 5.7: Reactor vessel 2D. Displacements visualization.

TABLE 5.1: Reactor vessel 2D. Spatial octree approximation results.

	Max error [%]	**Average error [%]**	**Compression ratio [%]**
Mean value	75.28	0.6268	55.3
Linear regression	44.23	0.1842	17.7
Trilinear regression	81.27	0.1751	16.9

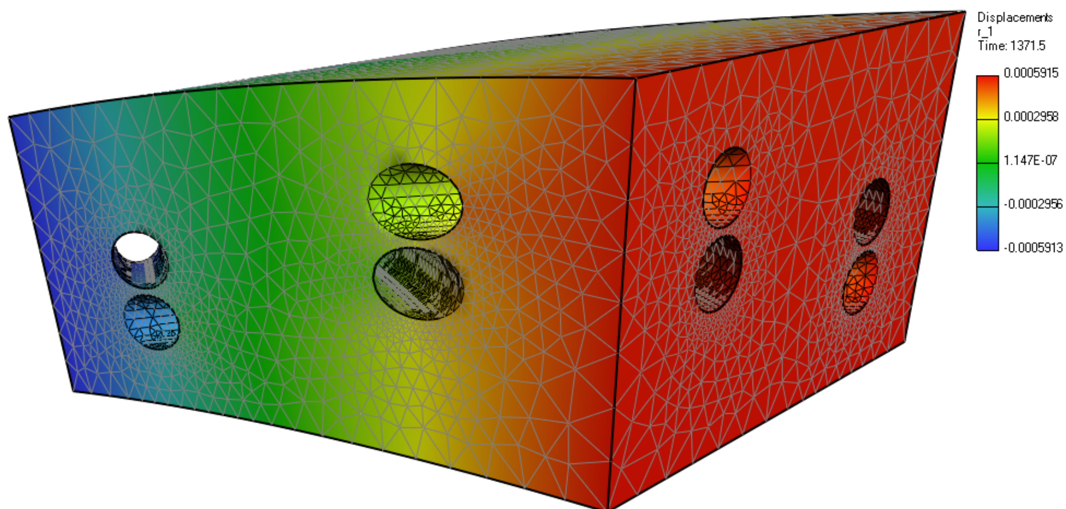


FIGURE 5.8: Segment of reactor containment. Displacements visualization.

TABLE 5.2: Segment of reactor containment. Spatial octree approximation results.

	Max error [%]	**Average error [%]**	**Compression ratio [%]**
Mean value	38.62	1.667	63.1
Linear regression	32.9	0.305	59.4
Trilinear regression	89.46	0.2237	56

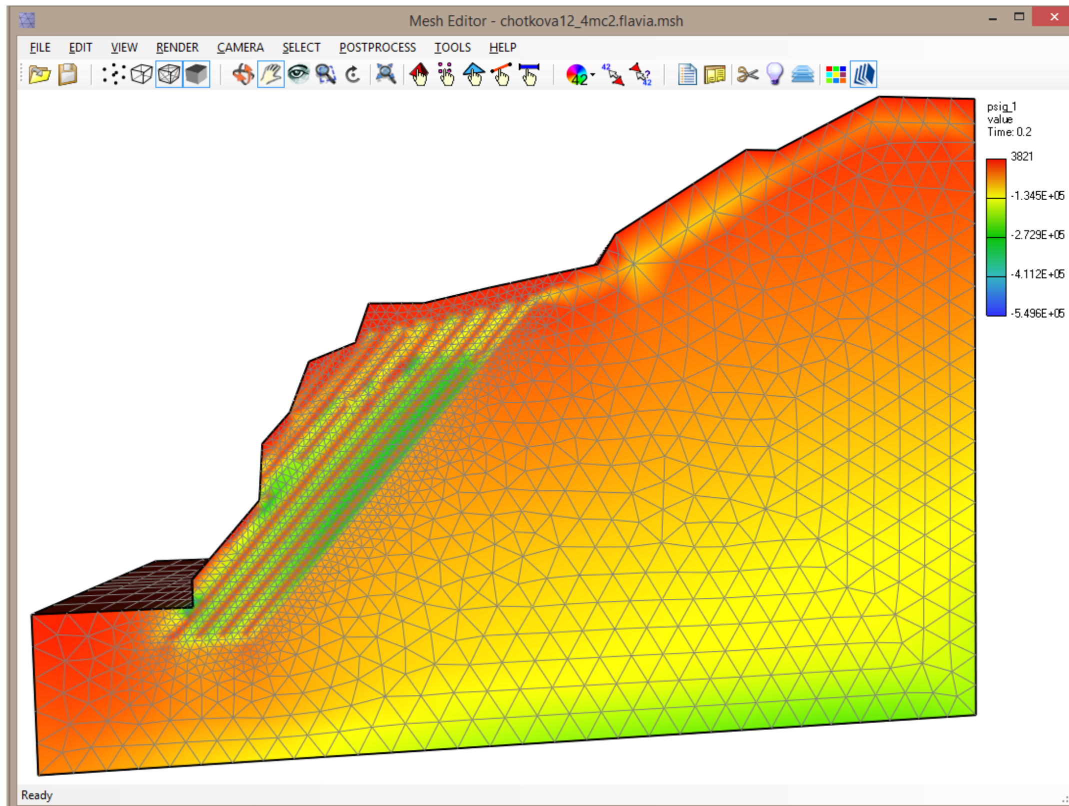


FIGURE 5.9: Geological layers simulation results. Exact data values, no approximation applied.

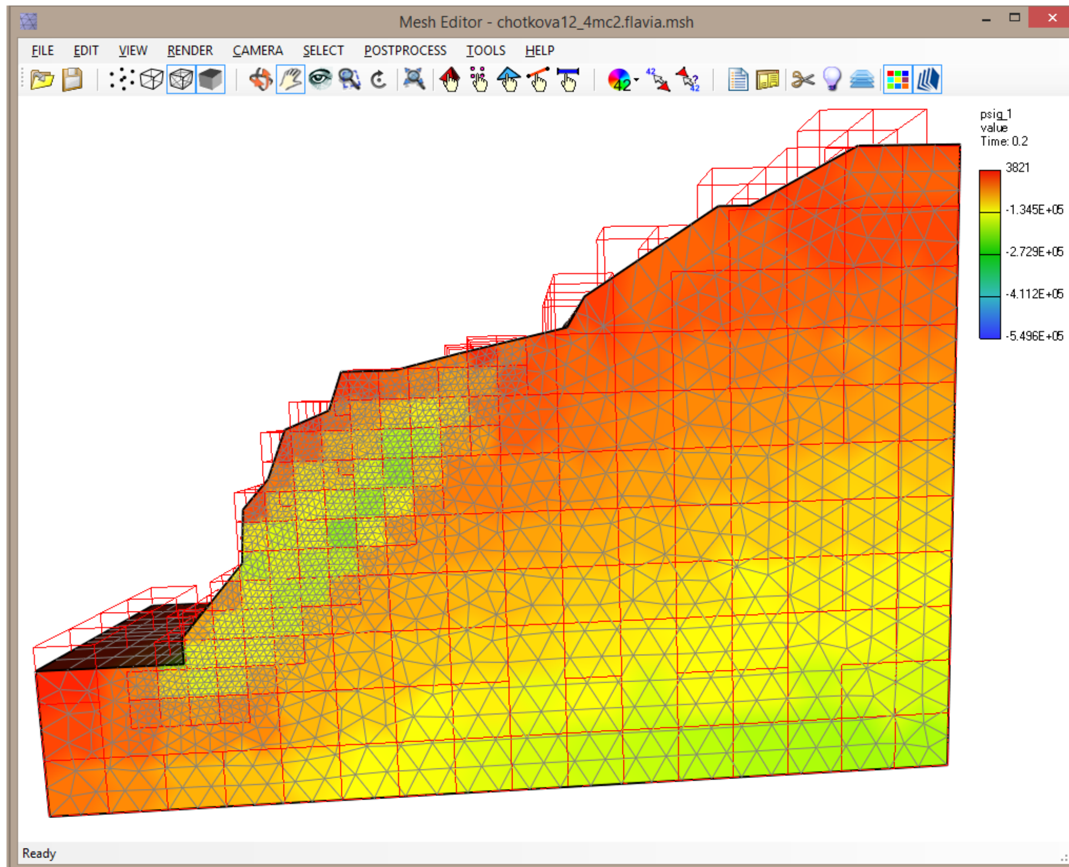


FIGURE 5.10: Geological layers simulation results. Mean value approximation of data values. Nodes in each cell have constant approximated value. High average error of an approximation. High memory consumption due to high average depth of the octree. Non-smooth transition between cells.

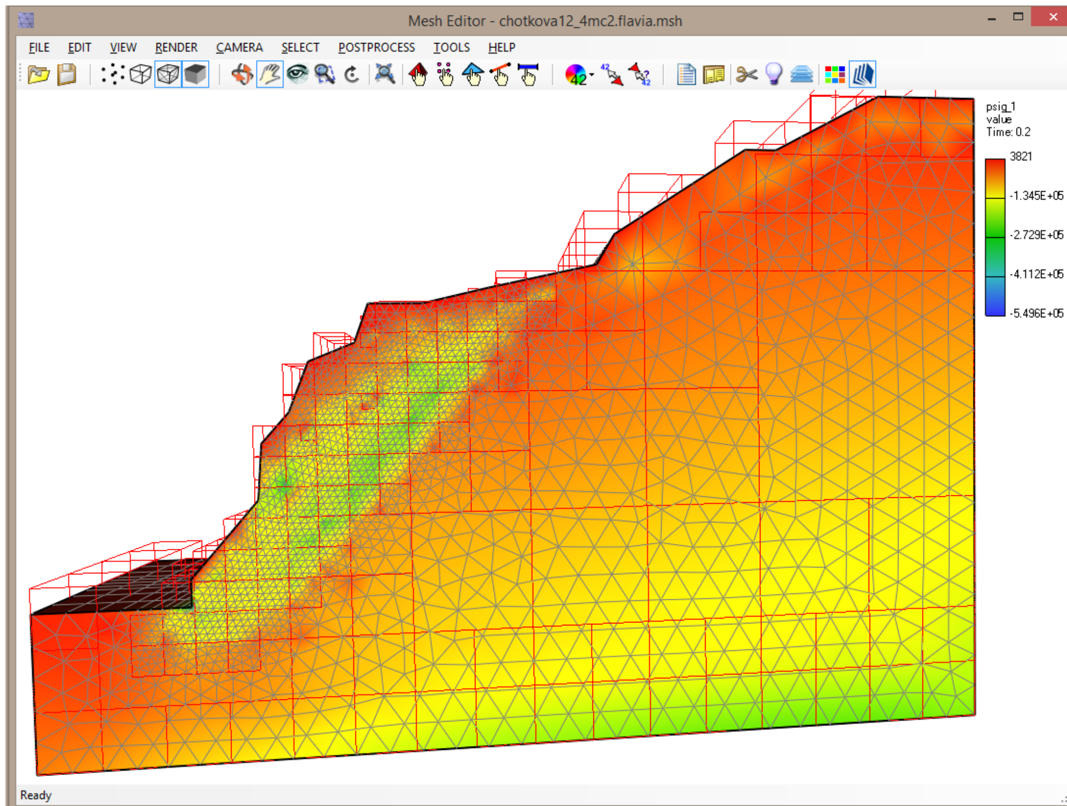


FIGURE 5.11: Geological layers simulation results. Trilinear approximation of data values. Seamless transition between cells. Low depth of the octree in smooth areas. Can't capture high frequent changes in data which is common disadvantage of all lossy compressions.

Figure 5.9 contains visualization of the exact data values whereas Figure 5.10 and Figure 5.11 contains visualization of approximation of the same data series but for different types of approximation functions to highlight the imperfections of the approximation algorithm. The red lines represent the octree cells.

5.2.3 Approximation in time

Additional data compression can be gained by focusing on temporal dimension of function values. Memory allocation can be lowered by eliminating unimportant time steps. To achieve that, it is necessary to find those time steps, in which function values are steady or are changing linearly and can be therefore interpolated from other time steps.

Figure 5.12 illustrates the idea of interpolation in time. Intermediate time steps can be interpolated from the key time steps if they have similar shape – they are *nearly* linear combinations of each other and therefore data in redundant time steps can be disposed. The decision whether to dispose time step or not is based on difference of two functions compared to experimentally designated threshold value. Mathematical background of this procedure is described later in this section.

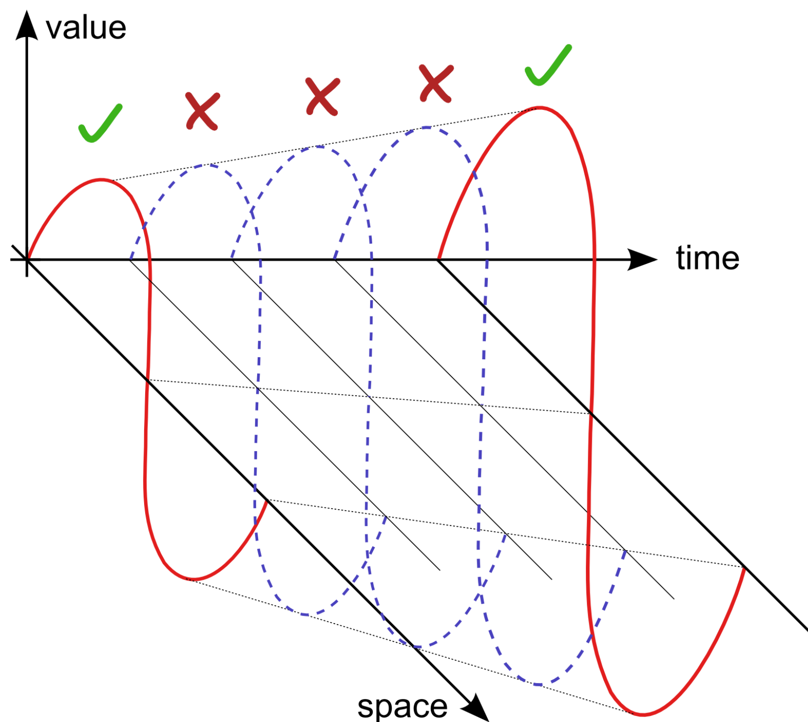


FIGURE 5.12: Interpolation in time. Intermediate time steps (blue) can be interpolated from the key time steps (red) if they have similar shape.

Several options to store temporal data were considered.

- Sequence of octrees. Each one for single time step.
- 4D tree. Extension of octal tree on temporal dimension. Each internal node has 16 children.
- Single ordinary octree containing data approximations with time component. Extending the approximation functions by temporal dimension, e.g., quadrilinear form instead of trilinear form.
- Combination of octree for spatial decomposition and binary tree for temporal dimension division.

Octree for each time frame of an animation is memory inefficient as well as 4D tree [76]. 4D approximation functions have many parameters and also can't capture intricate evolution of function in time. Therefore, different solution was suggested – single octree representing spatial decomposition that is formed by merging sequence of octrees from all time steps. However, the sequence of octrees is just a virtual term. Algorithm 1 treats all data component from all time steps the same and when it finishes its job, only single octree is left with data approximation object for various data component and time steps stored in data catalog in each octree node independent to each other. By merging it is meant the unifying all time steps of a single data component in each octree node into single data sequence object that can be then a subject of the time compression algorithm. This process of merging virtual octrees for each time step into single "time-tree" is illustrated in Figure 5.13. Notice that the octree

for each time step can have different structure and depth, because in each time step the original function can have different frequency spectrum for which the different sampling rate is needed.

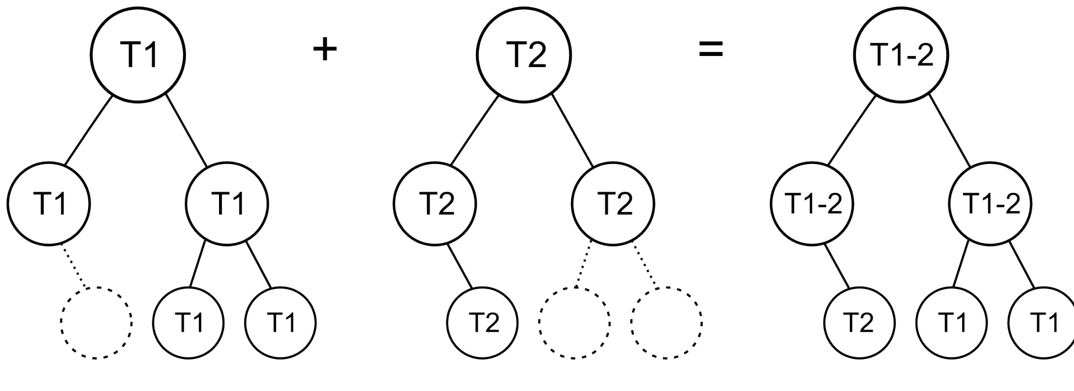


FIGURE 5.13: Illustration of unifying time steps in octree nodes. For the sake of simplicity the binary tree is shown instead of octree. T1 and T2 represent approximation functions of the same data component in different time steps. Result of the union is single octree that contains sequences of corresponding time steps in its nodes.

However, not all approximation functions are preserved during merging. Each octree cell contains list of approximation functions only for key time steps, that are necessary to cover time development of discrete function in area represented by the octree cell. These key (or fixed) time steps are either specified by the user or identified automatically by the algorithm presented in Listing 5.2.

Original nodal value retrieving is the same octree traversal as was depicted in Figure 5.6 except for the final phase – calculating value from approximation function in leaf octree nodes. If desired time instant is not present, then it must be interpolated from the key time steps. To find concrete time instant in the list of approximation functions the fast binary search algorithm is used.

Temporal-data-containing-octree is created by algorithm that works with already created octree having approximation functions for all time steps. At first it finds time steps that must be preserved. Those can be explicitly picked by the user or the algorithm itself determines automatically the first and the last time step as key times. Then the program traverses steps between key events of time interval. In each time step the algorithm iterates over space approximations for each loaded quantities and computes difference between each function and function created by interpolation of functions in key intervals. If this difference is lower than some preset fixed value, approximation function in current time step can be disposed because it can be created on demand in the future. Otherwise time interval is divided in current step and algorithm is recursively called on each of both intervals. It is therefore similar algorithm to approximation in space, but instead of octree a binary tree is used because there is only one temporal dimension instead of three spatial dimensions. Pseudo-code of approximation in time is shown in Listing 5.2.

LISTING 5.2: Approximation in time procedure.

```
function compressTimeSteps(fromIndex, toIndex)
{
  // recursion stopping criterion
  if (toIndex - 1 <= fromIndex)
    return

  fromApproximation = timeSteps[fromIndex].Approximation
  toApproximation = timeSteps[toIndex].Approximation
```

```
  for (index = fromIndex + 1; index < toIndex; index++)
  {
    currentTime = timeSteps[index]
    timeFactor = (currentTime - timeSteps[fromIndex]) / (timeSteps[toIndex] - timeSteps[fromIndex])

    testFunction = timeSteps.Values[index].Approximation
    interpolatedFunction = InterpolatePolynomial(fromApproximation, toApproximation, timeFactor)

    sumDiffSqr = 0.0, sumFuncSqr = 0.0

    // compute difference of testFunction and interpolatedFunction
    foreach (testPoint in domainPoints)
    {
      testValue = testFunction.ComputeValue(testPoint)
      interpolatedValue = interpolatedFunction.ComputeValue(testPoint)
      diff = testValue - interpolatedValue

      sumDiffSqr += diff * diff
      sumFuncSqr += testValue * testValue
    }

    absoluteError = sqrt(sumDiffSqr)
    absoluteValue = sqrt(sumFuncSqr)
    relativeError = absoluteError / absoluteValue

    // if functions are not similar
    if (relativeError <= MAX_RELATIVE_ERROR)
    {
      // discard current time step
      removeTimeStep(currentTime)
    }
    else
    {
      half = (toIndex + fromIndex) / 2
      // recursive call on first half
      compressTimeSteps(fromIndex, half)
      // recursive call on second half
      compressTimeSteps(half, toIndex)
      return
    }
  }
}

// Approximation in time is called upon holes between fixed (key) time steps
// Fixed time steps are at least first and last time step and any interleaved time step specified by user
for (i = 1; i < fixedTimesIndexes.Count; i++)
{
  compressTimeSteps(fixedTimesIndexes[i - 1], fixedTimesIndexes[i])
}
```

In the case of approximation in time, there are two continuous functions that need to be compared to each other. First function is approximation function for the time step that can be potentially removed. This function is already created by spatial approximation algorithm described above. Second function is computed ad-hoc from the key time steps to test if it can potentially replace the first function later. If the test succeeds (functions are similar enough) the first function can be removed entirely from octree data structure, because it can be computed from the functions in neighboring time steps.

The process of computing intermediate function from the key time steps is quite straightforward. With regard to the fact that all approximation functions have to be of the same polynomial type, each parameter of the interpolated function can be then computed as interpolation of the related parameters in two boundary functions in key time steps.

Approximation in time is made for each spatial octree node separately instead of globally for the whole mesh, because the quantity can change in time only in some parts of the mesh and in others can be constant.

Difference between two functions

The algorithm for approximation in time has to compare two continuous approximation functions to determine their similarity.

The difference d of two continuous square-integrable functions u and v is considered as scalar value computed as

$$d = \sqrt{\int_{\Omega} (u - v)^2 \, \mathrm{d}\Omega}. \tag{5.6}$$

If $u \neq 0$ relative difference $\hat{d}$ is then

$$\hat{d} = \frac{\sqrt{\int_{\Omega} (u - v)^2 \, \mathrm{d}\Omega}}{\sqrt{\int_{\Omega} u^2 \, \mathrm{d}\Omega}}. \tag{5.7}$$

To move from continuous to discrete world the integrals can be replaced by sums

$$\int_{\Omega} f(x) \, \mathrm{d}x = \sum_{i=1}^{m} f(x_i) w_i, \tag{5.8}$$

where w_i is the weight of the i-th test point with the meaning of volume surrounding the point, x_i is the location of the test point and m is the number of integration points. The test points in the formula are the data points situated in the area represented by approximation function f in the presented algorithm.

Relative error $\tilde{d}$ is then

$$\tilde{d} = \sqrt{\frac{\sum_{i=1}^{m} (u(x_i) - v(x_i))^2 w_i}{\sum_{i=1}^{m} u(x_i)^2 w_i}}. \tag{5.9}$$

The value of $\tilde{d}$ is then compared to ϵ value. The value $\epsilon = 0.001$ came from the experiments as the best-fitting value. If condition $\tilde{d} \leq \epsilon$ holds, functions u and v are considered equal in terms of approximation in time. The value ϵ matches `MAX_RELATIVE_ERROR` parameter in Listing 5.2.

Results

Results of the approximation in time are summarized in Table 5.3. For spatial approximation the trilinear regression was chosen as the method with the best results in the previous benchmark. The reactor vessel simulation results were used in the test (see Figure 5.7).

TABLE 5.3: Reactor vessel 2D. Spatial and temporal octree approximation results.

	Max error [%]	Average error [%]	Compression ratio [%]
Mean value	100.7 (+25.42)	0.7835 (+0.16)	7.53 (-47.77)
Linear regression	44.23 (+0.0)	0.2262 (+0.042)	2.57 (-15.13)
Trilinear regression	81.41 (+0.14)	0.2158 (+0.041)	2.54 (-14.34)

Time compression procedure presented in Listing 5.2 was applied to already created octree-based structure containing approximations of original data values. Generation of this octree is described in section 5.2.2.

5.3 Evaluation

The thermo-mechanical analysis of model of Charles Bridge in Prague serves as another benchmark that compares maximal relative approximation error, average error, and compression ratio. The analysis results contain displacement vector values

with three components (u, v and w) (see Figure 5.14) and scalar values of temperature distribution (see Figure 5.15). Analysis has 46 time steps. Total number of data sets that are processed by compression algorithm is therefore 184. Each data set has 73749 values that correspond to number of nodes in the finite element mesh. Results are summarized in Table 5.4.

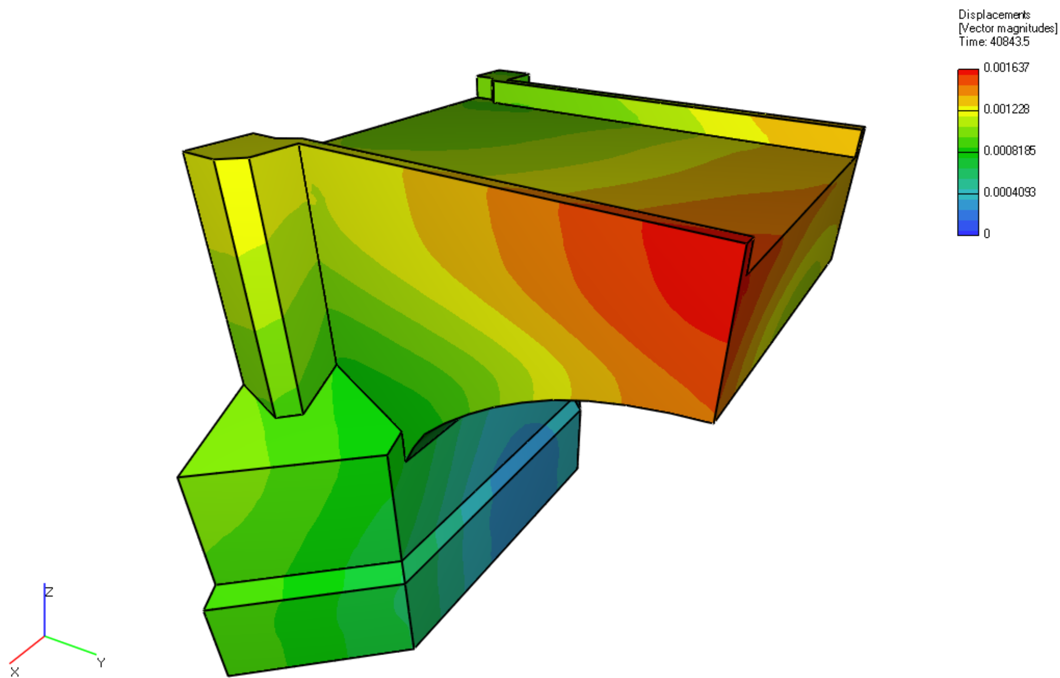


FIGURE 5.14: Charles Bridge analysis: heat transport analysis results (displacements).

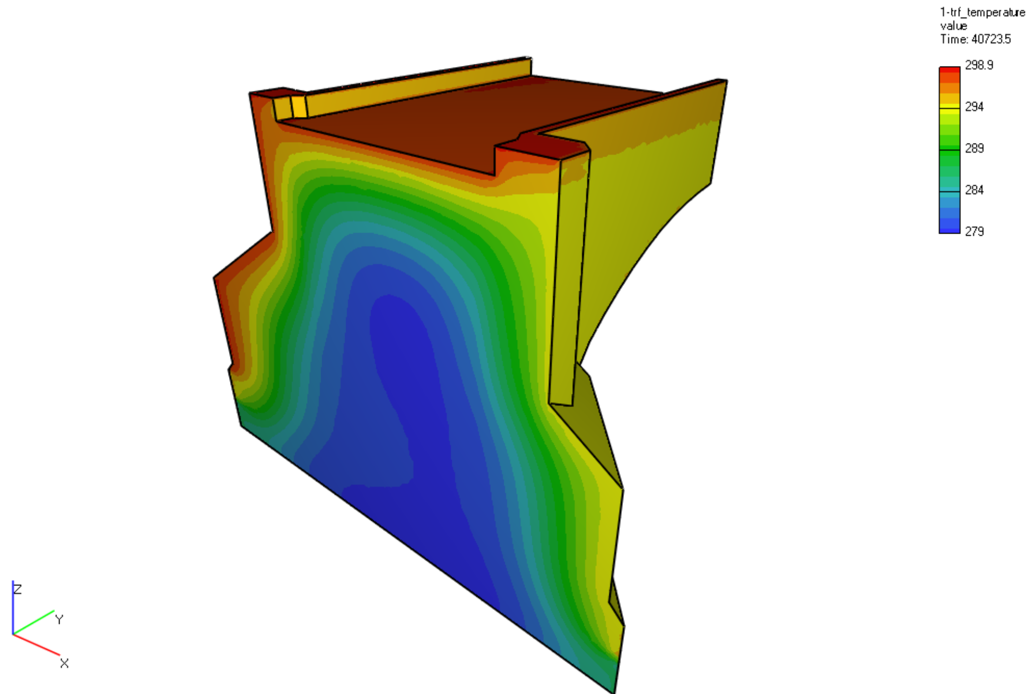


FIGURE 5.15: Charles Bridge analysis: heat transport analysis results (temperature).

TABLE 5.4: Benchmark results: heat transport analysis of Charles Bridge in Prague (approximation error and compression ratio)

	Max error [%]	**Average error [%]**	**Compression ratio [%]**
Displacement u	26.49	0.13	13.3
Displacement v	70.71	0.19	27.4
Displacement w	48.49	0.14	16.3
Temperature	92.46	0.41	59.0
Average		0.22	29.0

Figure 5.16 contains visualization of the exact data values, whereas Figure 5.17 contains visualization of approximation of the same data series. Significant visible approximation errors are marked by arrows. Figure 5.18 shows same kind of error on different data set. The red lines represent the octree segments.

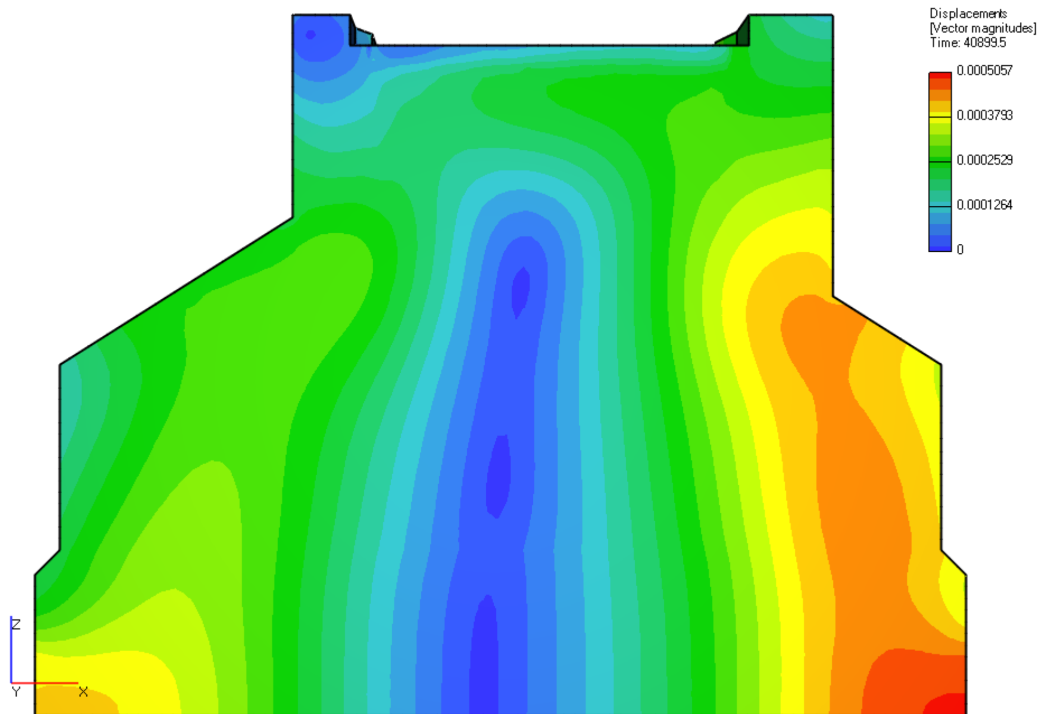


FIGURE 5.16: Charles Bridge analysis: exact data values of heat transport analysis results, no approximation applied.

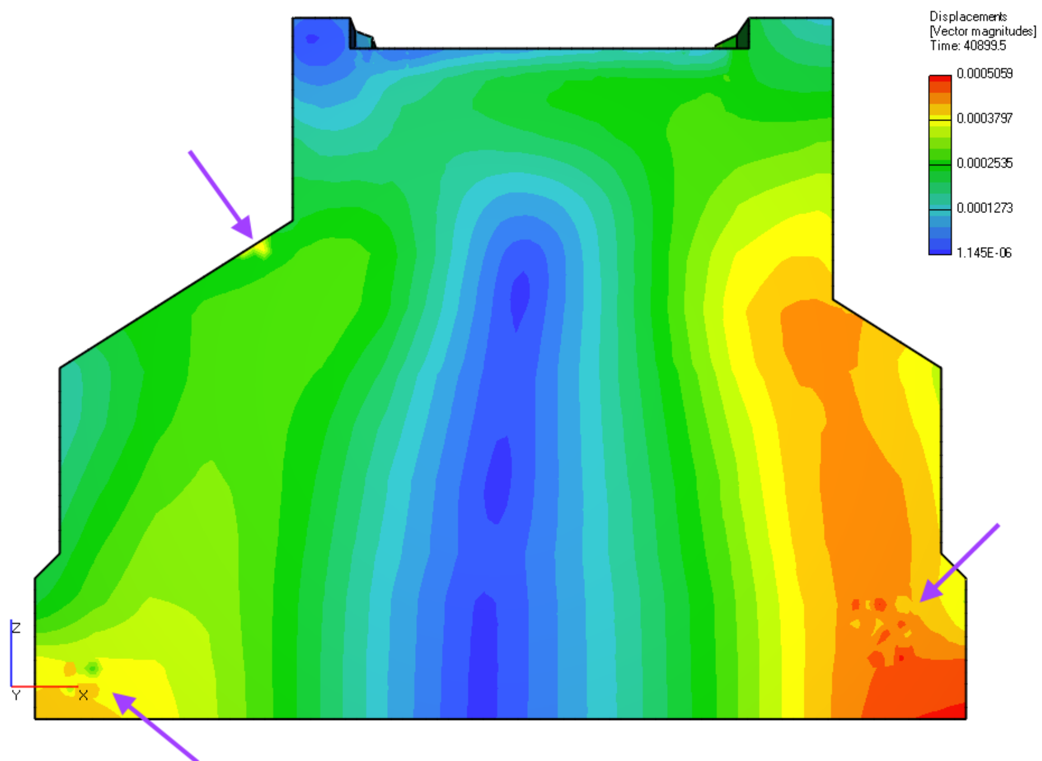


FIGURE 5.17: Charles Bridge analysis: heat transport analysis results, approximation method's artifacts (marked by arrows).

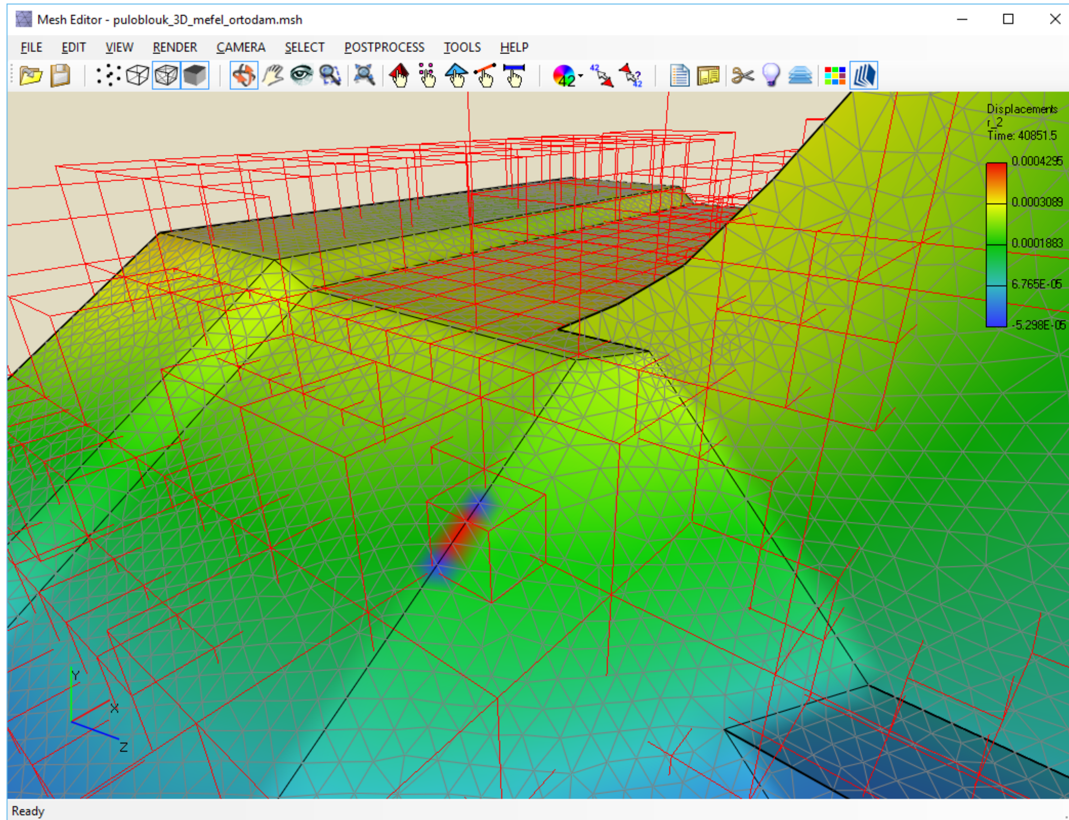


FIGURE 5.18: Charles Bridge analysis: glitches in approximation function caused by octree-based space decomposition. The red lines represent the octree segments.

The imperfections of the method can have two causes. Artifacts can appear typically for results with high-frequency changes in data, for which the octree data structure cannot be fine enough. Special condition in the octree creation algorithm specifies the minimum number of discrete function values to be contained in the octree cell to replace them by the continuous approximation function. This minimum number is related to the type and order of the approximation method that is used in the algorithm. If the condition is not met, the octree cell cannot be divided into eight child cells even if the approximation error is still too high. The possible solution could be to allow the use of higher-order approximation functions in these rare cases, but it would grow the memory consumption and considerably complicate the algorithm. The second reason for these kinds of errors is strictly local nature of the approximation algorithm that cares only about data values in current octree cell and does not take the neighbor segments into account. The transitions between cells can be sometimes far from smooth as can be seen in Figure 5.18.

The method described in this chapter is a lossy compression method. In average case the compression ratio is about 30% with quite low relative approximation error at 0.2%. However, in some extreme cases (rough, unpredictable function shape) the maximal error can be quite high, up to 100% and the method does not even guarantee any upper limit on the approximation error. The compression ratio is not as low as was expected at the beginning, but in the following work the compression ratio can be significantly decreased by applying the same approach also for time – the temporal dimension of the problem. The current algorithm produces the continuous

approximation functions for each time step. The algorithm can be extended to recognize the time steps in which the function does not change or changes linearly and can be therefore interpolated from neighboring time steps. Whole approximation functions in these steps can be then disposed, because they are not necessary – they can be computed from other time steps. However, isolated but unpredictable and excessive maximum approximation error is the crucial disadvantage of the method.

Chapter 6

SVD used for compression of FEA results

Singular Value Decomposition (SVD) is a well known factorization method that provides rich information about matrix systems [77, 78, 79, 80]. One of its many applications is image compression where it can significantly reduce size of data representing image while preserving quality of image appearance. Considering the fact that the results from FEM analyses can be viewed as a series of arbitrary rectangular matrices, the implementation of compression algorithm based on SVD is straightforward as it can be applied to any rectangular matrix. This chapter contains the description of the compression method based on SVD that is the key part of the storage format proposed in this thesis. The content of this chapter is also published in [81].

6.1 Mathematical background

Singular value decomposition is based on a theorem from linear algebra which says that a rectangular matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ can be decomposed into the product of three matrices - an orthogonal matrix $\mathbf{U} \in \mathbb{R}^{m \times m}$, a diagonal matrix $\mathbf{S} \in \mathbb{R}^{m \times n}$, and the transpose of an orthogonal matrix $\mathbf{V} \in \mathbb{R}^{n \times n}$:

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^{\mathsf{T}}, \tag{6.1}$$

where $\mathbf{U}^{\mathsf{T}}\mathbf{U} = \mathbf{I}$, $\mathbf{V}^{\mathsf{T}}\mathbf{V} = \mathbf{I}$. The columns of $\mathbf{U}$ are orthonormal eigenvectors of $\mathbf{A}\mathbf{A}^{\mathsf{T}}$, which are called the left singular vectors. The columns of $\mathbf{V}$ are orthonormal eigenvectors of $\mathbf{A}^{\mathsf{T}}\mathbf{A}$ called the right singular vectors. $\mathbf{S}$ (sometimes referred to as $\mathbf{\Sigma}$) is a diagonal matrix containing singular values in descending order, which are at the same time the nonzero square roots of the eigenvalues of $\mathbf{A}\mathbf{A}^{\mathsf{T}}$ and $\mathbf{A}^{\mathsf{T}}\mathbf{A}$.

SVD can be seen as a method for transforming correlated variables into a set of uncorrelated ones. At the same time, SVD is a method for ordering the dimensions based on variation and identifying the dimension with the largest variation. Once this dimension is identified, it is possible to find the best approximation of the original data points using fewer dimensions. Hence, SVD can be seen as a method for data reduction/compression.

6.1.1 SVD compression

This is the basic idea behind SVD: taking a high dimensional, highly variable set of data points and reducing it to a lower dimensional space that exposes the substructure of the original data more clearly and orders it from the largest variation to the least. What makes SVD practical for data compression applications is that variation

below a particular threshold can be simply ignored to massively reduce data with assurance that the main relationships of interest have been preserved.

The objective of a compression algorithm is to reduce amount of data representing FEM results and also the ability to reconstruct original data from its smaller representation. This saves storage capacity and also accelerates the data transfer between computers as the analysis itself and the post-processing of results is sometimes done on different workstations.

A compression method can be lossy or lossless. Lossless methods are able to fully reconstruct original data. Lossy methods, on the other hand, produce only approximations of original data.

SVD is used in this thesis as a part of the compression algorithm. The SVD method applied to arbitrary matrix produces decomposition that consists of corresponding singular values and singular vectors. This process is fully reversible (with the assumption that the numerical errors are negligible). The original matrix can be reconstructed by the multiplication of decomposed parts. However, the compression algorithm is based on modification of decomposition to create low-rank approximation matrix. The reconstructed matrix slightly differs from the original matrix and algorithm therefore performs lossy compression.

6.1.2 Low-rank approximation matrix

From the definition of SVD in (6.1) and from the properties of SVD, the fact follows that a matrix can be represented in the form of its SVD components as a sum of k rank-1 matrices

$$\mathbf{A} = \sum_{i=1}^{k} s_i \mathbf{u}_i \mathbf{v}_i^{\mathsf{T}}, \tag{6.2}$$

where s_i is the i-th singular value of matrix $\mathbf{A}$, $\mathbf{u}_i$ and $\mathbf{v}_i$ are corresponding singular vectors of matrix $\mathbf{A}$, and $k = \min(m, n)$. Considering the fact that singular values are ordered $s_1 \geq s_2 \geq s_3 \geq ... \geq s_k$, the above formula implies that the first term of the sum would have the highest contribution and the last term would have the lowest contribution to matrix $\mathbf{A}$. Therefore, if we take only first r members of the above summation we get an approximation matrix

$$\mathbf{A}' = \sum_{i=1}^{r} s_i \mathbf{u}_i \mathbf{v}_i^{\mathsf{T}}. \tag{6.3}$$

Quality of approximation depends on the magnitude of the singular values omitted from the approximation formula, namely $s_{r+1}...s_k$. The compression algorithm is based on an assumption that the first singular value is order-of-magnitude higher than singular values at the end of the decomposition sequence. In special cases, when $r = k$, or $s_i = 0$ for all $i > r$, the omitted singular values do not contribute to the sum and the compression is therefore lossless. In other cases, approximation error has to be calculated and taken into account to avoid loss of important details in data.

The main goal of the compression algorithm is to find a compromise between low approximation error and high compression ratio c which is calculated using the formula

$$c = \frac{r(m+n+1)}{mn}, \tag{6.4}$$

where m is the number of rows and n is the number of columns of matrix $\mathbf{A}$. Explanation of the compression ratio formula is best done using Figure 6.1. Light color represents the part of matrix decomposition that is to be stored in the output file as a low-rank approximation of the input.

FIGURE 6.1: Decomposition of input matrix $\mathbf{A}$ into diagonal matrix of singular values $\mathbf{S}$ and matrices of left and right singular vectors. Light color illustrates low-rank approximation.

6.1.3 Error estimation

Low-rank approximation matrix method, which was described above, is a lossy compression technique. Several error metrics are used to control the quality of results.

- **Mean Square Error**

$$MSE = \frac{1}{mn}\sum_{i=1}^{m}\sum_{j=1}^{n}(a_{ij} - a'_{ij})^2, \tag{6.5}$$

  where a_{ij} represents an element of the original matrix and a'_{ij} represents an element of the reconstructed matrix of dimension $m \times n$.

- **Rooted Mean Square Deviation**

$$RMSD = \sqrt{MSE}. \tag{6.6}$$

- **Normalized Rooted Mean Square Deviation**

$$NRMSD = \frac{RMSD}{X_{max} - X_{min}} = \frac{\sqrt{MSE}}{X_{max} - X_{min}}, \tag{6.7}$$

  where X_{min} and X_{max} are elements of input matrix $\mathbf{A}$ with minimum and maximum value, respectively. This error metric is able to measure and compare errors in datasets with different scales. Therefore, it is the main parameter that is used to control the quality of compression in the proposed compression algorithm.

- **Peak Signal to Noise Ratio**

  $PSNR$ is most commonly used to measure the quality of reconstruction of lossy compression methods (e.g., image compression). The signal in this case is the original data, and the noise is the error introduced by compression. $PSNR$ is an approximation to human perception of reconstruction quality. This metric

is not so important in area of FEM analyses, where the human perception of visualizations is not as important as the exact mathematical accuracy of approximations. The reason to include $PSNR$ in results is in particular to allow comparison with other image-related compression methods. $PSNR$ is usually expressed in terms of the logarithmic decibel scale (dB)

$$\begin{aligned} PSNR &= 10\log_{10}\frac{(X_{max}-X_{min})^2}{MSE} = \\ &= 20\log_{10}\frac{X_{max}-X_{min}}{\sqrt{MSE}} = 20\log_{10}\frac{1}{NRMSD} = \\ &= -20\log_{10} NRMSD. \end{aligned} \tag{6.8}$$

- **Normalized Maximum Error**

$$NME = \frac{\|\mathbf{A}-\mathbf{A}'\|_{\max}}{X_{max}-X_{min}} = \frac{\max\limits_{ij}(a_{ij}-a'_{ij})}{X_{max}-X_{min}}. \tag{6.9}$$

6.1.4 Randomized SVD

There are many algorithms with different approaches to compute singular value decompositions. One approach is based on diagonalization of the matrix which essentially yields the whole decomposition at the same time. The other approach is the use of an iterative algorithm that yields one or several singular values at a time and can be stopped after desired number of singular values and vectors has been computed. Although these algorithms have proven to work very well for relatively small matrices, they are not well suited for using with large data sets. The exact SVD of a $m \times n$ matrix has computational complexity $\mathrm{O}(\min(mn^2, m^2n))$ using the "big-O" notation. When applied on large data sets it tends to be very time-consuming. Also, the modern hardware architectures use caches to optimize reading of consecutive memory blocks. As these algorithms often need random access to the memory where the input matrix is stored, it can increase communication between different levels in memory hierarchy, which causes higher latency when accessing data. From a numerical linear algebra perspective, an additional problem resulting from increasing matrix sizes is that noise in the data, and propagation of rounding errors, become increasingly problematic.

In [82, 83, 84, 85], there are described randomized methods for constructing approximate matrix factorizations which offer significant speedups over classical methods. The particular implementation of the randomized decomposition is based on the algorithm described in [86]. The authors proposed an algorithm for efficient computation of low-rank approximation to a given matrix. The method uses random sampling to identify a subspace that captures most of the action of a matrix. The input matrix is compressed to this subspace, and deterministic manipulations are then used to obtain the desired low-rank factorization. For a matrix that is too large to fit in fast memory, the randomized techniques require only a constant number of passes over the data, as opposed to $\mathrm{O}(k)$ passes for classical algorithms.

The algorithm can be split into two main computational stages. The first stage is to construct a low-dimensional subspace that captures the action of the matrix. To be more formal, this stage is to compute an approximate basis for the range of the input matrix $\mathbf{A}$. This basis matrix $\mathbf{Q}$ is required to have orthonormal columns and

$$\mathbf{A} \approx \mathbf{Q}\mathbf{Q}^\mathsf{T}\mathbf{A}. \tag{6.10}$$

Matrix $\mathbf{Q}$ is desired to contain as few columns as possible while producing accurate approximation of matrix $\mathbf{A}$ at the same time.

The second stage is to use $\mathbf{Q}$ to obtain approximate SVD factorization of $\mathbf{A}$. This can be achieved using simple deterministic steps:

1. Construct $\mathbf{B} = \mathbf{Q}^\mathsf{T}\mathbf{A}$.
2. Compute an exact SVD of the small matrix: $\mathbf{B} = \mathbf{W}\widetilde{\mathbf{S}}\widetilde{\mathbf{V}}^\mathsf{T}$.
3. Set $\widetilde{\mathbf{U}} = \mathbf{Q}\mathbf{W}$.

The main challenge is therefore to efficiently construct r orthonormal vectors forming the matrix $\mathbf{Q}$ that (nearly) span the range of $\mathbf{A}$; r is the desired rank of approximation and is supposed to be substantially less then both dimensions of $\mathbf{A}$. After that an SVD that closely approximates $\mathbf{A}$ can be constructed (closely in the sense that the spectral norm of the difference between $\mathbf{A}$ and the approximation to $\mathbf{A}$ is small relative to the spectral norm of $\mathbf{A}$).

In order to estimate the range of matrix $\mathbf{A}$, it is applied to a collection of r random vectors. The result of applying $\mathbf{A}$ to any vector is a vector in the range of $\mathbf{A}$, and if the matrix is applied to r random vectors, the results will nearly span the range of $\mathbf{A}$ with extremely high probability. Mathematical proofs given in [86] and [87] show that the probability of missing a substantial part of the range of $\mathbf{A}$ is negligible if the vectors to which matrix $\mathbf{A}$ is applied are sufficiently random (i.e., entries of these vectors are independent and identically distributed).

Therefore, matrix $\mathbf{A}$ is applied to a random Gaussian matrix $\mathbf{\Omega}$ that contains r columns with random normally distributed entries yielding the matrix $\mathbf{Y} = \mathbf{A}\mathbf{\Omega}$. Applying the Gram-Schmidt process (or any other method for constructing QR decomposition) produces the decomposition $\mathbf{Y} = \mathbf{Q}\mathbf{R}$, where columns of $\mathbf{Q}$ are an orthonormal basis for the range of $\mathbf{Y}$, and since columns of $\mathbf{Y}$ nearly span the range of $\mathbf{A}$, $\mathbf{Q}$ is an orthonormal basis for the approximate range of $\mathbf{A}$.

$\mathbf{A}$ is then decomposed as

$$\mathbf{A} \approx \mathbf{Q}\mathbf{Q}^\mathsf{T}\mathbf{A} = \mathbf{Q}\mathbf{B} = \mathbf{Q}\mathbf{W}\widetilde{\mathbf{S}}\widetilde{\mathbf{V}}^\mathsf{T} = \widetilde{\mathbf{U}}\widetilde{\mathbf{S}}\widetilde{\mathbf{V}}^\mathsf{T}. \tag{6.11}$$

The algorithm produces matrices $\widetilde{\mathbf{U}}$ and $\widetilde{\mathbf{V}}$ with orthonormal columns being approximations of the left and the right singular vectors of matrix $\mathbf{A}$, and a nonnegative diagonal matrix $\widetilde{\mathbf{S}}$ that contains approximations of the first r singular values of matrix $\mathbf{A}$. For a dense input matrix, randomized SVD algorithm requires $\mathrm{O}(mn\log r)$ floating-point operations, substantially less than classical algorithms.

6.2 Implementation

Results from the finite element method are scalar, vector or tensor fields represented by discrete values calculated in nodes of the mesh or in integration points on finite elements. In order to compress data, an auxiliary matrix $\mathbf{A}$ has to be assembled from the results. The number of rows of the matrix $\mathbf{A}$ is equal to the number of incremental or time steps while the number of columns is equal to the number of points in which the results are stored. Such auxiliary matrix is assembled for each scalar field and for each component of the vector and tensor fields. It means, three matrices

corresponding to the displacement in the x, y, and z directions are assembled for the vector of displacements in three-dimensional problems.

There are two main reasons to store particular results in separate matrices. First, the size of matrices is smaller than the size of a matrix which contains all results and therefore SVD will be performed faster. Second, the magnitudes of particular fields are very different (the stress tensor components are several order of magnitude larger than the components of the displacement vector) and the data compression algorithm would suppress the fields with small magnitudes. Once the matrix $\mathbf{A}$ is assembled for each field, the compression algorithm can be applied on it. It is purely algebraic procedure and no information about geometry of the mesh is needed.

Let us assume that the matrix is not empty and is full rank. Then it follows from the formula (6.4) that if r is equal to the rank of matrix $\mathbf{A}$, the compression ratio is always higher than one. In other words the memory consumption of stored decomposition is bigger than the size of the original matrix. To make the compression algorithm applicable, the parameter r must satisfy the condition

$$r < \frac{mn}{m+n+1}. \tag{6.12}$$

Considering the usual shape of matrix containing FEM results, this inequality is easily satisfiable even for the r being close to the rank of the original matrix as in the typical case the number of nodes or integration points is much higher than the number of analysis steps and therefore $m \ll n$.

6.2.1 Algorithm description

Once SVD is calculated, the compression algorithm removes a certain number of singular values and corresponding singular vectors. The remaining singular values and vectors represent the compressed data. There are two strategies that influence the way how to preserve the number of singular values – resulting size and quality. Each strategy is assigned a control parameter that determines compression ratio or approximation error.

Compression ratio. If the focus is only on the size of compressed data, the rank r of the approximation matrix can be calculated by the formula

$$r = \left\lceil c \times \frac{mn}{m+n+1} \right\rceil, \tag{6.13}$$

where c is the compression ratio, $0 \leq c \leq 1$ (0 results in absolute compression while 1 results in no compression); $\lceil . \rceil$ is the ceiling function.

Approximation error. In a usual case, the most important measure to take into account is the approximation error. Algorithm is trying to minimize the compression ratio while at the same time ensuring that predefined approximation error threshold is not exceeded. To quantify the error, the Normalized root-mean-square deviation ($NRMSD$) is used. The normalized error metric enables working with various data sets that have different scales. $NRMSD$ is defined in Section 6.1.3.

To effectively calculate the final rank of the approximation matrix from the desired approximation error, the interesting property of singular values

$$\sum_{i=1}^{m}\sum_{j=1}^{n}(a_{ij})^2 = \sum_{i=1}^{k} s_i^2, \tag{6.14}$$

where $k = \min(m, n)$, i.e., the smallest of two dimensions of the matrix $\mathbf{A}$, is made use of. The above formula states that the sum of squared elements of the matrix $\mathbf{A}$ equals to the sum of squared singular values s_i of the same matrix $\mathbf{A}$.

Using formulas (6.2) and (6.3) the equation (6.14) can be applied to the difference between original matrix $\mathbf{A}$ and approximation matrix $\mathbf{A}'$

$$\sum_{i=1}^{m}\sum_{j=1}^{n}(a_{ij} - a'_{ij})^2 = \sum_{i=r+1}^{k} s_i^2, \tag{6.15}$$

where the term on the right-hand side is the sum of squares of those singular values of the matrix $\mathbf{A}$ that are going to be cut away by the compression algorithm. The equation can be rewritten using the definition of MSE in (6.5) to

$$MSE \times mn = \sum_{i=r+1}^{k} s_i^2 \tag{6.16}$$

and using (6.7) further to

$$(NRMSD \times (X_{max} - X_{min}))^2 \times mn = \sum_{i=r+1}^{k} s_i^2. \tag{6.17}$$

Then $NRMSD$ can be used as a quality metric for the compression algorithm because normalization makes it usable for different datasets. Calculation of rank of the approximation matrix is depicted as pseudo-code in Algorithm 1. Algorithm uses the inequality

$$e > \frac{\sqrt{\frac{\sum_{i=r+1}^{k} s_i^2}{mn}}}{X_{max} - X_{min}} \tag{6.18}$$

to test whether the desired rank has been reached; e is $NRMSD$ used as an error threshold that can not be exceeded to achieve desired quality of approximation.

6.2.2 Optimization

Computational complexity of the exact SVD algorithm is $\mathrm{O}(m^2 n)$, where $m < n$. This theoretical algorithm complexity is confirmed by two benchmarks where the dependency of the execution time on the varying matrix dimension is shown. The results of the benchmarks are depicted in Figure 6.2 and Figure 6.3. Several observations were made from the results:

- The algorithm is most efficient in cases where one dimension of the input matrix is very small compared to the other. However, this is almost always the case when compressing results from FEM – number of incremental or time steps seldom exceeds hundreds.
- Moreover, incremental or time steps can be devided into smaller ranges and the algorithm can be applied on each range separately. This will improve performance and can also increase quality of compression if the key time steps on the range boundaries are carefully selected.

Algorithm 1 Calculation of rank for approximation matrix from maximum allowed error.

INPUT: maximum allowed error ($e : e > 0$), array with singular values ($S : S.length > 0$), element count ($c : c > 0$), maximum element value (x_{max}), minimum element value ($x_{min} : x_{max} > x_{min}$)

OUTPUT: rank of resulting matrix

```
1:  procedure CALCULATERANK(e, S, c, x_max, x_min)
2:      MSE ← 0
3:      NRMSD ← 0
4:      rank ← S.length
5:      while NRMSD < e do                          ▷ repeat until max error is reached
6:          MSE ← MSE + S[rank]/c                   ▷ calculate MSE for current rank
7:          NRMSD ← √MSE/(x_max − x_min)            ▷ normalize error
8:          rank ← rank − 1                         ▷ decrement rank for next loop
9:      end while
10:     return rank + 1          ▷ Add one to not exceed maximum allowed error
11: end procedure
```

- The randomized SVD algorithm has the same order of algorithmic complexity when full decomposition is required, but yet can significantly reduce execution time. However, the benchmarks are not designed to highlight the benefits of randomized SVD algorithms. The main advantage of the randomized SVD is in the ability to choose the rank of the approximation matrix in advance. In that case only limited number of singular values and corresponding singular vectors are calculated and algorithm performs much faster.

Storage size of SVD itself can also be optimized. **S**, being a diagonal matrix, can be stored as single list of singular values s_i, or can be even multiplied with the matrix of left singular vectors **U**.

6.3 Results

All procedures presented here were tested on a common PC having Intel Core i5-4690K @ 3.5GHz CPU with 16GB RAM, running on Microsoft Windows 10 64-bit operating system.

The first benchmark was designed to measure computational complexity of the SVD algorithm. Series of 100 random matrices with standard distribution were generated and execution times were recorded and averaged. The execution time with respect to the number of the stored incremental or time steps (the number of rows in the matrix **A**) is depicted in Figure 6.2 while the execution time with respect to the number of points, where the results are stored (the number of columns of the matrix **A**), is depicted in Figure 6.3. Especially in Figure 6.3, it is clearly visible that the randomized SVD algorithm is much faster than the classical one.

These results confirm that the SVD implementation has computational complexity $O(m^2n)$, where m is number of rows, n is number of columns, and $m < n$. In case of $m > n$ the complexity would be $O(mn^2)$ as the algorithm takes advantage of non-squareness in that its complexity is quadratic only in the smaller dimension.

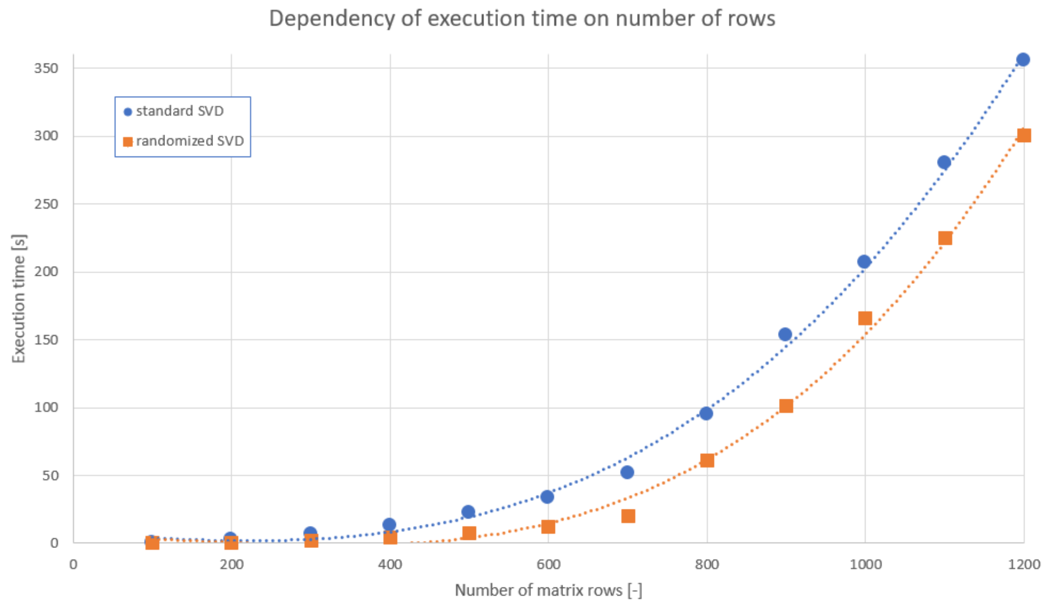


FIGURE 6.2: Dependency of SVD execution time on m (having fixed $n = 10000$ and $r = min(m, n)$).

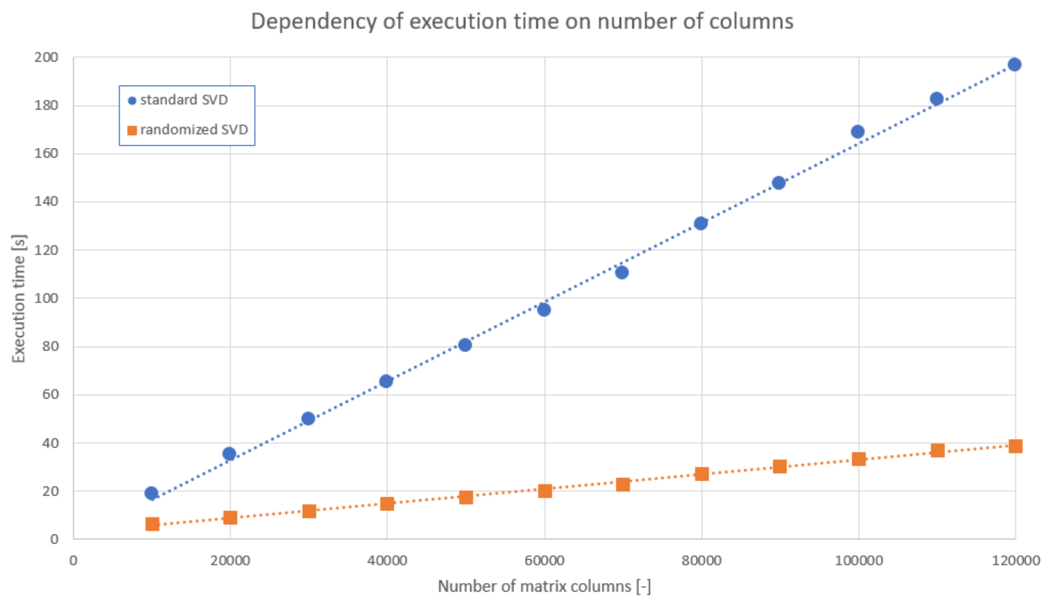


FIGURE 6.3: Dependency of SVD execution time on n (having fixed $m = 100$ and $r = min(m, n)$).

Behavior of the compression strategy introduced is presented on three real world examples. First example is an analysis of aging of nuclear power plant's containment made from prestressed concrete. The finite element mesh used in this analysis is in Figure 6.4. More details about the analysis can be found in [88] and [89]. This analysis includes high number of analysis time steps (thousands) with very little differences between them. There is therefore potential for compression to be very

effective (compression ratio to be very low) as proven in Figure 6.5 that examines the impact of changes in the compression ratio to the mean error of approximation.

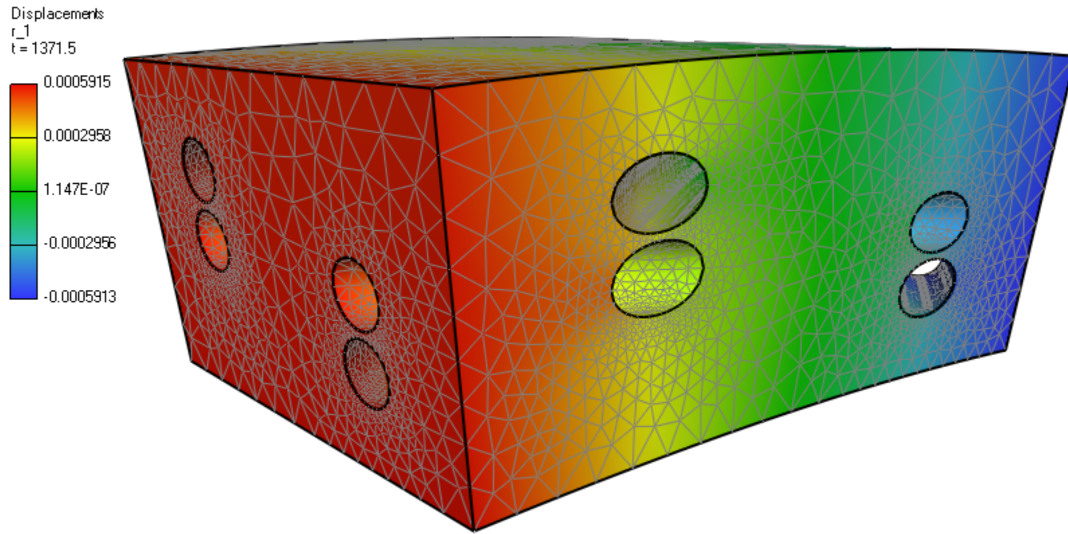


FIGURE 6.4: Segment of reactor containment analyzed. Results visualization (displacement field, x component).

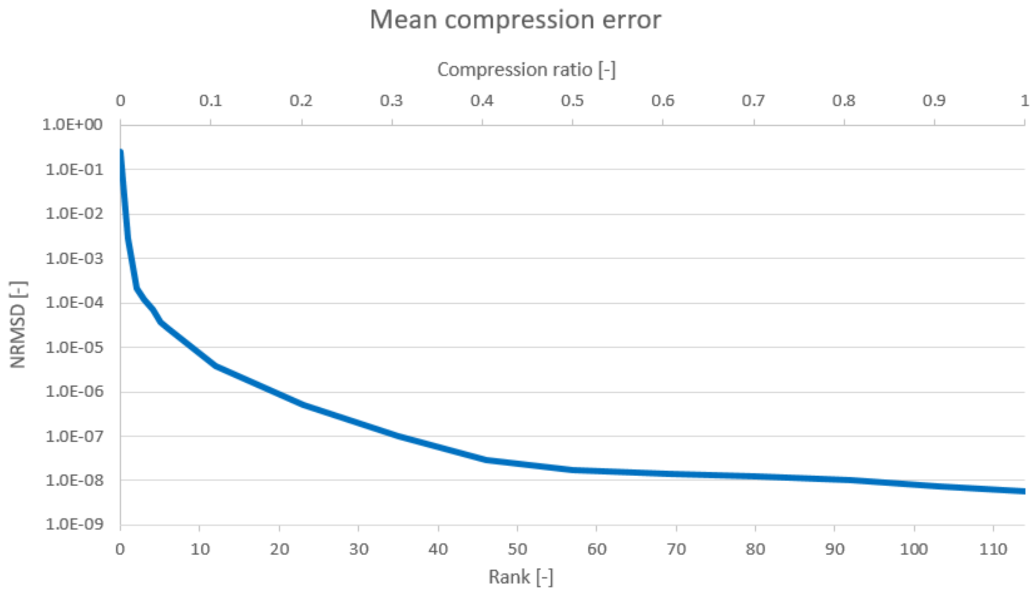


FIGURE 6.5: Dependence of $NRMSD$ on c and r for reactor containment analysis results.

Figure 6.6 shows results from an analysis of geological layers which was based on theory of plasticity. More details can be found in [90]. This project was chosen mainly to study behavior of compression algorithm when dealing with high discontinuities in data in spatial dimension (as can be seen in visualization). As summarized in Figure 6.7 and Figure 6.8 this has negligible effect ($NRMSD$ and NME are bellow 1% even for very small r – 3 out of 22) on quality of compression.

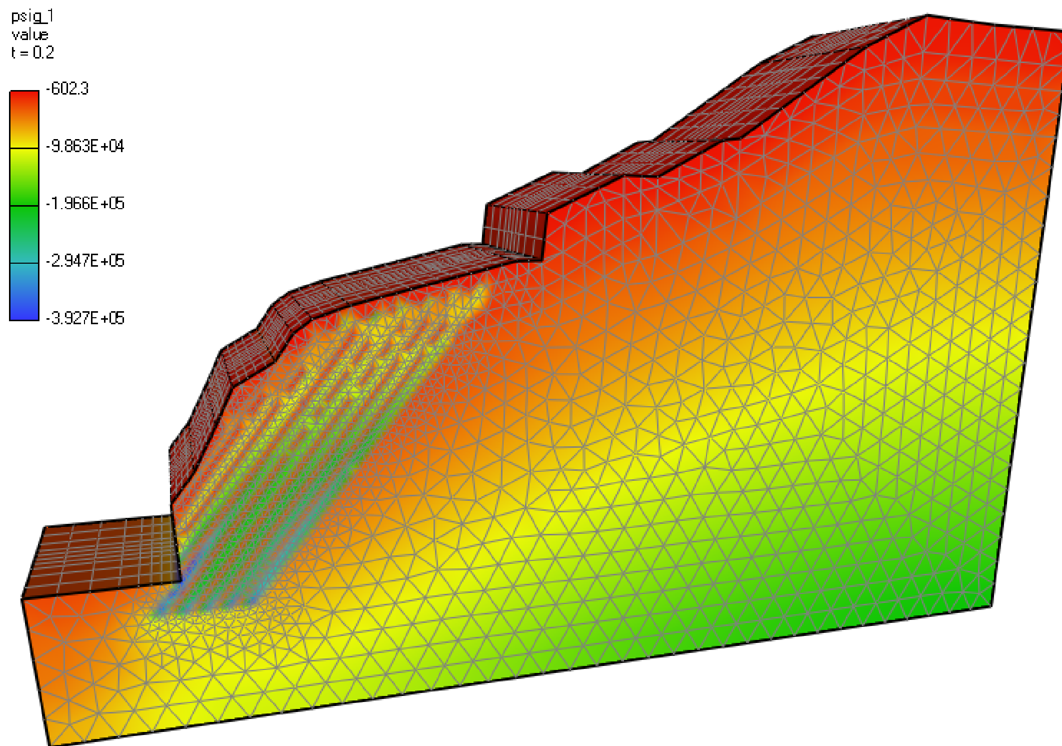


FIGURE 6.6: Analysis of geological layers. Results visualization (stress field, sigma XX component).

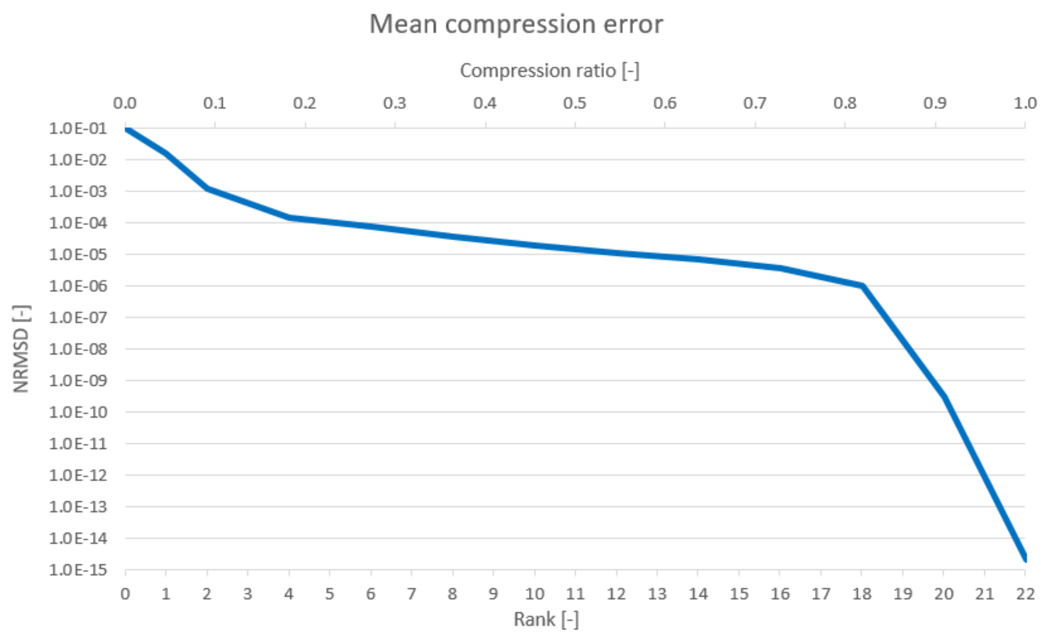


FIGURE 6.7: Dependence of $NRMSD$ on c and r for results of geological layers project.

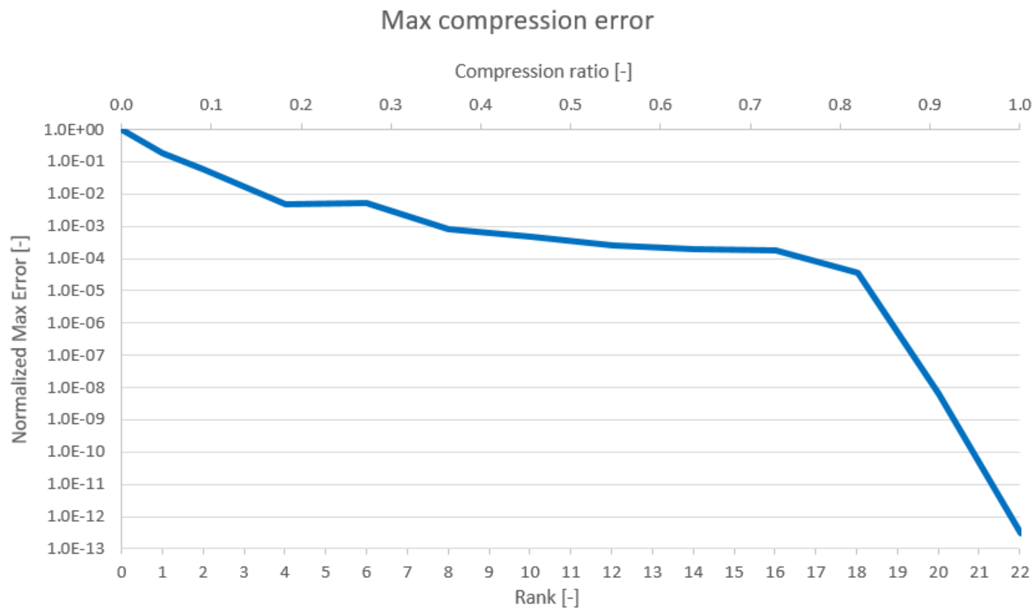


FIGURE 6.8: Dependence of NME on c and r for results of geological layers project.

Figure 6.9 contains visualization of results of two-dimensional analysis, where axisymmetric description was used for analysis of aging of a reactor vessel. Details about the analysis can be found in [91]. There are exactly 232 analysis time steps. The resulting data has linear function character with several discontinuities in temporal dimension. There are few time steps in which resulting discrete functions have very different values compared to neighboring time steps. This was supposed to have negative impact on the quality of compression. However, as can be seen in Figure 6.10, the quality is better than expected; e.g., if the rank of approximation matrix is set to 3 (compared to 232 being the rank of the original matrix) the normalized relative error ($NRMSD$) does not exceed 10^{-5}.

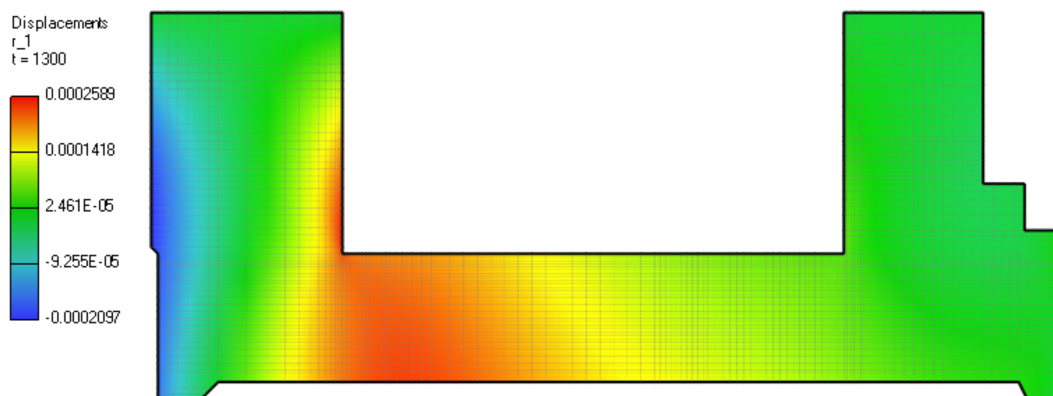


FIGURE 6.9: 2D model of a reactor vessel. Results visualization (displacement field, x component).

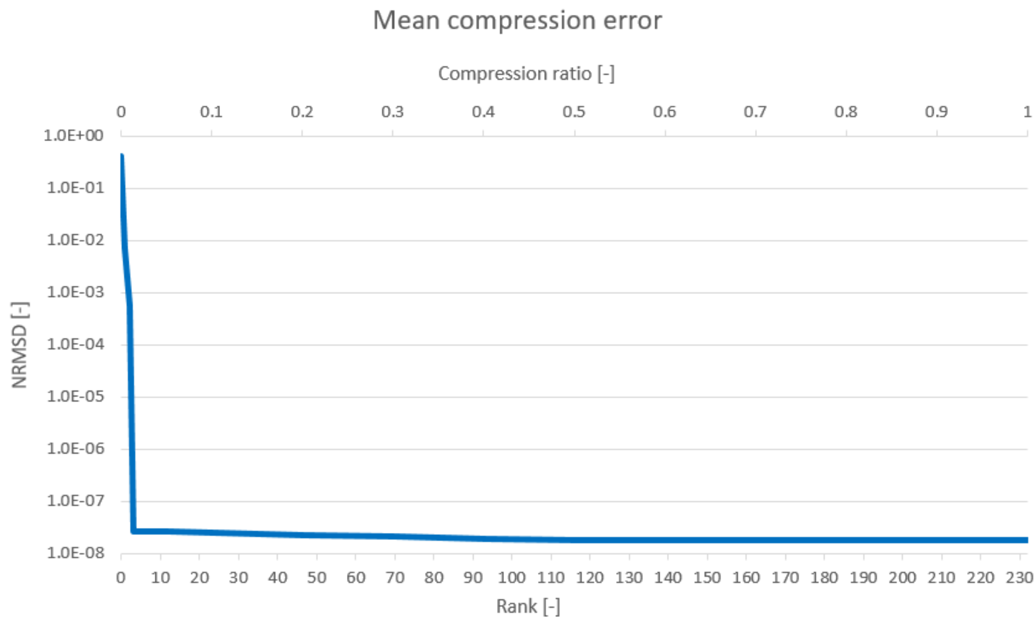


FIGURE 6.10: Dependence of $NRMSD$ on c and r for reactor vessel analysis results.

Figure 6.11 summarizes the compression error for all three benchmarks using $PSNR$ metric. $PSNR$ is defined using logarithm (see Equation (6.9) for definition), and is included here mainly as a comparison to other image-related compression methods whose quality is often expressed by $PSNR$. Figure 6.12 contains the same information for the randomized SVD algorithm.

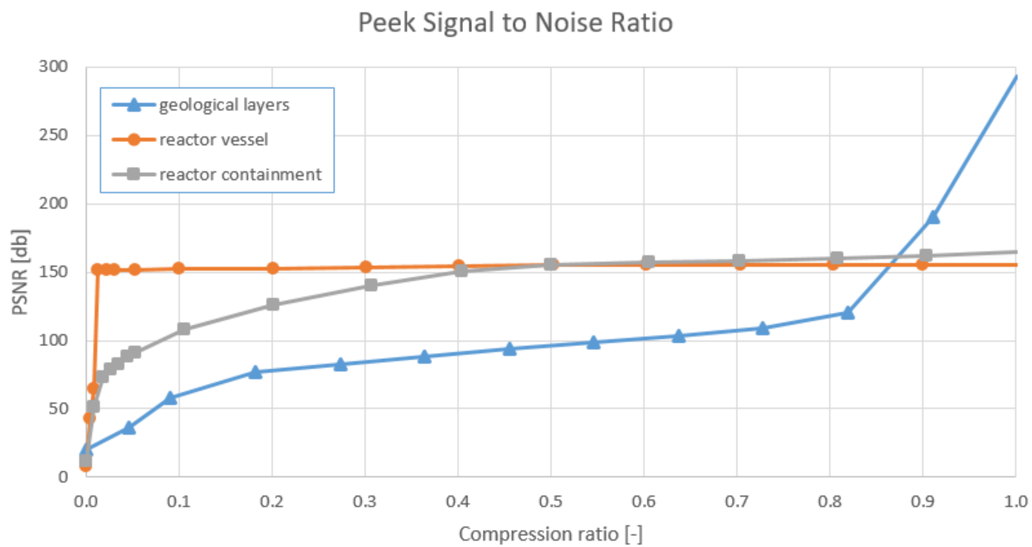


FIGURE 6.11: Dependence of $PSNR$ value on c and r calculated for different SVD decompositions.

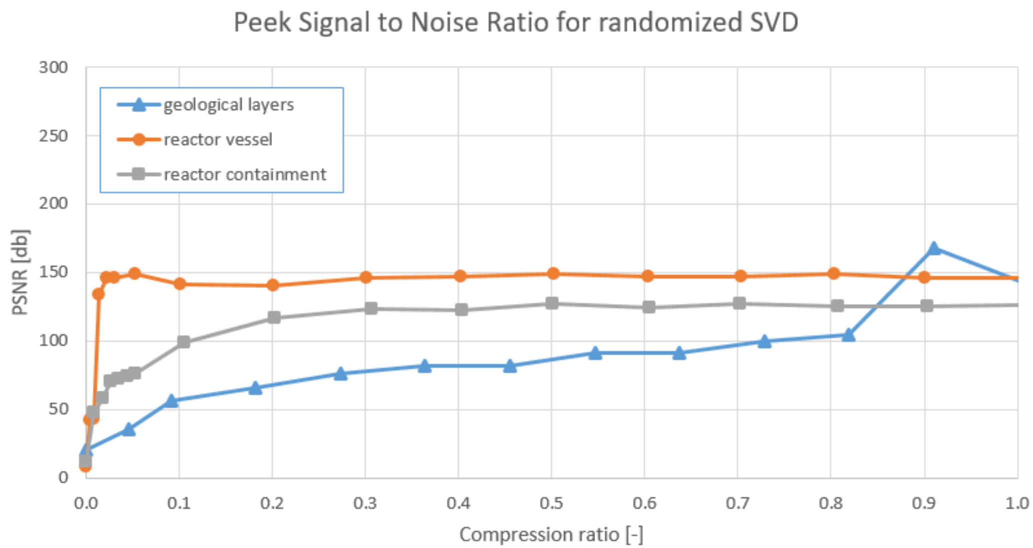


FIGURE 6.12: Dependence of $PSNR$ value on c and r calculated for different randomized SVD decompositions.

Besides the error also the execution speed of compression algorithm was measured. In Figure 6.13, there is a comparison of execution times for standard versus randomized SVD compression algorithms. Interestingly, execution time of standard SVD is independent of target rank whereas execution time of randomized SVD decreases linearly with decreasing target rank. If the rank is known ahead, the fact can be taken advantage of.

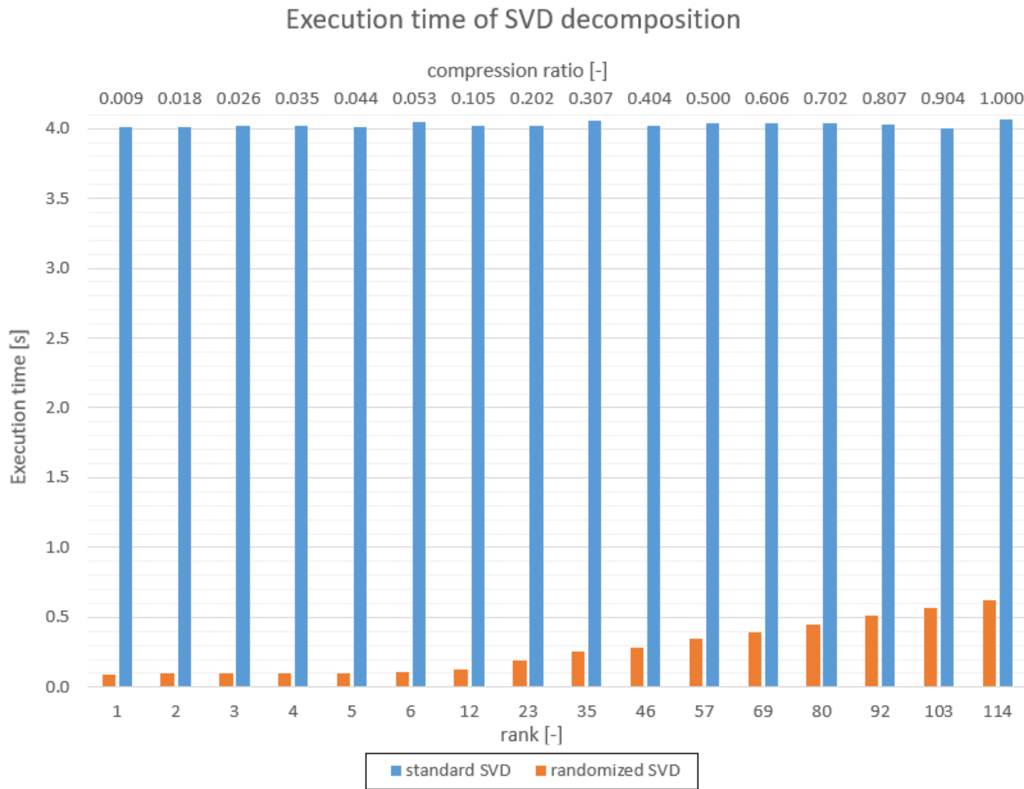


FIGURE 6.13: Variation of execution time of standard and randomized SVD decompositions calculated for reactor containment analysis results.

The memory consumption of compressed results for reactor containment analysis is summarized in Table 6.1. For different values of compression ratio it shows memory size in megabytes. In this benchmark, the compression ratio c is an input parameter to the compression algorithm. As follows from the Equation (6.13), the value of the compression ratio directly affects the amount of singular values ought to be removed from SVD. Size factor describes the final outcome of compression when compared to the original size.

TABLE 6.1: Memory consumption of compressed results. 3D reactor containment analysis.

compression ratio (c)	memory consumption [MB]	size factor
1.00	2002.1	1
0.50	1006.9	0.5002
0.10	211.9	0.1053
0.01	35.3	0.0176

Chapter 7

Conclusions

There were three main goals of the thesis as described in Section 1.2. The first goal was to design a new storage format for representation of the results from the finite element method with the support for compression. This goal was fulfilled. Besides the compression, the new storage format supports efficient querying of the data, unlike the standard unstructured file-based formats, which require parsing through the complete set of results to retrieve a specific information. Examples of the proposed format in form of JSON documents are given. Also, the application that can convert the results from the FEM solver to the new storage format was implemented. This converter application further supports generation of visual filters and it is designed to be run either locally on a PC or remotely in a cloud environment.

The second goal of the thesis was to investigate suitable methods for compression of results from FEM and develop a compression algorithm with reasonable performance characteristics and producing approximations with low and predictable error. The compression method based on singular value decomposition satisfies these requirements. The SVD compression method became the integral part of the storage format. The results of its application on real data have been presented. The algorithm is able to compress arbitrary data using low-rank approximation matrices. When the maximum allowed error was set to 10^{-5}, the compression ratio was at most 10% for all tested results. In many cases, the compression ratio can be even better – bellow 1% of the original size. The important property of the compression algorithm is the fact that the approximation error can be set in advance and there is a guarantee that it will not be exceeded. The disadvantage of the SVD-based compression method is the computational complexity. SVD is a very time-consuming operation. However, this operation is performed only once during the conversion of results from FEM solver to the storage format, before the post-processing is started. Also, the randomized version of the decomposition algorithm is much faster and can be used if a slight increase of the approximation error is tolerated.

The third goal was the implementation of two post-processors that demonstrate the proposed methods. Both the desktop and the web post-processor were implemented and described in detail. The desktop post-processor is a feature-rich visualization tool that allows to visualize the data in various formats including the new proposed storage format. It is able to create efficient surface representation of an arbitrary finite element mesh and it implements advanced techniques for manipulation with the mesh entities. The web-based post-processor is a simple cross-platform application that is able to visualize the simulation results located in a remote storage. As the hard work connected with processing of the results is offloaded to the server, the web application is just a thin client that works even on devices with limited CPU and memory resources.

Besides the presented goals, the thesis also outlines the architecture of the data

access system for complex FEA consisting of several independent services. The system is designed as a collaborative framework that can be accessed by users from different client devices. The web post-processor is built on top of this data management system, it directly communicates with the web API service to provide the user with the access to FEA simulations running on a remote server. The database schema for project and simulation related data is given as well as the description of individual services.

As a not very suitable method for data reduction is considered the approximation of the FEM results by polynomial functions presented in Chapter 5. The method was inspired by the multigrid method (and generally other multi-mesh methods, hence the name of the thesis) that was at the beggining of the research work. The multigrid method allows to solve partial differential equations using the hierarchy of domain discretizations. The idea was to connect the FEM solution phase with the post-processing by reusing the mesh hierarchy used by the multigrid method also in the post-processing of the results. Although the presented approximation method is capable of a significant reduction of the data size (up to 2.5% of the original size), the maximal approximation error can be very high (up to 100% in extreme cases, e.g., when there are discontinuities or singulatities in the data). The unpredictibility of the error and the high decompression time are the reasons the method is excluded from the implementation of the post-processors in favor of the SVD compression method presented in Chapter 6.

7.1 Future work

The proposed solutions can be further extended to support features that are beyond the scope of this research project. There are also ways to improve the methods and algorithms presented in this thesis. The list of a few selected ideas follows.

- The quality of output from the SVD compression method can be improved by incorporating the sparse matrix of details [82] along with the already used low-rank approximation matrix.
- The list of visual filters supported in the storage format can be extended to include additional filters used in other post-processing tools (clip, stream-lines, etc.).
- The proposed data management system can be extended to support the pre-processing phase of FEA. Probably the most feasible way to do it would be to implement the import of the geometric model generated by an existing CAD tool and create a basic pre-processor that will be able to assign attributes to the model.
- To avoid the overhead related to the conversion of the FEM results, the FEM solver could be updated to store the results directly in the proposed storage format.

Appendix A

Data format for storage and transport of FEM results

LISTING A.1: Example of solution.json document.

```
{
    "Id": 42,
    "ProjectName": "Shear beam 3D",
    "Location": "https://fea-cloud-service.net/postprocess/42",
    "Results": [
      {
        "MeshRecordNames": [
          "beam.msh"
        ],
        "DataRecordNames": [
          "beam.res"
        ]
      }
    ],
    "Layers": [
      {
        "Id": "5b585758-9f64-4790-a765-64709951931a",
        "Name": "master",
        "FilterType": null,
        "Children": [
          {
            "Id": "09dfdc8c-a75b-48e1-b319-a9624312a5a5",
            "Name": "deformation (scale: 0.6)",
            "FilterType": "Deformation",
            "Children": [
              {
                "Id": "ee52969e-f862-4daf-85b9-5a8224197669",
                "Name": "slice (offset: -0.1569261)",
                "FilterType": "Slice"
              },
              {
                "Id": "a86486b1-53eb-43cc-aa22-d670bcdea163",
                "Name": "slice (offset: -0.4757478)",
                "FilterType": "Slice"
              },
              {
                "Id": "6c6c4b8a-2280-4b4b-a51f-fcd5149f1d94",
                "Name": "slice (offset: -0.8751443)",
                "FilterType": "Slice"
              }
            ]
          },
          {
            "Id": "96ca950c-0767-4439-b57a-35384ea351a7",
            "Name": "isosurface DISPLACEMENTS/X(1) = -0.0001",
            "FilterType": "IsoSurface"
          },
          {
            "Id": "2a3c47be-2f06-48f5-adfd-30777aea092c",
            "Name": "isosurface DISPLACEMENTS/X(1) = -0.0003",
            "FilterType": "IsoSurface"
          },
          {
            "Id": "7e5f043b-9900-427e-8d38-839ff1b8e27c",
            "Name": "isosurface DISPLACEMENTS/X(1) = -0.0007",
            "FilterType": "IsoSurface"
          }
        ]
      }
    ]
}
```

LISTING A.2: Example of summary.json document.


```
{
    "Id": "5b585758-9f64-4790-a765-64709951931a",
    "Name": "master",
    "ParentId": null,
    "Filter": null,
    "Meshes": [
      {
        "Index": 1,
        "TimeSteps": [1.0, 2.0, 3.0, 4.0, 5.0, 6.0],
        "Attributes": [
          {
            "Index": 1,
            "FieldName": "ElementProperty",
            "Location": "Cells"
          }
        ]
      }
    ],
    "Fields": {
      "DISPLACEMENTS": {
        "Components": {
          "X(1)": {
            "TimeSteps": {
              "1": { "MeshIndex": 1, "DataIndex": 4 },
              "2": { "MeshIndex": 1, "DataIndex": 4 },
              "3": { "MeshIndex": 1, "DataIndex": 4 },
              "4": { "MeshIndex": 1, "DataIndex": 4 },
              "5": { "MeshIndex": 1, "DataIndex": 4 },
              "6": { "MeshIndex": 1, "DataIndex": 4 }
            }
          },
          "X(2)": {
            "TimeSteps": {
              "1": { "MeshIndex": 1, "DataIndex": 5 },
              "2": { "MeshIndex": 1, "DataIndex": 5 },
              "3": { "MeshIndex": 1, "DataIndex": 5 },
              "4": { "MeshIndex": 1, "DataIndex": 5 },
              "5": { "MeshIndex": 1, "DataIndex": 5 },
              "6": { "MeshIndex": 1, "DataIndex": 5 }
            }
          },
          "X(3)": {
            "TimeSteps": {
              "1": { "MeshIndex": 1, "DataIndex": 6 },
              "2": { "MeshIndex": 1, "DataIndex": 6 },
              "3": { "MeshIndex": 1, "DataIndex": 6 },
              "4": { "MeshIndex": 1, "DataIndex": 6 },
              "5": { "MeshIndex": 1, "DataIndex": 6 },
              "6": { "MeshIndex": 1, "DataIndex": 6 }
            }
          }
        }
      },
      "CRACK_WIDTH": { ... },
      "EXTERNAL_FORCES": { ... },
      "STRAIN": { ... },
      "STRESS": { ... }
    }
}
```


LISTING A.3: Example of mesh.json document.


```
{
    "LayerId": "5b585758-9f64-4790-a765-64709951931a",
    "Index": 1,
    "NumberOfPoints": 434,
    "NumberOfCells": 324,
    "Center": [0.6375, 0.095, 0.16],
    "Radius": 0.6719607,
    "PointCoordinates": "//9/MwAAADIK16M+//9/MwAAADJvEoM+gDSjPQAAADIK16M+//9/M1yPwj0K16M+gDSjPQAAAD...",
    "CellConnectivity": "SwAAADgAAAA0AAAASQAAAD4AAAArAAAAKAAAADwAAAA+AAAAKwAAACgAAAA8AAAAOQAAACUAAA...",
    "CellTypes": "DAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwMD..."
}
```


LISTING A.4: Example of attribute.json document.


```
{
    "LayerId": "5b585758-9f64-4790-a765-64709951931a",
    "Index": 1,
    "MeshIndex": 1,
    "FieldName": "ElementProperty",
    "Location": "Cells",
    "Compression": null,
    "Encoding": {
      "DataType": "Int32",
      "OriginalLength": 324,
      "Offset": 160,
      "Length": 164,
      "DefaultValue": "18"
    },
    "Data": "EwAAABMAAAATAAAAEwAAABMAAAATAAAAEwAAABMAAAATAAAAEwAAABMAAAATAAAAEwAAABMAAAATAAAAEwAAAB..."
}
```

LISTING A.5: Example of result.json document.


```json
{
    "LayerId": "5b585758-9f64-4790-a765-64709951931a",
    "Index": 4,
    "MeshIndex": 1,
    "FieldName": "DISPLACEMENTS",
    "ComponentName": "X(1)",
    "TimeSteps": [1.0, 2.0, 3.0, 4.0, 5.0, 6.0],
    "Location": "Points",
    "Compression": {
      "Method": "SVD",
      "Rows": 6,
      "Columns": 434,
      "Rank": 3
    },
    "Encoding": {
      "DataType": "Float64",
      "OriginalLength": 2640,
      "Offset": 0,
      "Length": 2640
    },
    "Data": "47HYqWHLML9OLoc9P8tAv5nfIqKPa1S/BoMRvASgbL+E47yyIMJ5v47IymmBdYK/ZcAwsn55IL+uuEUHN3swv1..."
}
```

Bibliography

[1] J. Kruis, T. Koudelka, and T. Krejčí. "Efficient computer implementation of coupled hydro-thermo-mechanical analysis". In: *Mathematics and Computers in Simulation* 80.8 (2010), pp. 1578–1588.

[2] T. Krejčí, T. Koudelka, and J. Kruis. "Numerical modeling of coupled hydro-thermo-mechanical behavior of concrete structures". In: *Pollack Periodica* 10.1 (2015), pp. 19–30.

[3] J. Fish and T. Belytschko. *First Course in Finite Elements*. 1st ed. Chichester: John Wiley & Sons, Ltd., 2007. ISBN: 0470035803;9780470035801;

[4] P. J. Frey and P. L. George. *Mesh Generation: Application to Finite Elements*. Hermes Science, Oxford, 2000.

[5] D. Rypl. *Sequential and parallel generation of unstructured 3D meshes*. Czech Technical University, 1998.

[6] J.-M. Bergheau and R. Fortunier. *Finite Element Simulation of Heat Transfer*. 1st ed. Iste, 2008. ISBN: 1848210531;9781848210530;

[7] T. J. Hughes, J. A. Cottrell, and Y. Bazilevs. "Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement". In: *Computer methods in applied mechanics and engineering* 194.39 (2005), pp. 4135–4195.

[8] G. Legrain. "A NURBS Enhanced eXtended Finite Element Approach for Unfitted CAD Analysis". In: *Computational Mechanics* (2013).

[9] D. Marsh. *Applied geometry for computer graphics and CAD*. Springer Science & Business Media, 2006.

[10] J. SairaBanu, R. Babu, and R. Pandey. "Parallel Implementation of Singular Value Decomposition (SVD) in Image Compression using OpenMP and Sparse Matrix Representation". In: *Indian Journal of Science and Technology* (2015).

[11] B. Li and X. Chen. "Wavelet-based numerical analysis: a review and classification". In: *Finite Elements in Analysis and Design* 81 (2014), pp. 14–31.

[12] P. Alliez and C. Gotsman. "Recent advances in compression of 3D meshes". In: *Advances in multiresolution for geometric modelling*. Springer, 2005, pp. 3–26.

[13] A. Evans et al. "3D graphics on the web: A survey". In: *Computers & Graphics* 41 (2014), pp. 43–61.

[14] H. Hoppe. "Progressive meshes". In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. ACM. 1996, pp. 99–108.

[15] H. Hoppe. "Efficient implementation of progressive meshes". In: *Computers & Graphics* 22.1 (1998), pp. 27–36.

[16] U. Güdükbay, O. Arıkan, and B. Özgüç. "Visualizer: a mesh visualization system using view-dependent refinement". In: *Computers & Graphics* 26.3 (2002), pp. 491–503.

[17] S. Valette, A. Gouaillard, and R. Prost. “Compression of 3D triangular meshes with progressive precision”. In: *Computers & Graphics* 28.1 (2004), pp. 35–42.

[18] S. Valette, R. Chaine, and R. Prost. “Progressive lossless mesh compression via incremental parametric refinement”. In: *Computer Graphics Forum*. Vol. 28. 5. Wiley Online Library. 2009, pp. 1301–1310.

[19] G. Lavoué, L. Chevalier, and F. Dupont. “Streaming compressed 3D data on the web using JavaScript and WebGL”. In: *Proceedings of the 18th international conference on 3D web technology*. ACM. 2013, pp. 19–27.

[20] P. Alliez and M. Desbrun. “Progressive compression for lossless transmission of triangle meshes”. In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM. 2001, pp. 195–202.

[21] A. Maglo et al. “Progressive compression of manifold polygon meshes”. In: *Computers & Graphics* 36.5 (2012), pp. 349–359.

[22] M. Limper et al. “Fast delivery of 3D web content: a case study”. In: *Proceedings of the 18th International Conference on 3D Web Technology*. ACM. 2013, pp. 11–17.

[23] S.-K. Ueng, Y.-J. Su, and C.-T. Chang. “LoD volume rendering of FEA data”. In: *Proceedings of the conference on Visualization’04*. IEEE Computer Society. 2004, pp. 417–424.

[24] D. T. Robaina et al. “An adaptive graph for volumetric mesh visualization”. In: *Procedia Computer Science* 1.1 (2010), pp. 1747–1755.

[25] A. Stahl, T. Kvamsdal, and C. Schellewald. “Post-processing and visualization techniques for isogeometric analysis results”. In: *Computer Methods in Applied Mechanics and Engineering* 316 (2017), pp. 880–943.

[26] A. Watson. “Image compression using the discrete cosine transform”. In: *Mathematica journal* 4.1 (1994), p. 81.

[27] C. Lui. “A Study of the JPEG-2000 Image Compression Standard”. In: (2001).

[28] *OpenCTM; the open compressed triangle mesh file format*. 2010. URL: http://openctm.sourceforge.net/.

[29] K. McHenry and P. Bajcsy. “An overview of 3d data content, file formats and viewers”. In: *National Center for Supercomputing Applications* 1205 (2008), p. 22.

[30] C. Groton et al. “The Initial Graphics Exchange Specification (IGES) Version 5. x”. In: (2006).

[31] M. J. Pratt. “Introduction to ISO 10303—the STEP standard for product data exchange”. In: *Journal of Computing and Information Science in Engineering* 1.1 (2001), pp. 102–103.

[32] *Abaqus*. URL: https://www.3ds.com/products-services/simulia/products/abaqus/.

[33] C. Geuzaine and J.-F. Remacle. “Gmsh: A 3-D finite element mesh generator with built-in pre-and post-processing facilities”. In: *International journal for numerical methods in engineering* 79.11 (2009), pp. 1309–1331.

[34] J. Ahrens, B. Geveci, and C. Law. “ParaView: An end-user tool for large-data visualization”. English. In: The Visualization Handbook. Elsevier, 2005.

[35] *VTK File Formats*. 2015. URL: https://www.vtk.org/wp-content/uploads/2015/04/file-formats.pdf.

[36] *GAMBIT file format*. URL: http://web.stanford.edu/class/me469b/handouts/gambit_write.pdf.

[37] *GiD - The personal pre and post processor*. URL: https://www.gidhome.com/.

[38] *GiD Reference Manual: Postprocess data files*. URL: http://www-opale.inrialpes.fr/Aerochina/info/en/html-version/gid_17.html.

[39] P. Ivanyi. "Finite element mesh conversion based on regular expressions". In: *Advances in Engineering Software* 51 (2012). DOI: 10.1016/j.advengsoft.2012.05.002, pp. 20–39.

[40] J. Peng, D. Liu, and K. H. Law. "An engineering data access system for a finite element program". In: *Advances in Engineering Software* 34.3 (2003), pp. 163 –181. ISSN: 0965-9978. DOI: https://doi.org/10.1016/S0965-9978(02)00129-1. URL: http://www.sciencedirect.com/science/article/pii/S0965997802001291.

[41] J. Peng and K. H. Law. "Building finite element analysis programs in distributed services environment". In: *Computers & structures* 82.22 (2004), pp. 1813 –1833.

[42] G. Heber and J. Gray. "Supporting finite element analysis with a relational database backend, part i: There is life beyond files". In: *arXiv preprint cs/0701159* (2007).

[43] G. Heber and J. Gray. "Supporting finite element analysis with a relational database backend, part ii: Database design and access". In: *arXiv preprint cs/0701160* (2007).

[44] H.-M. Chen and Y.-C. Lin. "Web-FEM: An internet-based finite-element analysis framework with 3D graphics and parallel computing environment". In: *Advances in Engineering Software* 39.1 (2008), pp. 55 –68. ISSN: 0965-9978. DOI: https://doi.org/10.1016/j.advengsoft.2006.12.001. URL: http://www.sciencedirect.com/science/article/pii/S0965997806002316.

[45] W.-C. Weng. "Web-based post-processing visualization system for finite element analysis". In: *Advances in Engineering Software* 42.6 (2011), pp. 398 –407. ISSN: 0965-9978. DOI: https://doi.org/10.1016/j.advengsoft.2011.03.003. URL: http://www.sciencedirect.com/science/article/pii/S0965997811000317.

[46] I. Ari and N. Muhtaroglu. "Design and implementation of a cloud computing service for finite element analysis". In: *Advances in Engineering Software* 60-61.Supplement C (2013). CIVIL-COMP: Parallel, Distributed, Grid and Cloud Computing, pp. 122 –135. ISSN: 0965-9978. DOI: https://doi.org/10.1016/j.advengsoft.2012.10.003. URL: http://www.sciencedirect.com/science/article/pii/S096599781200141X.

[47] C. Mouton, K. Sons, and I. Grimstead. "Collaborative visualization: current systems and future trends". In: *Proceedings of the 16th International Conference on 3D Web Technology*. ACM. 2011, pp. 101–110.

[48] A. Charland and B. Leroux. "Mobile application development: web vs. native". In: *Communications of the ACM* 54.5 (2011), pp. 49–53.

[49] S. Jourdain, U. Ayachit, and B. Geveci. "Paraviewweb, a web framework for 3d visualization and data processing". In: *IJCISIM* 3 (2011), pp. 870–7.

[50] *SimScale*. URL: https://www.simscale.com/.

[51] C. Marion and J. Jomier. "Real-time collaborative scientific WebGL visualization with WebSocket". In: *Proceedings of the 17th international conference on 3D web technology*. ACM. 2012, pp. 47–50.

[52] *WebSockets specification*. 2009. URL: http://www.w3.org/TR/websockets/.

[53] J. Behr et al. "Using images and explicit binary container for efficient and incremental delivery of declarative 3D scenes on the web". In: *Proceedings of the 17th international conference on 3D web technology*. ACM. 2012, pp. 17–25.

[54] *RedSVD (RandomizED SVD)*. URL: https://code.google.com/archive/p/redsvd/wikis/English.

[55] *ASP.NET Core*. URL: https://docs.microsoft.com/en-us/aspnet/core/.

[56] *Entity Framework Core (EF Core)*. URL: https://docs.microsoft.com/en-us/ef/core/.

[57] *Aurelia; JavaScript client framework*. URL: http://aurelia.io/.

[58] *Bootstrap; front-end web framework*. URL: https://getbootstrap.com/.

[59] *Three.js; JavaScript library for 3D graphics*. URL: https://threejs.org/.

[60] *Mono; cross-platform, open source .NET framework*. URL: http://www.mono-project.com/.

[61] *OpenTK; The Open Toolkit library, cross-platform OpenGL .NET wrapper*. URL: https://github.com/opentk/opentk/.

[62] Š. Beneš and J. Kruis. "Efficient methods to visualize finite element meshes". In: *Advances in Engineering Software* 79 (2015), pp. 81–90.

[63] B. G. Baumgart. *Winged edge polyhedron representation*. Tech. rep. Stanford University, Computer Science Department, 1972.

[64] L. De Floriani, L. Kobbelt, and E. Puppo. "A survey on data structures for level-of-detail models". In: *Advances in multiresolution for geometric modelling*. Springer, 2005, pp. 49–74.

[65] P. Shirley, M. Ashikhmin, and S. Marschner. *Fundamentals of computer graphics*. A K Peters/CRC Press, 2009.

[66] D. E. Knuth. *The Art of Computer Programming. 3: Sorting and Searching*. Addison-Wesley Professional, 1998.

[67] *GiD Reference Manual: Pre and Post Processing System for Numerical Simulations*. International Center For Numerical Methods In Engineering (CIMNE)., 2013.

[68] *VisIt User's Manual*. version 1.5. 2005. URL: https://wci.llnl.gov/codes/visit/1.5/VisItUsersManual1.5.pdf.

[69] Š. Beneš and J. Kruis. "Approximation of large data from the finite element analysis allowing fast post-processing". In: *Advances in Engineering Software* 97 (2016), pp. 17 –28. ISSN: 0965-9978. DOI: http://dx.doi.org/10.1016/j.advengsoft.2016.02.008. URL: http://www.sciencedirect.com/science/article/pii/S0965997816300497.

[70] Š. Beneš and J. Kruis. "Approximation methods for post-processing of large data from the finite element analysis". In: *Pollack Periodica* 11.3 (2016), pp. 165–176.

[71] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A multigrid tutorial*. SIAM, 2000.

[72] V. V. Shaidurov. *Multigrid methods for finite elements*. Vol. 318. Springer Science & Business Media, 2013.

[73] W. Hackbusch. *Multi-grid methods and applications*. Vol. 4. Springer Science & Business Media, 2013.

[74] H. C. Elman, D. J. Silvester, and A. J. Wathen. *Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics*. Numerical Mathematics & Scientific Computation, 2014.

[75] N. Roma and L. Sousa. "A tutorial overview on the properties of the discrete cosine transform for encoded image and video processing". In: *Signal Processing* 91.11 (2011), pp. 2443–2464.

[76] D. Reilly. "Investigating octree generation for interactive animated volume rendering". In: *University of Dublin, Trinity College* (2011).

[77] K. Baker. "Singular value decomposition tutorial". In: *The Ohio State University* 24 (2005).

[78] D. Kalman. "A Singularly Valuable Decomposition: The SVD of a Matrix". In: *The College Mathematics Journal* 27.1 (1996), pp. 2 –23. ISSN: 07468342. DOI: 10.2307/2687269. URL: http://dx.doi.org/10.2307/2687269.

[79] G. Golub and C. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, London: The Johns Hopkins University Press, 1996. ISBN: 0-8018-5414-8.

[80] E. J. Duintjer Tebbens et al. *Analýza metod pro maticové výpočty: základní metody*. Czech. 1st ed. Praha: Matfyzpress, 2012. ISBN: 9788073782016;8073782014;

[81] Š. Beneš and J. Kruis. "Singular Value Decomposition used for compression of results from the Finite Element Method". In: *Advances in Engineering Software* 117 (2018), pp. 8–17.

[82] E. Candès et al. "Robust principal component analysis?" In: *Journal of the ACM (JACM)* 58.3 (2011), p. 11.

[83] F. Woolfe et al. "A fast randomized algorithm for the approximation of matrices". In: *Applied and Computational Harmonic Analysis* 25.3 (2008), pp. 335 –366. ISSN: 1063-5203. DOI: http://dx.doi.org/10.1016/j.acha.2007.12.002. URL: http://www.sciencedirect.com/science/article/pii/S1063520307001364.

[84] P. Martinsson, V. Rokhlin, and M. Tygert. "A randomized algorithm for the decomposition of matrices". In: *Applied and Computational Harmonic Analysis* 30.1 (2011), pp. 47 –68. ISSN: 1063-5203. DOI: http://dx.doi.org/10.1016/j.acha.2010.02.003. URL: http://www.sciencedirect.com/science/article/pii/S1063520310000242.

[85] A. Szlam, Y. Kluger, and M. Tygert. "An implementation of a randomized algorithm for principal component analysis". In: *Journal of the ACM (JACM)* 1.1 (2014).

[86] N. Halko, P. Martinsson, and J. Tropp. "Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions". In: *SIAM Review* 53.2 (2011), pp. 217–288. ISSN: 0036-1445.

[87] R. Witten and E. Candes. "Randomized algorithms for low-rank matrix factorizations: sharp performance bounds". In: *Algorithmica* 72.1 (2015), pp. 264–281.

[88] J. Kruis, T. Koudelka, and T. Krejčí. "Hygro-Thermo-Mechanical Analysis of a Reactor Vessel". In: *Acta Polytechnica. Journal of Advanced Engineering* 52.6/2012 (2012). ISSN 1210-2709, e-ISSN 1805-2363, pp. 67–73.

[89] T. Koudelka, T. Krejčí, and J. Šejnoha. “Analysis of a Nuclear Power Plant Containment”. In: *Proceedings of the Twelfth International Conference on Civil, Structural and Environmental Engineering Computing*. Ed. by R. B. B.H.V. Topping L.F. Costa Neves. Paper 132, doi:10.4203/ccp.91.132. Stirlingshire, UK: Civil-Comp Press Ltd, 2009.

[90] P. Koudelka and T. Koudelka. “Risk Assessment of a Heterogeneous Stratified Rock Cliff under an Elevated Road”. In: *XIIIth Danube European Conference on Geotechnical Engineering*. Vol. 2. Ljubljana: Slovenian Geotechnical Society, 2006, pp. 56–61.

[91] J. Kruis et al. “Hygro-Thermo-Mechanical Analysis of a Nuclear Power Plant Prestressed Concrete Reactor Vessel”. In: *Proceedings of the Tenth International Conference on Civil, Structural and Environmental Engineering Computing*. Ed. by B. H. V. Topping. ISBN 1-905088-01-9. Stirling, Scotland, UK: Civil-Comp Press Ltd, 2005.