**Akademia Górniczo-Hutnicza**
**im. Stanisława Staszica w Krakowie**

Wydział Informatyki, Elektroniki i Telekomunikacji

INSTYTUT INFORMATYKI


PRACA DOKTORSKA

MARCIN SKOTNICZNY

# EFEKTYWNE ALGORYTMY DO SZYBKICH SYMULACJI ADAPTACYJNEJ METODY ELEMENTÓW SKOŃCZONYCH W DZIEDZINIE CZASO-PRZESTRZENNEJ

PROMOTOR:
prof. dr hab. Maciej Paszyński

Kraków 2023

**AGH**
**University of Science and Technology in Krakow**

Faculty of Computer Science, Electronics and Telecommunications

INSTITUTE OF COMPUTER SCIENCE


DISSERTATION FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

MARCIN SKOTNICZNY

# ALGORITHMS FOR FAST SIMULATIONS OF SPACE-TIME ADAPTIVE FINITE ELEMENT METHODS

SUPERVISOR:
prof. Maciej Paszyński, Ph.D.

Krakow 2023

**Acknowledgements**

I would like to express sincere gratitude to my supervisor, prof. Maciej Paszyński, for the invaluable guidance and support during the years of my PhD journey. Without your patience with pushing me forward, I would never get to the end of this path. I hope I wasn't too much of a burden.
I also owe a special debt of gratitude to my colleagues, Marcin Sieniek and Piotr Gurgul, for including me in their project those many years ago. Without you, I would never get interested in the domain of my PhD in the first place.
I am also greatful to my alma mater, AGH University of Science and Technology, for providing me with place to grow as an engineer, computer scientist, and a person.
Without the inspiration and support from my parents, all this would not ever happen. I will be always thankful for your encouragement with choosing the career path.
In the end, I would like to thank all my family and friends that were supporting me during those long and often difficult years. I am grateful for being surrounded with so many great people I can rely on.

## Streszczenie

Symulacje komputerowe do rozwiązywania problemów niestacjonarnych są szeroko stosowane w wielu zagadnieniach inżynieryjnych, począwszy od symulacji z dziedziny inżynierii materiałowej, geologii, poprzez symulacje wypadków samochodowych, aż do symulacji biomedycznych. Klasyczne algorytmy do rozwiązywania problemów niestacjonarnych stosują schemat jawny Eulera lub schematy niejawne typu Cranka-Nicolsona do rozwiązywania poszczególnych kroków czasowych w pętli. Możliwe jest stosowanie adaptacji w poszczególnych krokach czasowych, jak również stosowanie adaptacyjnych schematów całkowania w dziedzinie czasowej. Wszystkie te podejścia wymagają rozwiązania problemu przestrzennego na sekwencji adaptacyjnych siatek obliczeniowych w poszczególnych krokach czasowych. Podejście alternatywne polega na generacji siatki adaptacyjnej w dziedzinie czaso-przestrzennej. Podejście to umożliwia stosowanie krótszych kroków czasowych w jednym obszarze siatki przestrzennej oraz dłuższych kroków czasowych w innym obszarze przestrzennym. W niniejszej pracy doktorskiej oszacowana zaostała złożoność obliczeniowa solwerów na wielowymiaorowych siatkach adaptacyjnych. W szczególności porównano koszty obliczeniowe rozwiązywania problemow niestacjonarnych rozwiązywanych na adaptacyjnych siatkach czaso-przestrzennych z kosztami obliczenowymi solwerów rozwiązujących problem niestacjonarny za pomocą schematu iteracyjnego na sekwencji adaptacyjnych siatek obliczeniowych w dziedzinie przestrzennej. Zaproponowano również algorytm solwera dokładnego o quasi-optymalnej złożoności obliczeniowej na wielowymiarowych siatkach obliczeniowych.

# Abstract

Computer simulations for solving non-stationary problems are widely used in a wide range of engineering problems, from simulations in materials engineering, geology, through simulations of automobile accidents to biomedical simulations. Classical algorithms for solving non-stationary problems use an explicit Euler scheme or implicit Crank-Nicolson type schemes to solve individual time steps in a loop. It is possible to use adaptation in individual time steps, as well as to use adaptive integration schemes in the time domain. All these approaches require solving the spatial problem on a sequence of adaptive computational grids at individual time steps. An alternative approach is to generate an adaptive mesh in the time-space domain. This approach allows the use of shorter time steps in one spatial grid area and longer time steps in another spatial area. In this dissertation, the computational complexity of solvers on multidimensional adaptive grids was estimated. In particular, the computational cost of solving non-stationary problems solved on adaptive time-space grids was compared with the computational cost of solvers solving a non-stationary problem using an iterative scheme on a sequence of adaptive grids in the spatial domain. An exact solver algorithm with quasi-optimal computational complexity on multidimensional computational grids is also proposed.

# Table of contents

Chapter 1

# Introduction

The Finite Element Method FEM is one of the most basic methods used in simulations and other calculations in a broad spectrum of problems in many sciences and industries. Due to the vast applicability of the Finite Element Method combined with its high computational cost, a significant part of world's computational power is being spent on running its implementations. As such, any improvements in the speed of algorithms used for Finite Element Method solvers can not only bring significant savings to the world's economy but also expand the scope of problems that can be solved using today's computational infrastructure. First step to gauge any such improvements would be to measure their impact on the cost of the computation. Such measurement is most often done experimentally, however it can also be done by the calculation of the theoretical time complexity of the improved algorithm.

In the finite element community, there are two common approaches to deal with time-dependent Partial Differential Equations (PDEs). One of the most popular is the time–marching FE scheme [48], where the time-dependency is discretized with a finite difference-type scheme and a standard FE approximation in space is considered. Recently, the study of FE discretizations of full space-time formulations has become popular in the community [35]. An advantage of time stepping schemes is that their oblivious nature allows for optimal storage requirements if one is only interested in the final state. As soon as one considers problems where the entire history of the evolution problem is of interest, this advantage becomes less beneficial. Furthermore, the flexibility of time stepping schemes with respect to space–time local mesh refinement is very limited, such that possible local space–time singularities prevent the optimal usage of computational resources. When it comes to the development of parallel solvers, the sequentiality of time stepping schemes imposes severe difficulties. For this and various other reasons, simultaneous space–time finite element discretizations, where time is treated as just another spatial variable, have been proposed in recent years. They all rely basically on the standard well-posed variational space–time formulation of parabolic equations [35]. Time–marching schemes must remain with the sequential nature of the refinement, implying a possible increment in the storage requirements depending on the nature of the singularity. Finally, space-time formulations allow for parallel static condensation, thus possibly being faster than time–marching schemes when a sufficient amount of computational power is available. Even though there is still extensive research on the space-time formulations and the time–marching schemes, to the author's knowledge, there are no works related to theoretical aspects of the computational complexity of both methodologies available in the literature.

## 1.1. Motivation

Computational complexity and, especially, time complexity is one of the most fundamental concepts of theoretical computer science, and was first defined in 1965 by Hartmanis and Stearns [29]. The time complexity of

timewise-optimal algorithms for certain classes of meshes, especially regular meshes, is well known, however, as of writing of this book, there is no known general formula or method to calculate an optimal time complexity of direct solver algorithms for Finite Element Method (FEM) for any mesh. Considering that the whole spectrum of various problems that can be solved using FEM is very broad, it is neither hard to imagine that such a formula or a method might not be discovered any time soon, nor that such a formula or a method would be simple and practical enough to be applicable in real world problems.

Because of the complexity of the aforementioned problem, in this book we focus on a specific class of problems, namely, problems in which there exists a singularity of some sort. We propose an optimal, or close to an optimal, algorithm for solving a particular class of problems, that is problems with singularities, using the $h$-adaptive Finite Element Method with fixed global polynomial order of approximation $p$, usually $p = 2$, since quadratic polynomials on adaptive grids are considered a good approximation to a wide class of engineering problems [51]. We start from the design and analysis of a class of algorithms generating quasi-optimal order of elimination of the variables in the sparse matrix resulting from FEM discretization. Based on the proposed algorithm, we derive a general formula for computational cost of multi-frontal solver executed on computational grids with singularities.

Of particular interest to the research community are computations of time-dependent problems. In this work, we study complexity aspects of adaptive time–marching schemes and adaptive space-time formulations. We explore the complexity of iterative and direct solvers when considering hypercubic $h$–refined grids towards singularities. For this purpose, we propose a general method-independent strategy simulating the best-scenario possibility for each case. We proceed as follows: To simulate an $h$–adaptive space–time procedure, we start from a uniform $d$–dimensional mesh, where $d-1$ dimensions correspond to the spatial discretization and the last dimension to the time discretization, and we consider regular hypercubic geometrical refinements towards singularities. To simulate a time–marching scheme, we consider a sequence of $d-1$ dimensional meshes obtained from considering time cuts of the space–time grids, with a fixed time step of length equal to the smallest element dimension in the time axis direction. To motivate these assumptions, the reader can consider a parabolic problem, for instance, the heat equation, as the PDE of interest. Therefore, the assumption for the space–time formulation is natural, assuming that information of the singularity is available, for instance, an a posteriori error estimator. In contrast, the assumption for the time–marching scheme will be the ideal scenario for the explicit-in-time Euler scheme ensuring that the Courant–Friedrichs–Lewy condition (CFL condition) [38] is satisfied, and also information on the singularity is provided.

To derive the estimations, we start by considering refinements toward a space-time "edge", resulting from a point traveling through space and time. This space-time refinement pattern corresponds to a sequence of spatial meshes refined towards "points" located on the space-time edge at particular time moments. Next, we consider the refinement towards a space-time "face", resulting from an edge traveling through space and time. Again, this space-time refinement pattern corresponds to a sequence of spatial meshes refined towards "edges" located on the space-time face at a particular moment. We also consider the refinement towards a space-time "hyperface" resulting from a face traveling through space and time. This space-time refinement pattern corresponds to a sequence of spatial meshes refined towards "faces" located on the space-time hyperface at a particular time instant. We compare the computational complexity of the iterative solver executed over the $d$ dimensional space-time domain to the computational complexity of the iterative solver performed multiple times over the $d$-1 dimensional meshes during the time–marching scheme. We compare the computational complexity of the direct solver run over the $d$ dimensional space-time domain to the computational complexity of the direct solver executed multiple times over $d$-1 dimensional meshes within the time–marching scheme.

## 1.2. Main thesis

The main thesis of this work may be summarized as follows:

> There exist direct solver algorithms that, when used for the finite element method over meshes with hierarchical refinement over a singularity, will have a computational complexity of no more than $\mathcal{O}(N^{\max(3\frac{(q-1)}{q},1)})$, where $N$ is the number of variables of the problem after the refinement step, and $q$ is the dimensionality of the singularity. In particular, the time complexity for such meshes does not depend on the dimensionality of the space the mesh is built on. These algorithms can be applied for solution of space-time problems, when using either time marching scheme or space-time formulation.

## 1.3. Open problems

1. There is no general formula for calculating the computational time complexity for hierarchically adapted multidimensional grids with hierarchical basis functions.
2. There are no estimations of computational time complexities of multi-frontal direct solvers for space-time finite element methods on hierarchically adapted space-time grids.
3. There are no estimations of computational time complexities of iterative solvers for space-time finite element methods on hierarchically adapted space-time grids.
4. There are no estimations of computational time complexities of multi-frontal direct solvers for time-marching schemes on hierarchically adapted grids employed in every time step.
5. There are no estimations of computational time complexities of iterative solvers for time-marching schemes on hierarchically adapted grids employed in every time step.
6. There is no algorithm for generation of quasi-optimal element partition trees for hierarchically adapted grids utilized in space-time finite element method computations.
7. There is no algorithm for translation of element partition trees generated for space-time finite element method computations into efficient orderings in multi-frontal direct solvers.

## 1.4. Main scientific results

The main scientific results of this dissertation are the following:

1. We define a set of constraints that a computational Finite Element Method meshes that will result in quasi-optimal computational complexities of direct solvers working on them. This applies to stationary FEM, time-marching scheme and space-time formulations.
2. We prove that there exist direct solver algorithms that work on regular meshes fulfilling the constraints referred in point 1 with a time complexity of $\mathcal{O}(N^{\max(3\frac{(q-1)}{q},1)})$, where $N$ is the amount of variables and $q$ is the dimensionality of the mesh, as long as the number of variables is linearly dependent on the number of elements. Again, this applies to stationary FEM, time-marching scheme and space-time formulations.
3. We prove that for hierarchically adapted meshes fulfilling the constraints referred in point 1 refined towards $q$-dimensional singularities there exist algorithms that belong to the same time complexity class as the time complexity of algorithms referred to in point 2 for regular meshes constructed in the space of dimensionality $q$.

4. We derive a general formula for the number of unknowns for the mesh of arbitrary dimension $d$, refined towards a singularity of dimension $q$. This applies for point singularity ($q = 0$), edge singularity ($q = 1$), face singularity ($q = 2$), hyperface singularity ($q = 3$), focusing on the computational complexities of space-time formulations and time–marching schemes. In our general estimates, we do not consider the polynomial order of approximation $p$, and we assume that this is a constant in our formulas. Additionally, the singularities in our case may have an arbitrary shape, therefore, we only fix the dimension of the space $d$ and the dimension of the singularity $q$.
5. We estimate the time complexity of finite element method simulations performed on adaptive $d$–dimensional space–time meshes. We also assess the time complexity of finite element method simulations performed on a sequence of adaptive $(d-1)$–dimensional meshes resulting from the corresponding time–marching scheme. In particular, we estimate $N$ = the number of unknowns for the whole adaptive space–time formulation, $n$= the number of unknowns from a mesh from a sequence of adaptive meshes resulting from the time–marching scheme.
6. We estimate the time complexity of the iterative solvers executed on both time-marching scheme and space-time formulation. Thus, we have the lower bounds (expressed by the number of unknowns) and upper bounds (described by the time complexity of the sparse direct solver).
7. We compare the computational complexities of direct and iterative solvers for transient simulations, executed on $d$-dimensional space-time refined mesh versus a sequence of $d-1$ dimensional meshes solved by time-marching scheme.

## 1.5. Structure of this book

This book is divided into 9 chapters, starting with the introduction contained in the current Chapter 1.

Chapter 2 covers the current state of the art and contains a brief introduction to the concepts used later in the book. First of all, it defines the concept of computational complexity, then introduces the basics of linear equation system solvers, finite element method, adaptive algorithms, and the formal definition of the finite element, in the context of $h$-adaptive and $hp$-adaptive computational meshes. Finally, the Chapter concludes with a brief overview of the time complexities of direct solver-based Finite Element Method algorithms known so far, and some references to practical applications of the space-time finite element method.

Chapter 3 defines the basic tools that will be used in the proofs in later chapters. First of all, it defines formally how the grids used in the proofs are constructed, secondly, it lays out the basic algorithms which the time complexity will be calculated of, and, finally, concludes with the definition of the Element Partition Tree (EPT), and it explains how the EPT can define the ordering for permutation of the matrix of system of linear equations generated by FEM that reduces the computational complexity of the solver.

Later chapters cover the proofs of time complexities themselves. Chapter 4 covers a relatively simple proof of the time complexity for a simple case of problems that contain a point singularity. Chapter 5 introduces a more complex proof for the case of $h$-adaptive meshes with singularities of any integer dimensions embedded in space of an arbitrary dimensionality. Finally, Chapter 6 extends the ideas from the previous chapters to conclude with an universal proof for any well-formed type of Finite Element Method with singularities of any integer dimensionality.

Chapter 7 presents comparison of computational costs of direct and iterative solvers on model adaptive grids for space-time formulations and time-marching schemes. Next Chapter 8 presents numerical experiments illustrating the theoretical findings for space-time formulations and time-marching grids.

We finish the book in the final *Conclusions and further work* Chapter 9.

Chapter 2

# Background

## 2.1. Computational complexity

Computational complexity is one of the most fundamental concepts in the theoretical computer science. It has been first defined by Hartmanis and Stearns in 1965[29]. The computational complexity is a property of an algorithm. It is a measure of how much running of the algorithm will cost, usually stated as a function of the input size. Most commonly, two types of computational complexities are used: time complexity, that is the running time of the algorithm, stated in either seconds of its running time, or, more commonly, in the number of some elementary operations performed during its computation; and space complexity, that is the amount of extra memory that given algorithm will use during its runtime, either in terms of bytes, or, for example, length of Turing's machine tape. In this book, we focus on the time complexity, as it is the more limiting factor for the algorithms analysed.

Usually, the time complexity is stated as either the worst-case complexity, i.e. the maximum of all running times for all inputs of given size; or as expected (or average) complexity, i.e. the average running time for those inputs. In this book, unless specified otherwise, we will analyze the worst-case complexity, however for all the analyzed problems it will be virtually synonymous with the expected complexity. Worst-case complexity is however difficult (if not impossible) to calculate exactly and, because of that, in practice only its asymptotic behavior is analyzed and the time complexity is expressed using big-$\mathcal{O}$ notation.

### 2.1.1. Big-$\mathcal{O}$ notation

The big-$\mathcal{O}$ notation is a mathematical notation used to describe the behavior of a function as its argument approaches a value or infinity. In this book, we will only use this notation when describing how a function behaves when the argument or arguments grow towards infinity. Using the notation, we will denote a class of functions that asymptotically behave similarly to function $g(x)$ when $x$ tends to infinity as $\mathcal{O}(g(x))$. In particular:

$$f(x) \in \mathcal{O}(g(x)) \iff \exists x_0 \in \mathbb{N}, k > 0 : \forall x \geq x_0 : |f(x)| \leq k \cdot g(x) \tag{2.1}$$

The same definition can also be also stated in terms of limits:

$$f(x) \in \mathcal{O}(g(x)) \iff \limsup_{x \to \infty} \frac{|f(x)|}{g(n)} < \infty \tag{2.2}$$

By a slight abuse of the notation, we will write simply $f(x) = \mathcal{O}(g(x))$ instead of $f(x) \in \mathcal{O}(g(x))$, as it is customary.

In other words, $f(x) = \mathcal{O}(g(x))$ means that there exists some argument $x_0$ and some positive constant $k$ such that for each argument $x$ no lesser than $x_0$ the value of $f(x)$ does not exceed $g(x)$ multipled by $k$. If $f(x) = \mathcal{O}(g(x))$, we can also say that $f(x)$ is bounded above by $g(x)$.

### 2.1.2. Big-$\Omega$ and big-$\Theta$ notations

Apart from big-$\mathcal{O}$, we will also use two other similar notations: big-$\Omega$ and big-$\Theta$. The big-$\Omega$ can be defined by the following equation:

$$\begin{aligned} f(x) \in \Omega(g(x)) &\iff \exists x_0 \in \mathbb{N}, k > 0 : \forall x \geq x_0 : f(x) \geq k \cdot g(x) \\ &\iff \liminf_{x \to \infty} \frac{f(x)}{g(n)} > 0 \end{aligned} \tag{2.3}$$

In other words, big-$\Omega$ is kind of an opposite of big-$\mathcal{O}$ notation: $f(x) = \Omega(g(x))$ means that there exists some argument $x_0$ and some positive constant $k$ such that for each argument $x$ no lesser than $x_0$ the value of $f(x)$ is not smaller than $g(x)$ multipled by $k$. We can also say that this means that $f(x)$ is bounded below by $g(x)$.

Big-$\Omega$ notation is not as useful as big-$\mathcal{O}$, as we will rarely consider the minimal running time of an algorithm, however it is used to define big-$\Theta$ notation:

$$f(x) = \Theta(g(x)) \iff f(x) = \Omega(g(x)) \wedge f(x) = \mathcal{O}(g(x)) \tag{2.4}$$

To put it in words, $f(x) = \Theta(g(x))$ means that $f(x)$ is bounded below and above by $g(x)$. If $f(x)$ and $g(x)$ have positive values for all arguments (and this is true for computational complexity functions), this also means that $f(x) = \Theta(g(x)) \iff g(x) = \Theta(f(x))$.

## 2.2. Linear equation system solvers

There are two main groups of computational methods that are used to solve systems of linear equations. Firstly, there are direct methods, such as Gauss elimination, and there are iterative methods, for example: Jacobi method, Gauss–Seidel method, successive over-relaxation method, etc. In this book we analyze only direct methods, but in this section we will briefly cover the differences between those two groups of algorithms.

The direct solvers work similarly to the algorithms usually teached in the classes of mathematics, that is by repetetive simplification of the equation system by adding or subtracting some equation from another to reduce the number of terms. On the other hand, the iterative methods work by generating a sequence of approximations of the solution using some iterative mathematical operation that depends on the exact method selected. Direct solvers provide exact results and only source of error is the rounding errors generated by floating point operations. Iterative methods have inherent approximation error, but generally tend to be faster than direct solvers.

Iterative methods do not guarantee convergence for every matrix, thus different solvers or preconditioners will be required for different applications. For example, see [8, 9, 10, 11, 25] for isogeometric Finite Element Method, [37] for elasticity simulations, [31] for simulations of the propagation of electromagnetic waves or [6] for simulations of fluid dynamics.

Direct solvers can be used effectively for problems with one matrix and multiple right-hand sides. After performing the costly LU factorization, solving for single right-hand side requires solving a simple triangular system. Difficult problems, such as the computational electromagnetics problems solved on adaptive grids usually requires applications of direct solvers, since it is very difficult to obtain convergence of iterative solvers [13, 14].

The most known direct algorithm is the Gaussian elimination that works most effectively on dense matrices, having a time complexity of $\mathcal{O}(n^3)$. For sparse matrices there are faster implementations of direct solvers, for example the frontal [32, 20] and multi-frontal solvers [21, 26]). In this book we will estimate the computational complexity of multi-frontal solvers, in particular for adaptive space-time computational problems.

## 2.3. Finite Element Method

The *Finite Element Method* (FEM) is one of the most commonly used tools to solve mathematical problems in engineering and physics that cannot be solved analytically. In particular, Finite Element Method is used to solve numerically partial differential equations. To obtain a solution, the domain, which the problem is defined over, is split into smaller and simpler parts called *finite elements*. Within each finite element the unknown function can be well approximated by another function that is simple to analyze. We employ $d$ dimensional hypercube elements (rectangles in 2D, hexahedral in 3D). We utilize hierarchical basis functions of fixed order $p$. We define one-dimensional hierarchical shape-functions, which will be used later to define the two-, three-, and higher-dimensional basis functions by their tensor products and by gluing together the shape functions from adjacent elements. For more details, we refer to [16].

$$\hat{\chi}_1(\xi) = 1-\xi \quad \hat{\chi}_2(\xi) = \xi \quad \hat{\chi}_l(\xi) = (1-\xi)\xi(2\xi-1)^{l-3} \quad l = 3,4,...,p$$

These shape functions are used first to define shape functions over two-, three-dimensional and higher-dimensional finite elements.

Below, we define formally the 3D hexahedral finite element with uniform polynomial order of approximation $p$, and the computational mesh constructed out of hexahedral finite elements

**Definition 2.3.1.** *The reference 3D h finite element with fixed order $p$ is a triple $(\hat{K}, X(\hat{K}), \Pi_p)$*

1. *Geometry $\hat{K} = [0,1]^3$*
2. *Selection of nodes. There are eight vertex nodes $\hat{a}_1$, ..., $\hat{a}_8$, twelve edge nodes $\hat{a}_9$, ... , $\hat{a}_{20}$, six face nodes $\hat{a}_{21}, ..., \hat{a}_{26}$, and one interior node $\hat{a}_{27}$*
3. *Definition of element shape functions $X(\hat{K})$*

$$X(\hat{K}) = span\left\{\hat{\phi}_j \in \mathcal{Q}^{(p,p,p)}(\hat{K})\right\}_{j=1,...,(p+1)^3}$$

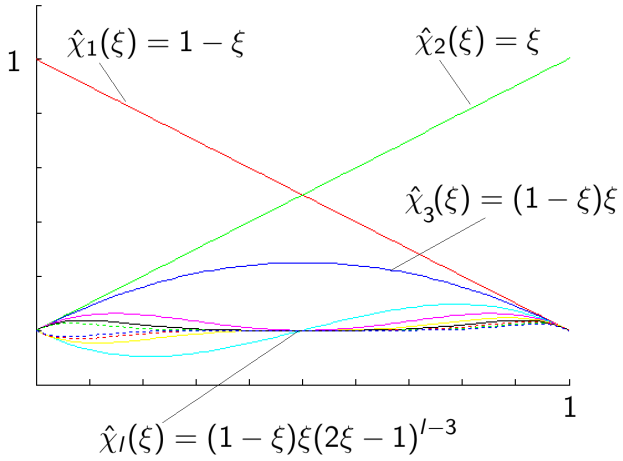



Figure 2.1: 1D hierarchical shape functions

*where $\mathcal{Q}^{(p,p,p)}$ are polynomials of order $p$ with respect to spatial variables*

4. *Definition of the projection based interpolation operator $\Pi_p : H^1(0,1) \to X(\hat{K})$, given a function $u \in H^1(\hat{K})$, it computes its projection-based interpolant $\Pi_p u \in X(\hat{K})$*

The shape functions are constructed as tensor products of 1D hierarchical shape functions The *vertex shape functions* of the 3D $h$ finite element with fixed order $p$ are:

$$\begin{aligned}
\hat{\phi}_1(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_1(\xi_1)\hat{\chi}_1(\xi_2)\hat{\chi}_1(\xi_3)\\
\hat{\phi}_2(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_2(\xi_1)\hat{\chi}_1(\xi_2)\hat{\chi}_1(\xi_3)\\
\hat{\phi}_3(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_2(\xi_1)\hat{\chi}_2(\xi_2)\hat{\chi}_1(\xi_3)\\
\hat{\phi}_4(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_1(\xi_1)\hat{\chi}_2(\xi_2)\hat{\chi}_1(\xi_3)\\
\hat{\phi}_5(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_1(\xi_1)\hat{\chi}_1(\xi_2)\hat{\chi}_2(\xi_3)\\
\hat{\phi}_6(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_2(\xi_1)\hat{\chi}_1(\xi_2)\hat{\chi}_2(\xi_3)\\
\hat{\phi}_7(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_2(\xi_1)\hat{\chi}_2(\xi_2)\hat{\chi}_2(\xi_3)\\
\hat{\phi}_8(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_1(\xi_1)\hat{\chi}_2(\xi_2)\hat{\chi}_2(\xi_3)
\end{aligned}$$

The *edge shape functions* for 3D $h$ finite element with fixed order $p$ are defined as:

$$\begin{aligned}
\hat{\phi}_{9,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_{2+j}(\hat{\xi}_1)\hat{\chi}_1(\hat{\xi}_2)\hat{\chi}_1(\xi_3), & \hat{\phi}_{10,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_2(\hat{\xi}_1)\hat{\chi}_{2+j}(\hat{\xi}_2)\hat{\chi}_1(\xi_3),\\
\hat{\phi}_{11,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_{2+j}(\hat{\xi}_1)\hat{\chi}_2(\hat{\xi}_2)\hat{\chi}_1(\xi_3), & \hat{\phi}_{12,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_1(\hat{\xi}_1)\hat{\chi}_{2+j}(\hat{\xi}_2)\hat{\chi}_1(\xi_3),\\
\hat{\phi}_{13,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_{2+j}(\hat{\xi}_1)\hat{\chi}_1(\hat{\xi}_2)\hat{\chi}_2(\xi_3), & \hat{\phi}_{14,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_2(\hat{\xi}_1)\hat{\chi}_{2+j}(\hat{\xi}_2)\hat{\chi}_2(\xi_3),\\
\hat{\phi}_{15,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_{2+j}(\hat{\xi}_1)\hat{\chi}_2(\hat{\xi}_2)\hat{\chi}_2(\xi_3), & \hat{\phi}_{16,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_1(\hat{\xi}_1)\hat{\chi}_{2+j}(\hat{\xi}_2)\hat{\chi}_2(\xi_3),\\
\hat{\phi}_{17,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_1(\hat{\xi}_1)\hat{\chi}_1(\hat{\xi}_2)\hat{\chi}_{2+j}(\xi_3), & \hat{\phi}_{18,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_1(\hat{\xi}_1)\hat{\chi}_2(\hat{\xi}_2)\hat{\chi}_{2+j}(\xi_3),\\
\hat{\phi}_{19,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_2(\hat{\xi}_1)\hat{\chi}_1(\hat{\xi}_2)\hat{\chi}_{2+j}(\xi_3), & \hat{\phi}_{20,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_2(\hat{\xi}_1)\hat{\chi}_2(\hat{\xi}_2)\hat{\chi}_{2+j}(\xi_3),
\end{aligned}$$

where $j = 1,\ldots,p-1$. Here $p$ denotes the polynomial order of approximation over the 3D hexahedral finite element. The edge shape functions are constructed by multiplying one higher-order shape function and $d$-1 linear shape functions over a finite element. The edge shape functions are hierarchical. Namely, they define $(p-1)$ shape functions for the approximation of order $p$.

The *face shape functions* for 3D $h$ finite element with fixed order $p$ are defined as

$$\begin{aligned}
\hat{\phi}_{21,i,j} &= \hat{\xi}_{2+i}(\chi_1)\hat{\xi}_{2+j}(\chi_2)\hat{\xi}_1(\chi_3)\hat{\phi}_{22,i,j} = \hat{\xi}_{2+i}(\chi_1)\hat{\xi}_{2+j}(\chi_2)\hat{\xi}_2(\chi_3)\\
\hat{\phi}_{23,i,j} &= \hat{\xi}_{2+i}(\chi_1)\hat{\xi}_1(\chi_2)\hat{\xi}_{2+j}(\chi_3)\hat{\phi}_{24,i,j} = \hat{\xi}_{2+i}(\chi_1)\hat{\xi}_2(\chi_2)\hat{\xi}_{2+j}(\chi_3)\\
\hat{\phi}_{25,i,j} &= \hat{\xi}_1(\chi_1)\hat{\xi}_{2+i}(\chi_2)\hat{\xi}_{2+j}(\chi_3)\hat{\phi}_{26,i,j} = \hat{\xi}_2(\chi_1)\hat{\xi}_{2+i}(\chi_2)\hat{\xi}_{2+j}(\chi_3)
\end{aligned}$$

where $i = 1,\ldots,p-1, j = 1,\ldots,p-1$. The face shape functions are constructed by the tensor product of $d$ higher-order shape functions. The interior shape functions are hierarchical. Namely, they define $(p-1)^d$ shape functions for the approximation of order $p$.

The *interior shape functions* of 3D $h$ finite element with fixed order $p$ are defined as

$$\begin{aligned}
\hat{\phi}_{27,i,j,k} &= \hat{\xi}_{2+j}(\chi_1)\hat{\xi}_{2+j}(\chi_2)\hat{\xi}_{3+j}(\chi_3)\\
i = 1,\ldots,p-1, j = 1,\ldots,p-1, k = 1,\ldots,p-1
\end{aligned}$$

**Definition 2.3.2.** *The computational mesh is obtained by partitioning the domain $\Omega$ into a finite set $(K, X(K), \Pi_p) \in T_{hp}$ of h finite elements and selecting arbitrary polynomial orders of approximation.*

The interior shape functions are constructed by multiplying $d$ higher-order shape functions over a finite element.

This tensor product construction can be generalized to arbitrary $d$-dimensional finite elements. For example, we consider shape functions over vertices, edges, and interiors in three dimensions. In four dimensions, we consider shape functions over vertices, edges, faces, hyperfaces, and interiors.

### 2.3.1. Adaptive finite element methods

To maximize the quality of the approximation we can reduce the size of the finite elements, or in other words, to refine the mesh. This operation will however increase the number of finite elements and increase the computational cost of the calculations. The simplest case of finite element mesh would a so called regular mesh, where each finite element is equal in size (and most often rectangular or cubic). To refine a regular mesh, we can decrease the size of each element. This will result however in a significant growth of the number of elements – dividing the side of the elements by 2 will increase the total number of elements by $2^D$, where $D$ is the number of dimensions of the problem space. Often, a refinement is required only for a small area of the domain, which would mean that the increase of number of elements would need to be unnecessarily large.

To approach this problem, in this book we analyze direct solvers over so called hierarchically refined meshes, where, instead of refinining the mesh uniformly, the mesh is refined by iteratively finding finite elements over which the approximation error is deemed to be too big and splitting each such one separately into a set of smaller elements. This approach will avoid refinement in the areas of the domain that do not require it and in turn will not create new elements needlessly.

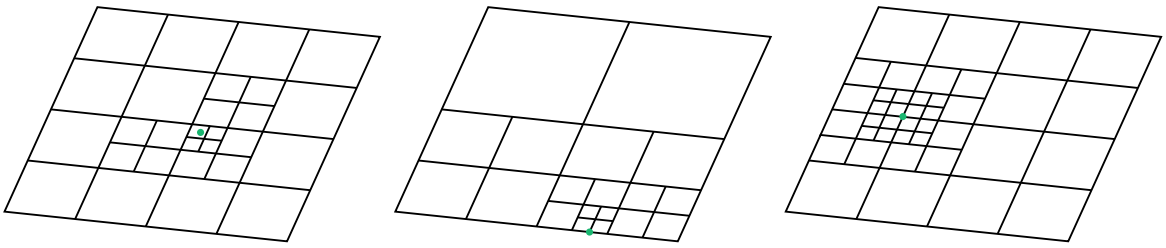


Figure 2.2: Examples of adaptive grids refinements towards different point singularity in 2D.

### 2.3.2. Adaptation towards a singularity

A common source of difficulties with using hierarchically refined meshes are singularities. A singularity, as understood in this book, is a subset of the space over which the Finite Element Method adaptation will never achive the required approximation quality, despite refining the containing elements repeatedly – or, in other words, the adaptation will never converge. Singularities of the mesh are usually caused by singularities in the analyzed function or other similar mathematical features of the problem.

The existence of a singularity might cause the mesh to grow infinitely large, so an arbitrary limit of refinement rounds is necessary – we will call it the refinement level of the mesh. As the refinement level of the mesh grows, the number of variables will grow as well, and so will the computational cost of the solver. In this book we analyze how singularities and their level of refinement can impact the computational cost of the whole algorithm. The

analysis is focused on hierarchical grids, but the findings can also apply to a broader class of meshes, as shown in later chapters.

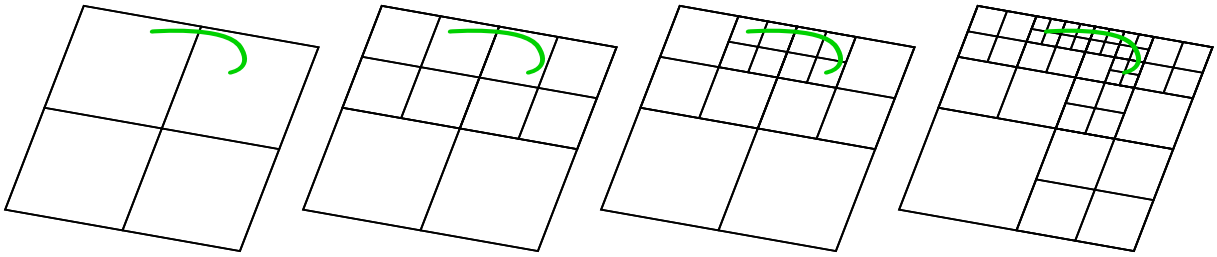


Figure 2.3: A sequence of refinements towards a line singularity on two-dimensional mesh.

### 2.3.3. $h$-adaptive and $hp$-adaptive grids

$hp$-FEM is one of common types of adaptive Finite Element Methods that employs both hierarchical refinements ($h$-refinements) and various polynomial degree refinements ($p$-refinements). It has been first defined in [15, 16, 17] and it has been shown that when using a suitable combination of both types of refinements, the method converges exponentially fast. In this method, the approximation of the solution is created from a set of polynomial basis functions that are defined over features of the mesh, in particular vertices, edges and faces, and in three dimensions also volumes, of the elements.

In particular, the definition of the finite element in $hp$ adaptive finite element method allows for varying the polynomial orders of approximation on finite element edges, faces and interios:

**Definition 2.3.3.** *The reference 3D hp finite element is a triple* $(\hat{K}, X(\hat{K}), \Pi_p)$

1. *Geometry* $\hat{K} = [0,1]^3$
2. *Selection of nodes. There are eight vertex nodes* $\hat{a}_1$, ..., $\hat{a}_8$, *twelve edge nodes* $\hat{a}_9$, ... , $\hat{a}_{20}$, *six face nodes* $\hat{a}_{21}, ..., \hat{a}_{26}$, *and one interior node* $\hat{a}_{27}$
3. *Definition of element shape functions* $X(\hat{K})$

$$X(\hat{K}) = span\Big\{\hat{\phi}_j \in \mathcal{Q}^{(p_{x_1},p_{x_2},p_{x_3})}(\hat{K})\Big\}_{j=1,...,(p_{x_1}+1)(p_{x_2}+1)(p_{x_3}+1)}$$

   *where* $\mathcal{Q}^{(p_{x_1},p_{x_2},p_{x_3})}$ *are polynomials of order* $p_{x_i}$
   *with respect to* $x_i$, $i = 1,2,3$
4. *Definition of the projection based interpolation operator* $\Pi_p : H^1(0,1) \to X(\hat{K})$, *given a function* $u \in H^1(\hat{K})$, *it computes its projection-based interpolant* $\Pi_p u \in X(\hat{K})$

The shape functions, similarly to the $h$ finite element, are constructed as tensor products of 1D hierarchical shape functions

$$\hat{\chi}_1(\xi) = 1 - \xi \quad \hat{\chi}_2(\xi) = \xi \quad \hat{\chi}_l(\xi) = (1-\xi)\xi(2\xi-1)^{l-3} \quad l = 3,4,...,p$$

The *vertex shape functions* in the $hp$ finite element are identical to the one of the $h$ finite element:

$$\hat{\phi}_1(\xi_1,\xi_2,\xi_3) = \hat{\chi}_1(\xi_1)\hat{\chi}_1(\xi_2)\hat{\chi}_1(\xi_3)$$
$$\hat{\phi}_2(\xi_1,\xi_2,\xi_3) = \hat{\chi}_2(\xi_1)\hat{\chi}_1(\xi_2)\hat{\chi}_1(\xi_3)$$
$$\hat{\phi}_3(\xi_1,\xi_2,\xi_3) = \hat{\chi}_2(\xi_1)\hat{\chi}_2(\xi_2)\hat{\chi}_1(\xi_3)$$

$$\hat{\phi}_4(\xi_1,\xi_2,\xi_3) = \hat{\chi}_1(\xi_1)\hat{\chi}_2(\xi_2)\hat{\chi}_1(\xi_3)$$
$$\hat{\phi}_5(\xi_1,\xi_2,\xi_3) = \hat{\chi}_1(\xi_1)\hat{\chi}_1(\xi_2)\hat{\chi}_2(\xi_3)$$
$$\hat{\phi}_6(\xi_1,\xi_2,\xi_3) = \hat{\chi}_2(\xi_1)\hat{\chi}_1(\xi_2)\hat{\chi}_2(\xi_3)$$
$$\hat{\phi}_7(\xi_1,\xi_2,\xi_3) = \hat{\chi}_2(\xi_1)\hat{\chi}_2(\xi_2)\hat{\chi}_2(\xi_3)$$
$$\hat{\phi}_8(\xi_1,\xi_2,\xi_3) = \hat{\chi}_1(\xi_1)\hat{\chi}_2(\xi_2)\hat{\chi}_2(\xi_3)$$

The *edge shape functions* for $hp$ finite element are different, they allow for local changing of the polynomial orders of approximation:

$$\hat{\phi}_{9,j}(\xi_1,\xi_2,\xi_3) = \hat{\chi}_{2+j}(\hat{\xi}_1)\hat{\chi}_1(\hat{\xi}_2)\hat{\chi}_1(\xi_3), j = 1,\dots,p_1-1$$
$$\hat{\phi}_{10,j}(\xi_1,\xi_2,\xi_3) = \hat{\chi}_2(\hat{\xi}_1)\hat{\chi}_{2+j}(\hat{\xi}_2)\hat{\chi}_1(\xi_3), j = 1,\dots,p_2-1$$
$$\hat{\phi}_{11,j}(\xi_1,\xi_2,\xi_3) = \hat{\chi}_{2+j}(\hat{\xi}_1)\hat{\chi}_2(\hat{\xi}_2)\hat{\chi}_1(\xi_3), j = 1,\dots,p_3-1$$
$$\hat{\phi}_{12,j}(\xi_1,\xi_2,\xi_3) = \hat{\chi}_1(\hat{\xi}_1)\hat{\chi}_{2+j}(\hat{\xi}_2)\hat{\chi}_1(\xi_3), j = 1,\dots,p_4-1$$
$$\hat{\phi}_{13,j}(\xi_1,\xi_2,\xi_3) = \hat{\chi}_{2+j}(\hat{\xi}_1)\hat{\chi}_1(\hat{\xi}_2)\hat{\chi}_2(\xi_3), j = 1,\dots,p_5-1$$
$$\hat{\phi}_{14,j}(\xi_1,\xi_2,\xi_3) = \hat{\chi}_2(\hat{\xi}_1)\hat{\chi}_{2+j}(\hat{\xi}_2)\hat{\chi}_2(\xi_3), j = 1,\dots,p_6-1$$
$$\hat{\phi}_{15,j}(\xi_1,\xi_2,\xi_3) = \hat{\chi}_{2+j}(\hat{\xi}_1)\hat{\chi}_2(\hat{\xi}_2)\hat{\chi}_2(\xi_3), j = 1,\dots,p_7-1$$
$$\hat{\phi}_{16,j}(\xi_1,\xi_2,\xi_3) = \hat{\chi}_1(\hat{\xi}_1)\hat{\chi}_{2+j}(\hat{\xi}_2)\hat{\chi}_2(\xi_3), j = 1,\dots,p_8-1$$
$$\hat{\phi}_{17,j}(\xi_1,\xi_2,\xi_3) = \hat{\chi}_1(\hat{\xi}_1)\hat{\chi}_1(\hat{\xi}_2)\hat{\chi}_{2+j}(\xi_3), j = 1,\dots,p_9-1$$
$$\hat{\phi}_{18,j}(\xi_1,\xi_2,\xi_3) = \hat{\chi}_1(\hat{\xi}_1)\hat{\chi}_2(\hat{\xi}_2)\hat{\chi}_{2+j}(\xi_3), j = 1,\dots,p_{10}-1$$
$$\hat{\phi}_{19,j}(\xi_1,\xi_2,\xi_3) = \hat{\chi}_2(\hat{\xi}_1)\hat{\chi}_1(\hat{\xi}_2)\hat{\chi}_{2+j}(\xi_3), j = 1,\dots,p_{11}-1$$
$$\hat{\phi}_{20,j}(\xi_1,\xi_2,\xi_3) = \hat{\chi}_2(\hat{\xi}_1)\hat{\chi}_2(\hat{\xi}_2)\hat{\chi}_{2+j}(\xi_3), j = 1,\dots,p_{12}-1$$

Similarly, the *face shape functions* for $hp$ finite element are different, they allow for varying the polynomial orders of approximation in both directions. They are defined as

$$\hat{\phi}_{21,i,j} = \hat{\xi}_{2+i}(\chi_1)\hat{\xi}_{2+j}(\chi_2)\hat{\xi}_1(\chi_3) i = 1,\dots,p_{1,h}-1, j = 1,\dots,p_{1,v}-1$$
$$\hat{\phi}_{22,i,j} = \hat{\xi}_{2+i}(\chi_1)\hat{\xi}_{2+j}(\chi_2)\hat{\xi}_2(\chi_3) i = 1,\dots,p_{2,h}-1, j = 1,\dots,p_{2,v}-1$$
$$\hat{\phi}_{23,i,j} = \hat{\xi}_{2+i}(\chi_1)\hat{\xi}_1(\chi_2)\hat{\xi}_{2+j}(\chi_3) i = 1,\dots,p_{3,h}-1, j = 1,\dots,p_{3,v}-1$$
$$\hat{\phi}_{24,i,j} = \hat{\xi}_{2+i}(\chi_1)\hat{\xi}_2(\chi_2)\hat{\xi}_{2+j}(\chi_3) i = 1,\dots,p_{4,h}-1, j = 1,\dots,p_{4,v}-1$$
$$\hat{\phi}_{25,i,j} = \hat{\xi}_1(\chi_1)\hat{\xi}_{2+i}(\chi_2)\hat{\xi}_{2+j}(\chi_3) i = 1,\dots,p_{5,h}-1, j = 1,\dots,p_{5,v}-1$$
$$\hat{\phi}_{26,i,j} = \hat{\xi}_2(\chi_1)\hat{\xi}_{2+i}(\chi_2)\hat{\xi}_{2+j}(\chi_3) i = 1,\dots,p_{6,h}-1, j = 1,\dots,p_{6,v}-1$$

Finally, the *interior shape functions* of the $hp$ finite element allows for changing the polynomial orders of approximation in three different directions:

$$\hat{\phi}_{27,i,j,k} = \hat{\xi}_{2+j}(\chi_1)\hat{\xi}_{2+j}(\chi_2)\hat{\xi}_{3+j}(\chi_3)$$
$$i = 1,\dots,p_{\xi_1}, j = 1,\dots,p_{\xi_2}-1, k = 1,\dots,p_{\xi_3}-1$$

In this book, we will focus on the $h$-adaptive Finite Element Method, that is using only hierarchical refinements, with polynomial order of approximation $p$ fixed uniformly over the entire computational mesh.

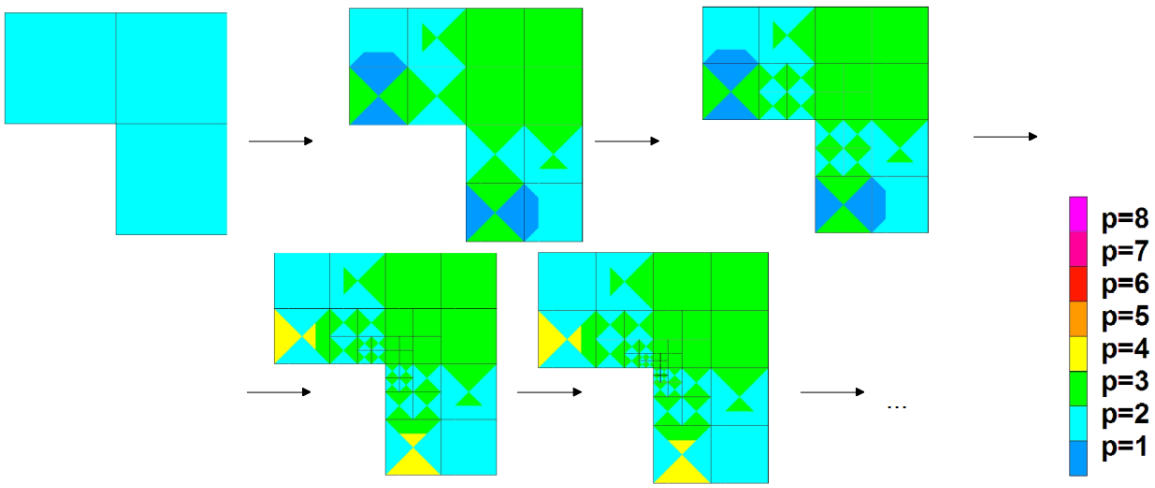



Figure 2.4: A sequence of $hp$ refined meshes in two dimensions. Different polynomial ordders of approximation of finite element edges and interiors (in both directions) are denoted by different colors.

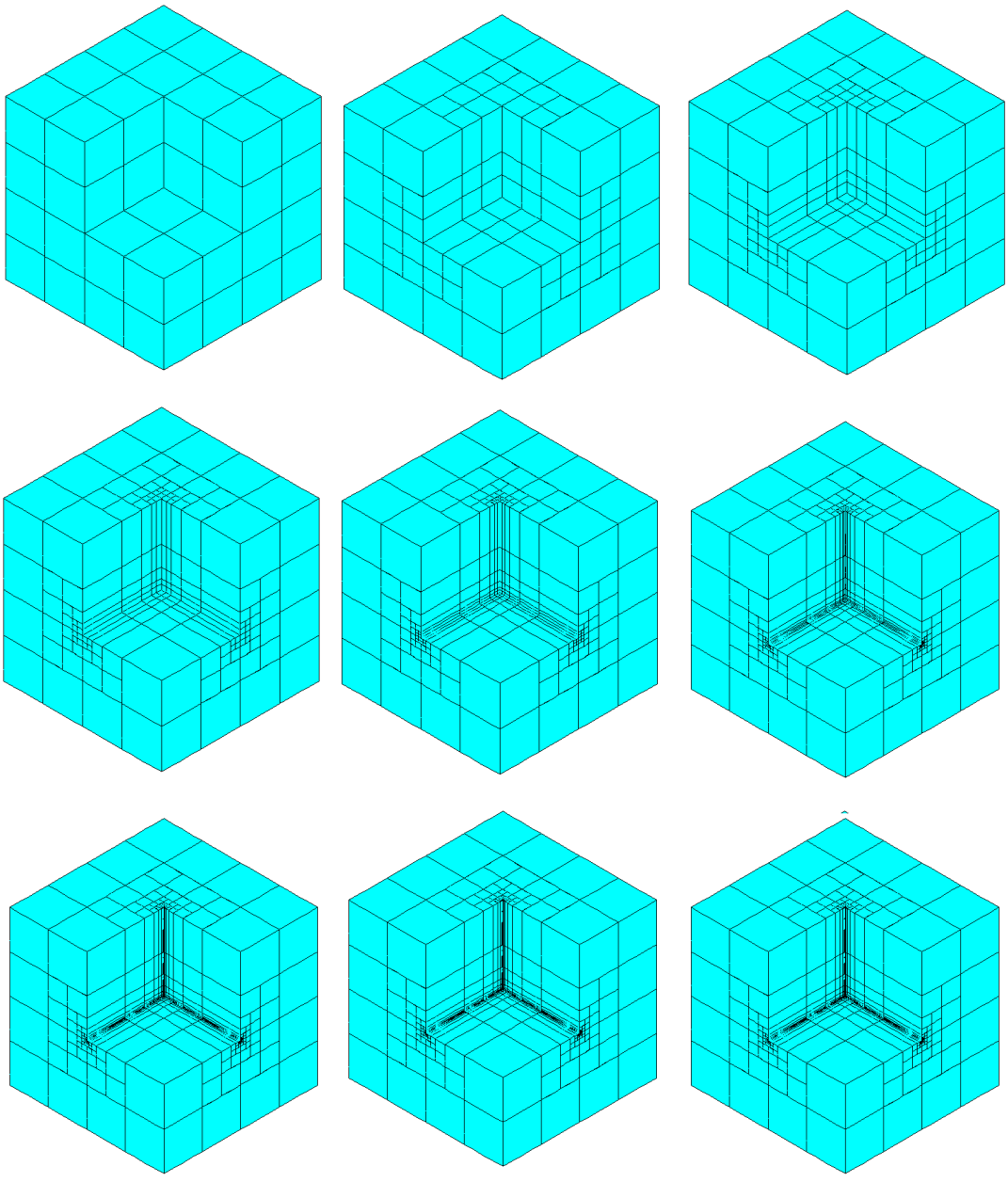


Figure 2.5: A sequence of $h$ refined meshes in three dimensions.

## 2.4. Discretization of the problem

To calculate a solution for a problem using the Finite Element Method, we need to discretize the unknown function to a finite set of variables over which we construct the linear equation system. To approximate the function, a set of basis fuctions generated for each mesh is used – the basis functions depend on the method and geometry of the mesh. In particular, for $h$-adaptive and $hp$-adaptive grids, the basis functions are spread over the finite element vertices, edges (in 1D and 2D), faces (in 2D), interiors (in 3D), hyperedges, hyperfaces, etc. and interiors in higher dimensions. The unknown function will be approximated by summation of the basis functions multiplied by corresponding variables, which will be calculated by the solver.

In order to discretize the problem, we first build a system of linear equations. For the sake of simplicity, we will consider the L-2 projection problem. For a given non-linear function $[0,1]^d \ni (x_1,\dots,x_d) \to \mathcal{F}(x_1,\dots,x_d) \in \mathcal{R}$ find its discrete piece-wise linear approximation $[0,1]^d \ni (x_1,\dots,x_d) \to u_h(x_1,\dots,x_d) \in \mathcal{R}$ such that $\|u-F\| \to 0$. This problem can be computed by solving a system of linear equations

$$\begin{aligned} b(u_h,v) &= l(v_h) \quad \forall v_h \in V_h \\ b(u_h,v_h) &= \int_{[0,1]^d} u_h(x_1,\dots,x_d)v_h(x_1,\dots,x_d)dx_1\dots dx_d \\ l(v_h) &= \int_{[0,1]^d} F(x_1,\dots,x_d)v_h(x_1,\dots,x_d)dx_1\dots dx_d \end{aligned} \tag{2.5}$$

where $V_h$ is the computational mesh approximaton space.

**Definition 2.4.1.** *Computational mesh problem: Find $\{u_h^i\}_{i=1}^{N_h}$ coefficients (dofs) of approximate solution $V \supset V_h \ni u_h = \sum_{i=1}^{N_h} u_h^i e_h^i$ fulfilling the problem (2.6).*

**Definition 2.4.2.** *The computational mesh approximation space is defined as*

$$V_h = span\{e_h^j : \forall K \in T_h|_K, \forall \phi_k \in X(K), \exists! e_h^i : e_h^i|_K = \phi_k\}$$

*where $e_h^i$ is a global basis function (element basis of $V_h$), $\phi_k$ is a shape function and $(k,K) \to i(k_1,K)$ is the mapping over the computational mesh assigning global number $i(k,K)$ of dofs (basis functions) related with shape function $k$ from element $K$*

**Remark 2.4.1.** *The approximation space $V_{hp} \subset V$ with basis $\{e_{hp}^i\}_{i=1}^{N_{hp}}$ is constructed by gluing together element-local shape functions.*

This is equivallent to solving the following sparse system of linear equations

$$\begin{bmatrix} b(e_h^1,e_h^1) & \cdots & b(e_h^1,e_h^N) \\ \cdots & \cdots & \cdots \\ b(e_h^N,e_h^1) & \cdots & b(e_h^N,e_h^N) \end{bmatrix} \begin{bmatrix} u_h^1 \\ \cdots \\ u_h^N \end{bmatrix} = \begin{bmatrix} l(e_h^1) \\ \cdots \\ l(e_h^N) \end{bmatrix} \tag{2.6}$$

The computational cost of solution the system of linear equations depends on its sparsity pattern and the solver algorithm being used.

In other words, the generated system of linear equations can be represented by a square matrix in which each row (equation) and each column (variable) corresponds to a basis function. A non-zero value on an intersection of a row and column means that there is an overlap of supports between the basis functions corresponding to that row and column. If two basis functions do not overlap, both intersections in the matrix will always equal zero.

The supports of the basis functions depends on the discretization method being used. For example, in the case of hierarchical basis functions, the support of basis functions is limited to all the elements sharing the edge or

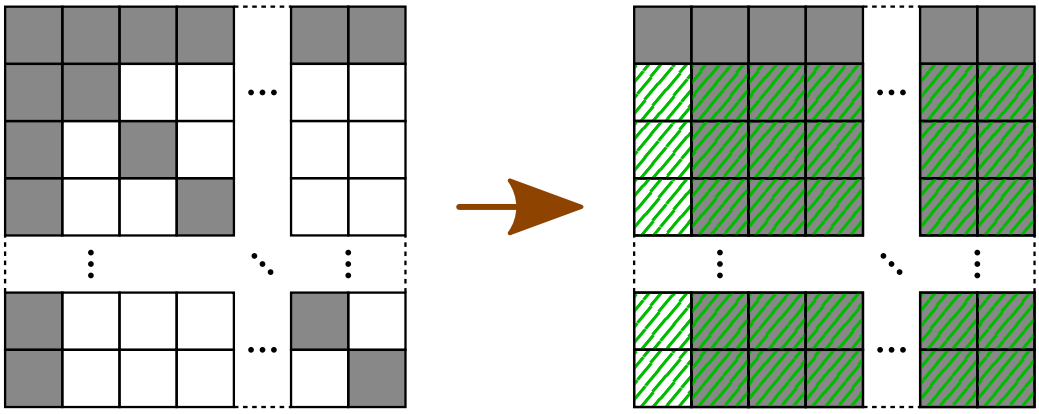
face or hyperface etc. The only exception to this rule is the case of neighboring elements with different refinement levels – in this case the supports of basis functions can spread over parent elements.

The abovementioned rule of intersections does not depend on the partial differential equations being solved. In particular, this means that the matrix resulting from discretization of the computational problem over adaptive multi-dimensional grid is sparse, and thus the computational complexity of factorization can be reduced if a smart order of elimination will be used.

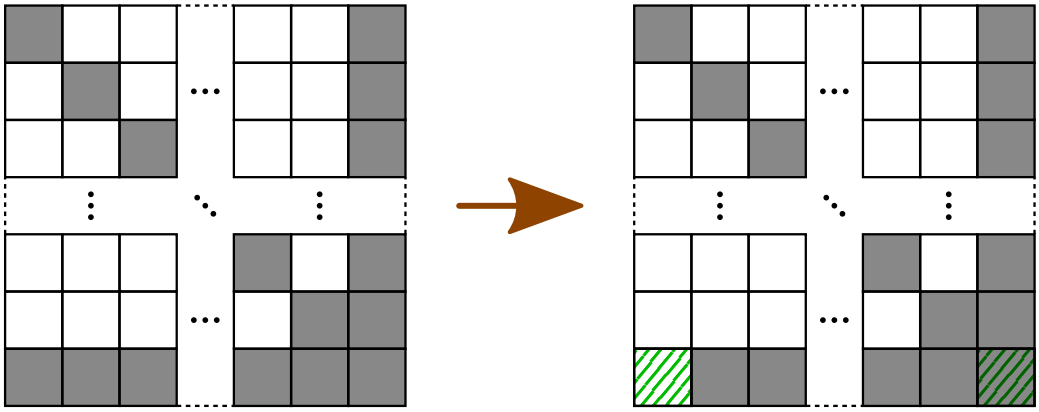
## 2.5. Matrix reordering

Depending on the order of row elimination, the computational cost of solution can vary by several orders of magnitude. As we will see later, with proper row reordering we can reduce theoretical time complexity from naive $\mathcal{O}(n^3)$ to $\mathcal{O}(n^\alpha)$ where $\alpha$ is some number that can be as low as 1 for certain classes of grids. An extreme example of how row ordering can impact the computational complexity can be seen in figure 2.6.

Figure 2.6: Examples of non-optimal and optimal row ordering for Gauss elimination



(a) In the case of this sparse matrix (gray denotes non-zero elements), if the rows are eliminated top to bottom, we be immediately working on an almost full matrix after just the first row eliminated (values that changed after the first step are marked in green). The time complexity of such factorization of this matrix will be $\mathcal{O}(n^3)$, where $n$ is the size of the input.



(b) When the rows and columns are reordered by moving the first row and column to the end, the same top to bottom order of elimination will result in linear time complexity ($\mathcal{O}(n)$).

Similar reduction of the number of non-zero entries generated during the Gauss elimination process can be achieved for sparse matrices obtained by discretization of multi-dimensional adaptive grids. This can be achieved with a smart reordering obtained by analyzing the topological structure of the grid. The problem of finding the optimal order of elimination of unknowns for the direct solver, in general, is NP-complete [50], however there are several heuristical algorithms analyzing the sparsity pattern of the resulting matrix [30, 45, 2, 23]. The methods that we analyze in this book differ from traditional methods by the fact that they generate the ordering by looking at the structure of the refined grid, and not by looking at the sparsity patterns of the matrix as most common methods do.

## 2.6. Known time complexity of direct solvers for Finite Element Method

The most time consuming part of direct solver Finite Element Method algorithms is the solving of the system of linear equations. In the general case, the time complexity of solving such a system in exact numbers will be the complexity of the general case of Gaussian factorization, that is a pessimistic $O(N^3)$, where $N$ is the amount of variables. However, on sparse matrices, the complexity of the factorization can be lowered if a proper row elimination order is used – in some cases even a linear time complexity can be achieved. The matrices corresponding to hierarchically adapted meshes are not only sparse, but also tend to have a highly regular structure that corresponds to the geometrical structure of the mesh (see figure 2.7 for examples).

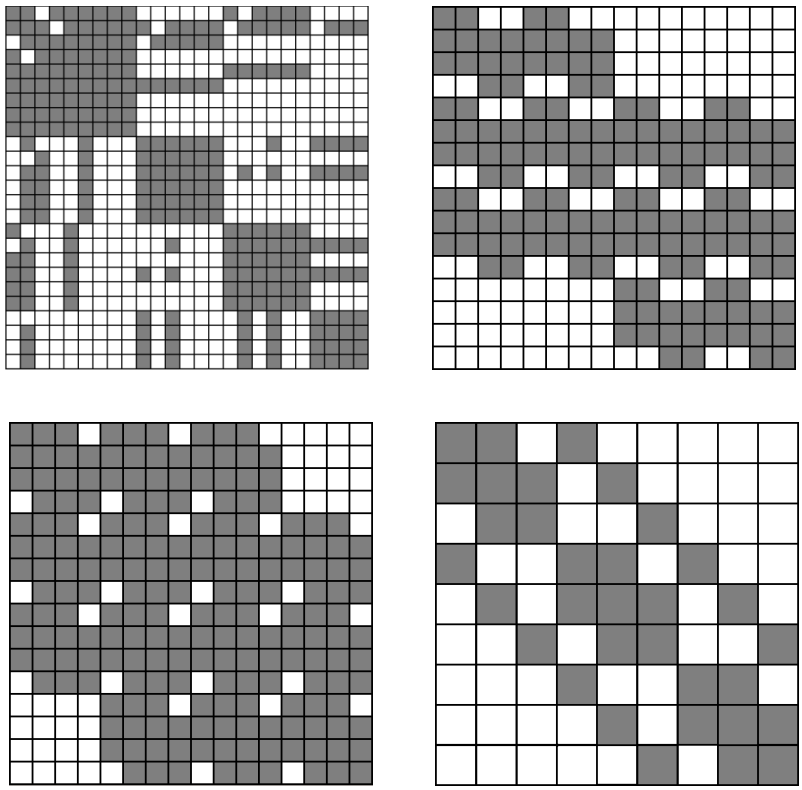


Figure 2.7: Some examples of sparse and highly regular matrices generated for FEM problems. The gray elements denote non-zero values.

The time complexity of the factorization can be indeed much better than $O(N^3)$. In particular, it has been shown that for three-dimensional uniformly refined grids, the computational cost is in the order of $\mathcal{O}(N^2)$ [36, 12]. For three-dimensional grids adapted towards a singularity in a point, edge, and face, the computational complexities

are $\mathcal{O}(N), \mathcal{O}(N)$, and $\mathcal{O}(N^{1.5})$, respectively [43]. These estimates assume a prescribed order of elimination of variables [40, 46].

Similarly, for two dimensions, the computational complexity for uniform grids is known to be $\mathcal{O}(N^{1.5})$, and for the grids refined towards a singularity in a point or an edge it is $\mathcal{O}(N)$ [42]. These estimates assume a prescribed order of elimination of variables [1]. The orderings resulting in such linear or quasi-linear computational costs can also be used as preconditioners for iterative solvers [41].

For all others, there is no known general formula or method of calculation of the computational complexity. It is neither hard to imagine that such a formula or method might not be discovered any time soon, nor that such formula or method would be simple enough to be applicable in real-world problems. Thus in this paper, we focus on a specific class of problems: hierarchically adapted meshes refined towards singularities, and generalize results discussed in [43, 40, 46, 42, 1] into an arbitrary spatial dimension and arbitrary type of singularity.

## 2.7. Space-time formulations

There is an increasing interest in the computational science community in developing space-time formulations for difficult computational problems. These space-time formulations usually encapsulate some additional stabilization methods. Paper [44] employs space-time stabilized formulation using an adaptive constrained first-order system with the least squares method. Another space-time discretizations for the constrained first-order system least square method (CFOSLS) are discussed in [49]. It is also possible to employ Discontinuous Petrov-Galerkin method for stabilization of the space-time formulation, as it is illustrated for the Schrödinger equation in [18] and for the acoustic wave propagation in [28, 22]. The least-square finite element method has been applied for stabilization of the parabolic problem in [24]. The mathematical details of the stabilization methods are beyond the scope of this book. However, the sparsity pattern of the space-time matrices follows the same rules as for standard formulations. The non-zero entries in the matrix result from an overlap of basis functions associated with the rows and columns of the matrix. The critical point of the space-time formulation is the computational cost of the solver. Some particular versions of the solvers are discussed in [39, 47]. In this book we will compare the computational costs of general direct and iterative solvers applied for the space-time formulation and for standard time-marching schemes.

Chapter 3

# Element partition tree based solvers and their complexity

The purpose of this chapter is to introduce the basic tools used in further chapters to analyze the computational complexities of the grids described in later chapters.

## 3.1. Hierarchically adapted meshes

Unless specified otherwise, in this book the word *grid* will be meant to denote orthogonal hierarchically adapted mesh. We will use word *mesh* to refer to the more general concept without assuming any particular geometry.

To construct a hierarchically adapted mesh, we start with some initial grid, typically containing just a single element. In this book we will assume that the initial grid contains only orthogonal elements of equal size in each direction, but our findings should also be true for other meshes. Then, as long as it is deemed necessary, the grid is adapted in an interative fashion by selecting all elements that need to be refined and dividing them into smaller parts.

In this book, we are focusing on *isotropic refinements*, that is refinements, where the refined elements are always divided in each cardinal direction into exactly $W$ smaller parts, where $W$ is some chosen constant. If no other value is specified, we will assume that $W = 2$. This way, each refined element will be divided into $W^D$ smaller elements, where $D$ is the number of dimensions of the space.

Normally, the elements chosen to be refined are those over which the approximation does not fulfill some acceptance criteria – in this book, however, we do not discuss the mathematical aspects of the refinement selection. In addition to the elements refined to improve the approximation, there will often be other elements that have to be refined due to some specific constraints on the structure of the grid imposed by the chosen Finite Element Method.

### 3.1.1. Meshes adapted towards a singularity

The focus of this book is the computational complexity of solvers working on meshes that have been hierarchically adapted to fit a problem with a singularity. A *singularity* can be understood as a closed subset of space over wich the Finite Element Method never converges. The existence of a singularity might cause the mesh to grow infinitely large, so an artificial limit of refinement rounds is necessary – we will call it the refinement level of the mesh. As the refinement level of the mesh grows, the number of variables will grow as well and so will the computational cost.

When referring to a mesh hierarchically refined towards a singularity $S$ up to a refinement level $R$ we will have in mind a mesh that has been through $R$ refinement rounds from its initial state. During each refinement round, all elements that have an overlap with the singularity $S$ of at least one point will undergo a division into smaller elements, then, if required by chosen type of the Finite Element Method, some extra elements might also get divided to keep some mesh constraints. An example of sequence of further refinemnt levels over a singularity has been shown on Figure 3.1. We generally assume that the hierarchical refinements of elements are done uniformly for all dimensions of the grid, however the findings from this paper should also extend to other types of refinements.

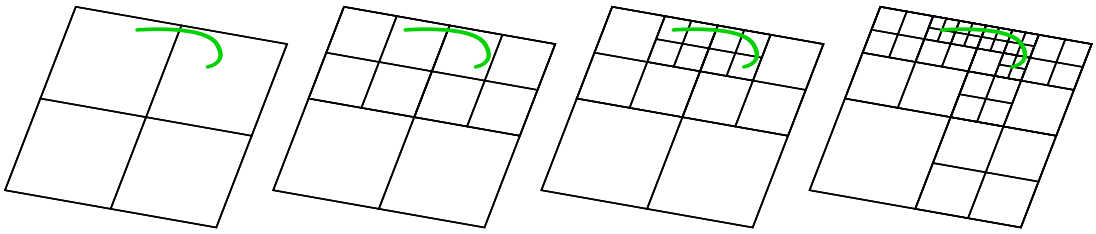


Figure 3.1: An example of consecutive refinement levels of hierarchical adaptation towards a singularity (in green). Here, apart from the refinement of elements overlapping the singularity, an additional constraint, the 1-irregularity rule is applied (see Section 3.2.4 for definition).

In this book, we will refer to singularities by their shape. For instance, a point singularity is a singularity consisting of a single point and a path singularity is a singularity consisting of some path. We can also refer to the singularities by their dimensionality. For example, a point singularity is a 0-dimensional singularity, an edge or edge singularities are 1-dimensional singularities, etc. It is worth noting, that a regular grid can be thought as a grid build over $D$-dimensional singularity containing the whole space ($D$ denotes the number of dimensions of the space).

### 3.1.2. Calculation cost and complexity of meshes refined towards singularities

In general, Finite Element Method calculations are being optimized for their precision and their calculation cost – often those two parameters being inversely related to each other. Usually, it is possible to shorten the running time by sacrificing the accuracy, or, conversely, reduce the error by increasing the amount of required computations. Unfortunately, it is impossible to analyze the relation between the precision and calculation cost in general case, as, for each problem, a different Finite Element Method might be the most applicable, and comparing results between two problems is often meaningless.

The abovementioned relation between precision and speed starts to be analyzable for a single problem for a preselected method. If we make a reasonable assumption that with each refinement level of hierarchically adapted meshes the precision grows, we can correlate that growth with the increase of the calculation cost. In particular, we will analyze in this book the time complexity of solvers for specific types of meshes.

There is a vital observation to be done here: if we consider a solver algorithm in general, its time complexity will be the worst-case scenario complexity[1]. For this reason, we will analyze the time complexity of solver algorithms for specific classes of meshes. This way we will arrive at several different time complexities of a single algorithm, different for each class of meshes. It is also imporant to note that in this book we analyze the relationship between the computational cost of the solver as a direct function of the number of variables. The number of variables might meaningfully correlate with the precision of the solution, but is not comparable directly.

In this book, we are analyzing the time complexity of direct solvers for meshes with singularity. For a given singularity, the mesh will grow in number of elements and variables as the refinement level increases. It would not

[1] This is true for analyzed solvers, not for all types of algorithms.

be unreasonable to consider the running time as a function of the refinement level – especially so, considering that the local accuracies of the solution usually depend on the size of elements, or their refinement level. While it is a useful calculation, it is not, strictly speaking, the time complexity of the solver. We can, however, define the time complexity $T$ of a solver with an alternative definition based on the refinement level $R$:

$$T(N) = \mathcal{O}(f(N)) \iff \exists R_0 \geq 0, C > 0 : \forall R \geq R_0 : |T(N(R))| \leq C \cdot f(N(R)) \tag{3.1}$$

## 3.2. $h$-adaptive Finite Element Method

Most of this book will describe the calculation costs of Finite Element Method over a variant of $h$-adaptive grids with uniform refinements. Such grids will have a single $p$ parameter selected over the whole domain. Unless another type of mesh is directly specified, $h$-adaptive grid can be understood as the variant described in this section.

$h$-adaptive grides have basis functions defined over geometrical features of the elements of the grid: vertices and segments in 1D, vertices, edges and interiors in 2D, vertices, edges, faces and interiors in 3D, vertices, edges, faces, hyperfaces and interiors in 4D, etc. The simplest basis function shapes in two-dimensional space with their supports have been shown on Figure 3.2.

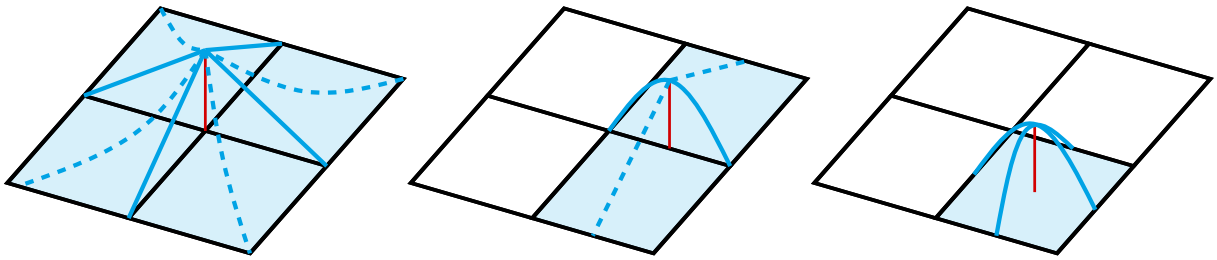


Figure 3.2: Basis functions of 2-dimensional $h$-adaptive grid with $p = 2$ and their support – based on vertex, edge and interior respectively.

### 3.2.1. Elementary basis function shapes of $h$-adaptive grids

In this book, we employ $d$ dimensional hypercube elements (rectangles in 2D, hexahedral in 3D). We utilize hierarchical basis functions of order $p$=2. We define one-dimensional hierarchical shape-functions, which will be used later to define the two-, three-, and higher-dimensional basis functions by their tensor products and by gluing together the shape functions from adjacent elements. For more details, we refer to [16].

$$\begin{aligned} \hat{\chi}_1(\xi) &= 1 - \xi \\ \hat{\chi}_2(\xi) &= \xi \\ \hat{\chi}_3(\xi) &= (1 - \xi)\xi(2\xi - 1) \end{aligned} \tag{3.2}$$

These shape functions are used first to define shape functions over two-, three-dimensional and higher-dimensional finite elements. For example, in two dimensions, equation 3.3 defines shape functions over each of the four vertices of the element:

$$\begin{aligned} \hat{\phi}_1(\xi_1, \xi_2) &= \hat{\chi}_1(\xi_1)\hat{\chi}_1(\xi_2) \\ \hat{\phi}_2(\xi_1, \xi_2) &= \hat{\chi}_2(\xi_1)\hat{\chi}_1(\xi_2) \\ \hat{\phi}_3(\xi_1, \xi_2) &= \hat{\chi}_2(\xi_1)\hat{\chi}_2(\xi_2) \end{aligned}$$

$$\hat{\phi}_4(\xi_1,\xi_2) = \hat{\chi}_1(\xi_1)\hat{\chi}_2(\xi_2) \tag{3.3}$$

The vertex shape functions are created by the tensor product of $d$ one-dimensional linear shape functions.

Next, Equation 3.4 defines quadratic shape functions over each of the four edges of the element:

$$\begin{aligned}
\hat{\phi}_{5,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_{2+j}(\xi_1)\hat{\chi}_1(\xi_2) \quad j = 1,...,p-1 \\
\hat{\phi}_{6,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_2(\xi_1)\hat{\chi}_{2+j}(\xi_2) \quad j = 1,...,p-1 \\
\hat{\phi}_{7,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_{2+j}(\xi_1)\hat{\chi}_2(\xi_2) \quad j = 1,...,p-1 \\
\hat{\phi}_{8,j}(\xi_1,\xi_2,\xi_3) &= \hat{\chi}_1(\xi_1)\hat{\chi}_{2+j}(\xi_2) \quad j = 1,...,p-1
\end{aligned} \tag{3.4}$$

where $p$ denotes the polynomial order of approximation over the finite element. The edge shape functions are constructed by multiplying one higher-order shape function and $d$-1 linear shape functions over a finite element. The edge shape functions are hierarchical. Namely, they define $(p-1)$ shape functions for the approximation of order $p$.

Finally, we define quadratic basis function over an element interior

$$\begin{aligned}
\hat{\phi}_{9,ij}(\xi_1,\xi_2) &= \hat{\chi}_{2+i}(\xi_1)\hat{\chi}_{2+j}(\xi_2) \\
i &= 1,...,p-1, j = 1,...,p-1
\end{aligned} \tag{3.5}$$

The interior shape functions are constructed by the tensor product of $d$ higher-order shape functions. The interior shape functions are hierarchical. Namely, they define $(p-1)^d$ shape functions for the approximation of order $p$.

This tensor product construction can be generalized to arbitrary $d$-dimensional finite elements. For example, we consider shape functions over vertices, edges, faces, and interiors in three dimensions. In four dimensions, we consider shape functions over vertices, edges, faces, hyperfaces, and interiors.

We identify nodes of the mesh with basis functions. We also consider supports of nodes, defined as supports of basis functions associated with the node. We have basis functions assigned to vertices, edges, faces, hiperfaces (in higher dimensions), and interiors. In general, the support of the node span over all the adjacent elements having the node. For example, in the case of two-dimensional regular mesh, the support of the vertex node spans into four elements having the node, the support of the edge node spans into two elements having the node, and the support of the interior node is equal to the single element.

### 3.2.2. Constrained nodes in $h$-adaptive grids

Each feature that is not a feature of all containing elements (for example a vertex of a smaller element might not be a vertex of a larger neighboring element) will produce so called *constrained nodes*. Such nodes would create dependent variables – for instance, Figure 3.3 shows an example of such case. To avoid creating dependent variables, the constrained nodes are omitted altogether.

### 3.2.3. Indentation of basis functions in $h$-adaptive grids

The procedure of constructing basis functions as described in the preceding subsections for hierarchically adapted grid will produce functions with supports contained within another basis function of lower refinement level. This would increase the number of overlapping basis functions (and related non-zero elements in the matrix) very significantly. To avoid that, we can modify slightly the basis functions. Owning to the mathematical properties of the shapes, whenever a basis function has support completely overlapping a support of another basis function of the same shape type with higher refinement level, we can subtract the inner basis function multiplied by some factor from the outer basis function so that some of the support of the outer basis function is zeroed. Such a process will be called in this book *indentation*. You can see an example of indented basis functions in the Figure 3.4.

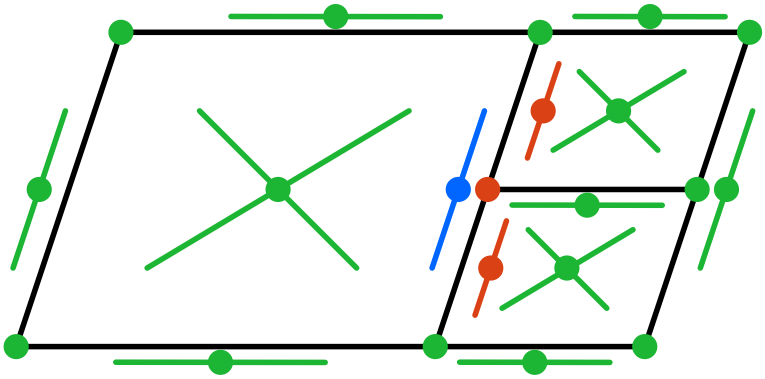
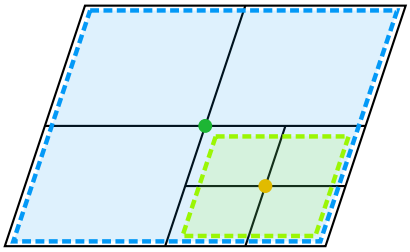
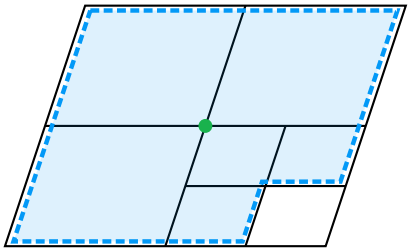


Figure 3.3: Nodes shown in red are constrained – they are not features of all containing elements. The basis function of the blue unconstrained node is a sum of the three basis functions of the red nodes multiplied by some factors – if we do not omit some of them, we will be left with an dependend system of equationsNodes marked in green belong to all elements containing them and are unconstrained.

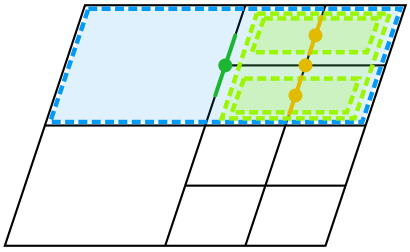
Figure 3.4: Examples of node supports.



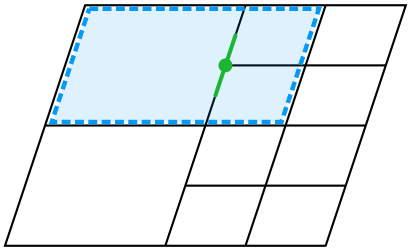
(a) Support of vertex node (in green) before indentation.



(b) Final support of vertex node after indentation.



(c) Support of edge node (in green) before indentation.



(d) Final support of edge node after indentation.

### 3.2.4. 1-irregularity rule

A correct $h$-adaptive grid should conform to an additional constraint: the 1-*irregularity rule*.

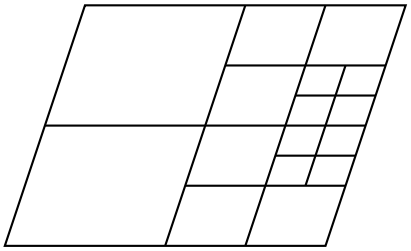
**Definition 3.2.1.** 1-***irregularity rule.*** *Two elements sharing an edge cannot differ in refinement level by more than 1. When splitting an element in the refinement process, any larger elements sharing an edge should be split as well.*

Without the 1-*irregularity rule* the complexity of basis function types would grow excessively, and, as will be shown later, this rule also serves as a way to avoid specific features in the grid that result in a higher than necessary time complexity of the solver. Additionally, note that if two elements share a vertex then there exists an element

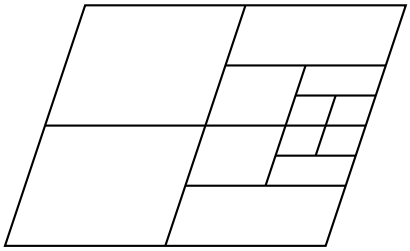
that shares an edge with both of them. Because of that, the 1-*irregularity rule* will also enforce the property of the grid that elements sharing a vertex differ in refinement level by no more than two refinement levels.

Figure 3.5 shows example of a correct and several incorrect $h$-adaptive grids.

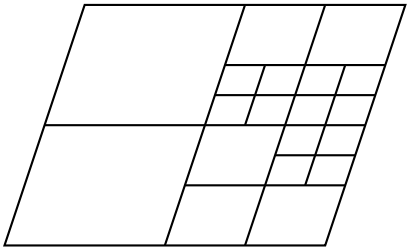
Figure 3.5: Correct and incorrect uniformly refined $h$-adaptive grids.



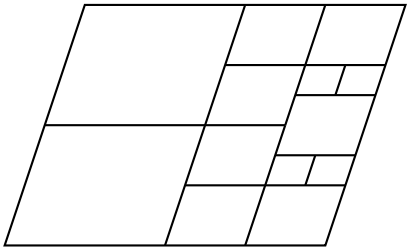
(a) A correct $h$-adaptive grid.



(b) Not a correct $h$-adaptive grid. Not all elements are squares, thus this grid cannot be constructed using uniform refinements.



(c) Not a correct $h$-adaptive grid. 1-irregularity rule is not being followed.



(d) Not a correct $h$-adaptive grid. It cannot be build by a sequence of hierarchical refinements.

### 3.2.5. Construction of an $h$-adaptive grid over a singularity

To construct an $h$-adaptive mesh around a singularity, we start with the one initial element and iteratively refine all elements that overlap with the singularity, ensuring that the 1-irregularity rule is being followed. For example, Algorithm 1 can be used to build $h$-adaptive mesh around a singularity with shape $S$ with refinement level $R$, where the symbol " [] " stands for initiation as an empty array. We assume that information on the singularity location is available. An exemplary singularity mesh construction is shown in Figure 3.6.

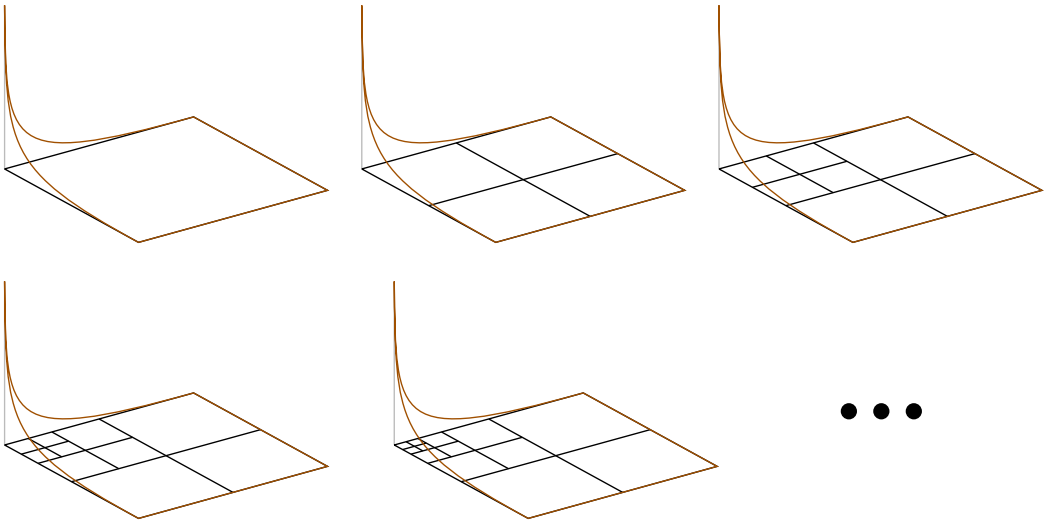


Figure 3.6: Mesh refined over a point singularity resulting from a gradient of a two-dimensional function. The shape of the singularity $S$ is a point in the corner.

**Algorithm 1** Mesh construction over a singularity

**procedure** CONSTRUCTMESH($RootElement, R, S$)
  $root \leftarrow RootElement$
  $G \leftarrow []$ ▷ $G[r]$ contains all elements of refinement level $r$.
  $G[0] \leftarrow [root]$ ▷ Initialize the mesh with single element.
  **for** $r \leftarrow 1$ **to** $R$ **do** ▷ **Step 1**: Refine all elements as necessary.
    $H \leftarrow G[r-1]$ ▷ Only elements of refinement $r-1$ can be refined further.
    $G[r-1] \leftarrow []$
    $G[r] \leftarrow []$
    **for all** $e \in H$ **do**
      **if** $e$ overlaps $S$ **then**
        **if** $i > 1$ **then**
          $K \leftarrow G[r-2]$
          $G[r-2] \leftarrow []$
          **for all** $f \in K$ **do**
            **if** $f$ shares at least an edge with $e$ **then**
              $G[r-1] \leftarrow$ CONCAT$(G[r-1],$ REFINE$(f))$
            **else**
              $G[r-2] \leftarrow$ APPEND$(G[r-2], f)$
            **end if**
          **end for**
        **end if**
        $G[r] \leftarrow$ CONCAT$(G[R],$ REFINE$(e))$ ▷ If analyzed element overlaps the singularity, refine it into smaller elements
      **else**
        $G[r-1] \leftarrow$ APPEND$(G[r-1], e)$ ▷ ...otherwise, leave it unrefined.
      **end if**
    **end for**
  **end for**
  $M \leftarrow []$
  **for** $r \leftarrow 0$ **to** $R$ **do**
    $M \leftarrow$ CONCAT$(M, G[r])$
  **end for**
  **return** $M$
**end procedure**

## 3.3. Element partition tree

The element partition tree is used for the construction of ordering for sparse matrix permutation in order to speed up the multi-frontal solver. Using some partitioning strategy, an element partition tree is created by recursive partitioning the mesh elements into two parts [1, 40, 42]. An example of an element partition is presented in Figure 3.7.

An element partition tree for a mesh consisting of a set of elements $E$ is a binary tree defined as $T = (E, V, c_1, c_2, e)$ with the following properties:

1. $V$ is a set of tree nodes,
2. $c_1 : V \to V$ and $c_2 : V \to V$ are the functions assigning left and right child to a node, respectively,
3. $e : V \to P(A)$ is a function assigning subsets of a set of all elements of the mesh $E$ to a node,
4. the root called node $ROOT$ contains all elements of the mesh, in other words, $e(ROOT) = E$,
5. each node $node$ for which $|e(node)| = 1$ is a leaf in the tree,
6. each node $node$ for which $|e(node)| > 1$ has exactly two children; $c_1(n)$ and $c_2(n)$,
7. for each node $node$ $e(node) = e(c_1(node)) \cup e(c_2(node))$ .

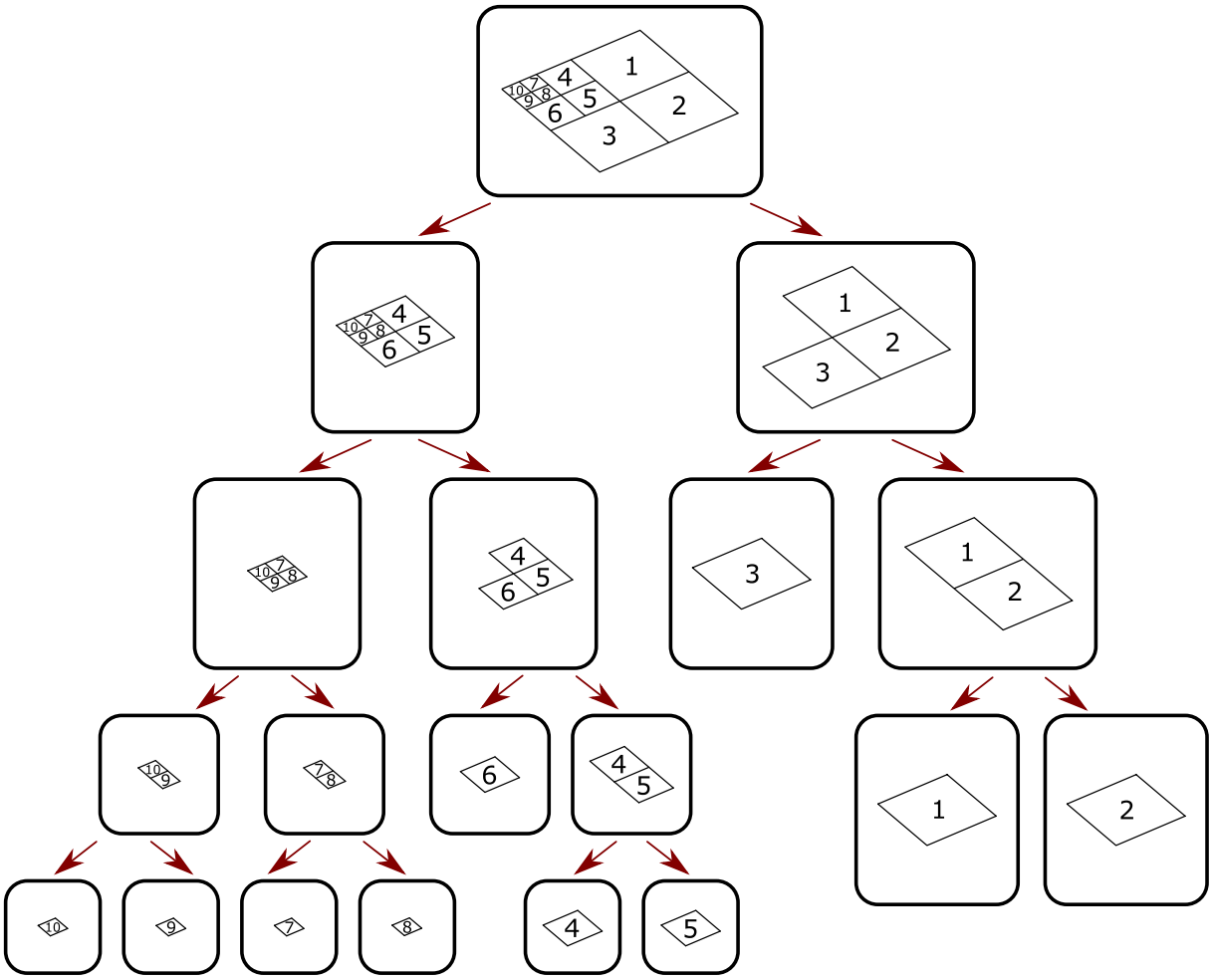



Figure 3.7: An example for an element partition tree for a small adaptive mesh. The post-order traversal of the tree results in the ordering of elements $(10, 9, 7, 8, 6, 4, 5, 3, 1, 2)$.

## 3.4. Ordering generation and solving using generated ordering

Given an element partition tree, we can generate a row elimination order for the matrix using a post-order traversal of the element partition tree. At each traversed partition tree node, we list all basis functions with support entirely contained by the tree node elements which have not been listed already. This produces a permutation (or ordering) of all nodes. Figure 3.8 presents an exemplary element partition tree with denoted basis functions for each node, which generates the ordering $(1, 4, 5, 7, 8, 9, 10, 12, 15, 11, 16, 17, 13, 14, 2, 3, 6)$.

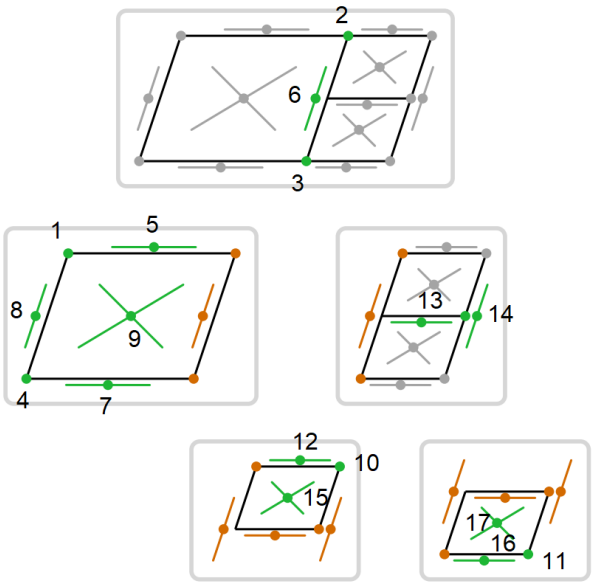



Figure 3.8: An example of element partition tree ordering generation. Nodes marked in green are eliminated at the given tree node. This example results in the permutation of $(1, 4, 5, 7, 8, 9, 10, 12, 15, 11, 16, 17, 13, 14, 2, 3, 6)$.

s

## 3.5. Sparse Gaussian elimination

We focus on the Gaussian elimination adapted to work over sparse matrices. The Algorithm 2 solves a linear equation system represented by $Mx = A$ ($M$ is a sparse matrix and $A$ is a vector) by multiplying both sides by some ordering $P$ and solving $PMx = PA$ ($P$ being a permutation matrix representing some ordering). In our case, the permutation of the matrix is based on the post-order traversal of the element partition tree. The practical implementation of the algorithm is the multi-frontal solver [3, 4, 5], avoiding zeros in matrices by constructing the elimination tree internally based on the proposed ordering and the sparsity pattern of the matrix. In our numerical experiments, we employ the multi-frontal solver from Octave.

**Algorithm 2** Permutation based sparse matrix Gauss elimination algorithm.

**procedure** SPARSEMATRIXGAUSSELIMINATION($M, A, P$)
  $M \leftarrow P \cdot M$
  $A \leftarrow P \cdot A$
  ▷ **Step 1**: forward elimination.
  **for** $i \leftarrow 1$ **to** ROWS($M$) **do**
    **for all** $j \in [i+1, \text{ROWS}(M)]$ where $M[j,i] \neq 0$ **do**
      ▷ With a proper sparse matrix implementation only actual non-zero $M[j,i]$ will be checked.
      $M[j] \leftarrow M[j] - \frac{M[j,i]}{M[i,i]} M[i]$ ▷ This operation can be done in $\mathcal{O}(N)$, where N is the number of non-zero elements in row $i$.
      $A[j] \leftarrow A[j] - \frac{M[j,i]}{M[i,i]} A[i]$
    **end for**
  **end for** ▷ At this stage the matrix will be in row echelon form.
  ▷ **Step 2**: back substitution.
  **for** $i \leftarrow$ ROWS($M$) **to** $1$ **do**
    **for all** $j \in [i+1, \text{ROWS}(M)]$ where $M[i,j] \neq 0$ **do**
      ▷ With a proper sparse matrix implementation only actual non-zero $M[i,j]$ will be checked.
      $A[i] \leftarrow A[i] - M[i][j] * A[j]$
      $M[i][j] \leftarrow 0$
    **end for**
    $A[i] \leftarrow A[i]/M[i][i]$
    $M[i][i] \leftarrow 1$
  **end for**
  **return** $A \cdot P^{-1}$
**end procedure**

## 3.6. Time complexity of element partition tree based solvers

This section shows that an element partition tree-based ordering gives a recursive formula for computation complexity that is easy to calculate. First, let us analyze the Gaussian elimination algorithm. It comprises two main steps–a first step corresponding to a forward elimination and a second step given by a backward substitution. The second step requires several operations proportional to the number of non-zero elements in the row form matrix. However, each non-zero element has to be non-zero at the beginning of the algorithm or originates from an operation performed in the first step. Therefore, the second step does not add anything to the computational complexity of the whole algorithm. For practical reasons, most sparse matrix algorithms keep an element in the memory even if it has been modified to be $0$. For the sake of brevity, we call a *non-zero element* to any element that is or has previously been set to a non-zero value, without regard to whether it is equal to $0$ at a given time.
The computational complexity of the first step can be analyzed as a sum of the complexities of eliminating rows for each element partition tree node. Let us make the following set of observations:

1. A non-zero element in the initial matrix happens when the two basis functions corresponding to that row and column have overlapping supports. Let us call the graph created by considering the initial matrix to be an adjacency matrix of a graph as an *overlap graph*. Two graph nodes cannot be neighbors in an overlap graph unless the supports of their corresponding basis functions overlap.

2. When a row is eliminated, the new non-zero elements are created on the intersection of columns and rows that has non-zero values in the eliminated row or corresponding column. If we analyze the matrix as a graph, then elimination of the row corresponding to a graph node produce edges between all pairs of nodes that were neighbors of the node being removed.
3. If at any given time during the forward elimination step a non-zero element exists on the intersection of a row and a column corresponding to two basis functions, then either those two basis functions have corresponding graph nodes that are neighbors in the overlap graph, or that there exists a path between those two nodes in the overlap graph that traverses only elements that have been eliminated already.
4. All variables corresponding to the neighboring nodes of the graph node of a variable $x_j$ in the overlap graph are either:

   (a) listed in one of the element partition tree nodes that are descendants of the element partition tree node listing the variable $x_j$ – and those variables are eliminated already by the time this variable is eliminated, or

   (b) listed in the same element partition tree node as the variable $x_j$, or

   (c) having the support of the corresponding basis function intersected by the boundary of the submesh of the element partition tree node containing the variable $x_j$ – those graph nodes are listed in one of the ancestors of the element partition tree node listing the variable $x_j$.

   Thus, in the overlap graph, there are no edges between nodes that belong to two different element partition tree nodes that are not in an ancestor-descendant relationship. At the same time, any path that connects a pair of non-neighboring nodes in the overlap graph has to go through at least one graph node corresponding to a variable that is listed in a common ancestor of the element partition tree nodes containing the variables from that pair of nodes.

These observations lead to a conclusion that a removal of a single row requires no more than $(z-1)z$ subtractions, where $z$ is the amount of variables related with graph nodes that were not removed earlier, have overlapping support and are listed at the same tree node. Last leads to the following recursive formula for the complexity of removing all mesh nodes at a single tree node:

$$\begin{aligned} T(node) &= T(child_1(node)) + T(child_2(node)) + \sum_{i=1}^{a}(b-i)(b-i+1) \\ &= T(child_1(node)) + T(child_2(node)) + \frac{1}{3}a(a^2+6b+3b^2+3ab+3a+2) \\ &= T(child_1(node)) + T(child_2(node)) + \mathcal{O}(ab^2), \end{aligned} \tag{3.6}$$

where $a$ denotes the number of variables removed at a given tree node, and $b$ is equal to $a$ plus the number of variables on the interface of that tree node.
In the following chapters, we denote the complexity of the removal of $a$ variables belonging to a tree node from $b$ variables having overlapping support with the corresponding node as:

$$C_r(a,b) = \frac{1}{3}a(a^2+6b+3b^2+3ab+3a+2) = \mathcal{O}(ab^2). \tag{3.7}$$

Chapter 4

# Computational complexity for point singularity meshes

This Chapter analyzes the time complexity of the direct solver being run on hierarchical meshes adapted toward a point singularity. While the analysis presented in the Chapter is not as generic and robust as the ones presented in the latter ones, its relative simplicity makes it a useful tool to help understand the impact of singularities on the calculation cost.

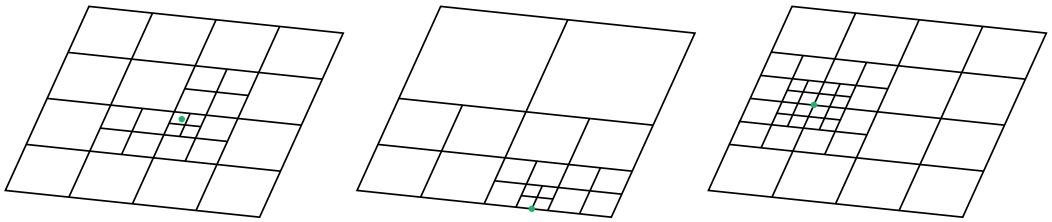
Firstly in this Chapter, we define how point singularity mesh is constructed in Section 4.1. Then, we show an example of how the computation cost can be calculated for a singularity mesh with a singularity in the corner in Section 4.2. In Section 4.3, we show a set of properties of a mesh that will guarantee that there exists a linear solver algorithm, to finally show that every well-formed $h$-adaptive mesh meets them in Section 4.4.

## 4.1. Point singularity mesh definition

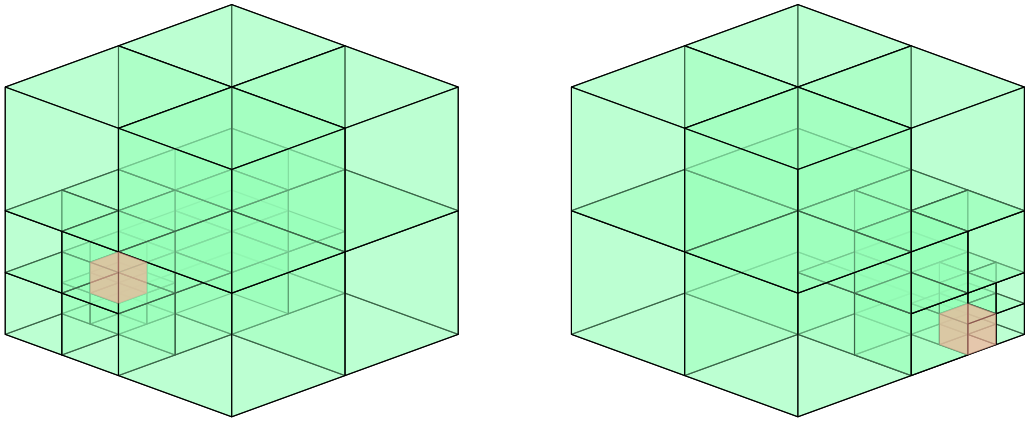
A point singularity mesh is a mesh refined hierarchically towards a single point. Let us consider a point singularity in a $D$-dimensional space refined towards point $q$ until some arbitrary refinement level $r$ – denoting such mesh as $\mathcal{G}_q^D(r)$. The singularity point $q$ can be placed either inside of an element or on a boundary of elements or elements. In particular, the singularity point can be placed on a boundary of the whole mesh. Examples of meshes with point singularities have been shown in Figure 4.1.

Figure 4.2 illustrates a process of building such a mesh using Algorithm 1.

Figure 4.1: Examples of point singulatities.

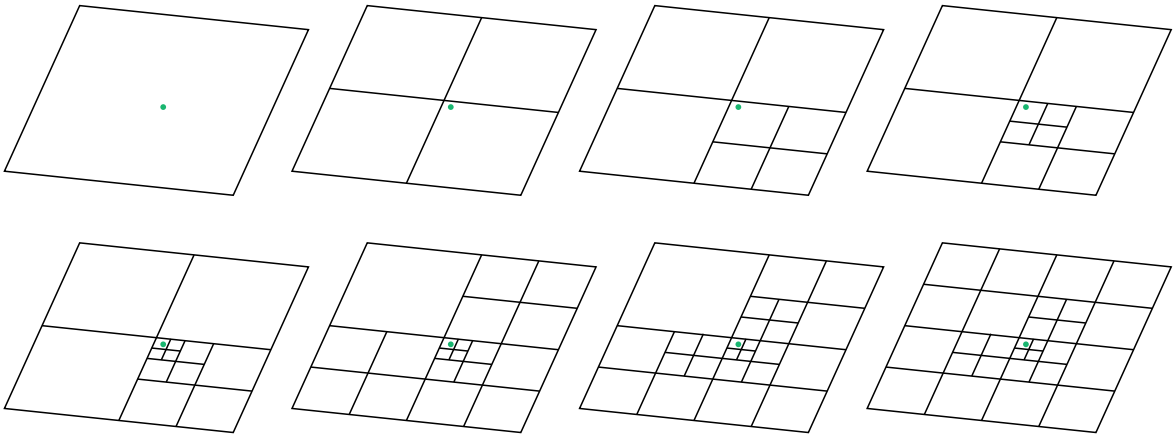


(a) Examples of $h$-adaptive singularity meshes in 2D. Green dots denote the singularity position.



(b) Examples of $h$-adaptive singularity meshes in 3D. Elements containing singularities marked in red.

Figure 4.2: A $G_q^2(4)$ mesh construction for a point $q$ close to the middle of the mesh.



## 4.2. Analysis: point singularity placed on the boundary of the mesh

As an example of a point singularity mesh we analyze a relatively simple case with a singularity point $q$ placed in one of the corners of the mesh and show a way to generate an ordering that results in linear complexity of the solver. Examples of $h$-adaptive meshes of this type in two and three dimensions can be seen in figure 4.3.

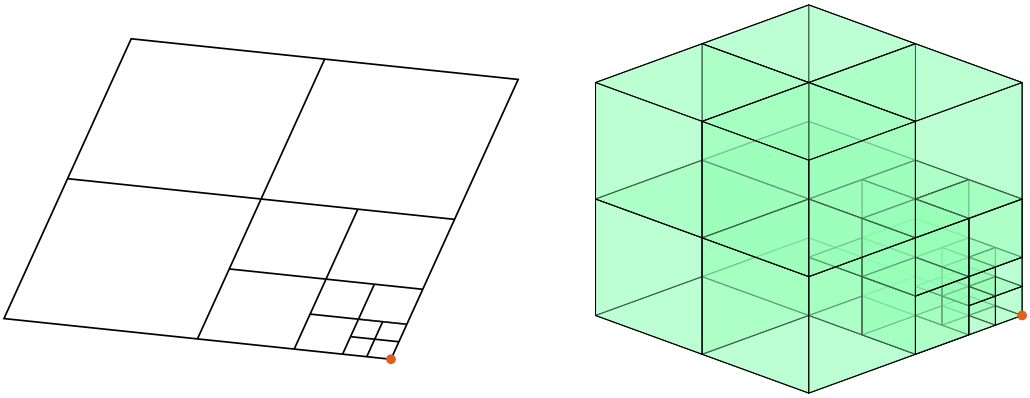


Figure 4.3: Meshes in 2D and 3D with corner point singularity marked in red.

In the case of such meshes, we do not have to do the extra step of making sure that 1-irregularity rule is met. Thus, such meshes have the number of elements $N_e$ described by the following formula:

$$N_e(G_q^d(r)) = \begin{cases} 1 & \text{if } r = 0 \\ N_e(G_q^d(r-1)) + 2^d - 1 & \text{if } r > 0 \end{cases} \tag{4.1}$$

where $r$ is the refinement level of the mesh and $d$ is the dimensionality of the mesh. The formula 4.1 is equivalent to:

$$\begin{aligned} N_e(G_q^d(r)) &= r(2^d - 1) + 1 \\ &= \Theta(2^d r) \end{aligned} \tag{4.2}$$

In other words, the number of elements in a corner point singularity mesh grows linearly with the refinement level $r$.

After the first refinement the mesh has $(2p+1)^d$ nodes. Each further refinement level adds a layer of $(2^d - 1)$ elements and for each new element, $p^d$ nodes are created. The following formula describes the number of the basis functions on such a mesh for $r \geq 1$:

$$\begin{aligned} N_v(G_Q^d(r)) &= (2p+1)^d + (r-1)(2^d - 1) * p^d \\ &= \Theta(2^d r p^d) \end{aligned} \tag{4.3}$$

If $p$ is constant, the number of variables will be linearly proportional to the refinement level $r$. A visual explanation of how the number of basis functions grow has been shown on the Figure 4.4.

### 4.2.1. Time complexity

To calculate the time complexity, we will use the element partition tree method of ordering generation. The element partition tree can be built by recursively removing the layer of $2^d - 1$ least refined (or largest) elements. The structure of such an element partition tree is shown in the Figure 4.5.

Let us denote each layer by $s_i$, where is $i$ is the number of the shell, counting from the most refined one, $s_0$ being the special case of the single element in the corner. It is easy to see that the whole tree is recursive and that each layer $s_i$ where $i > 0$ contains $2^d - 1$ elements.

During the solution process at each layer $s_i$ ($i \geq 1$), we wil first eliminate all the rows that correspond to the nodes with support entirely within $s_i$ (denoted by $If(s_i)$), and then eliminate all rows that correspond to nodes

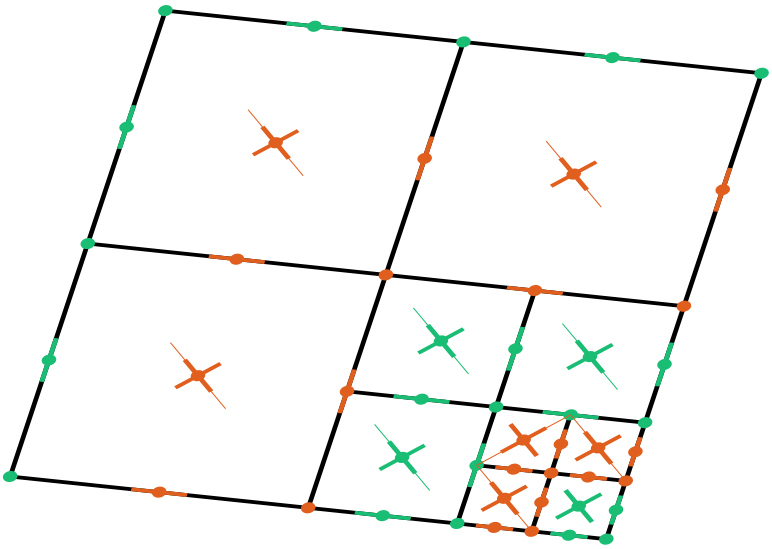


Figure 4.4: Each layer of refinement adds $2^d - 1$ elements and each element adds $p^d$ nodes.

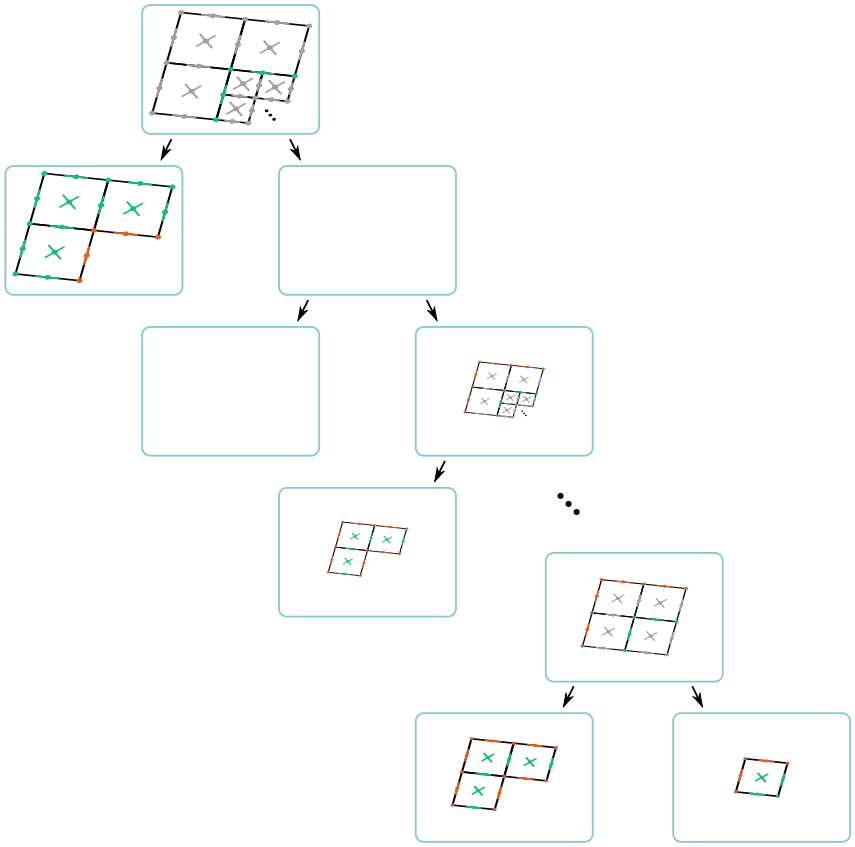


Figure 4.5: Corner point singularity element partition tree in 2D. Green nodes are eliminated at given partition tree node, gray nodes are eliminated deeper in the tree, and red nodes are eliminated in the ancestor of the tree node.

on the interface between $s_{i-1}$ and $s_i$ (denoted $If(s_{i-1}, s_i)$). This results result at the following cost formula for a single layer (excluding layers $s_0$, $s_1$ and $s_r$):

$$\begin{aligned} T(s_i) =& C_r(|If(s_i)|, |If(s_i)| + |If(s_{i-1}, s_i)| + |If(s_i, s_i + 1)|) \\ &+ C_r(|If(s_{i-1}, s_i)|, |If(s_{i-1}, s_i)| + |If(s_i, s_i + 1)|) \\ =& |If(s_i)|(|If(s_i)| + 2|If(s_{i-1}, s_i)|)^2 + 4|If(s_{i-1}, s_i)|^3 \end{aligned} \tag{4.4}$$

(4.5)

To calculate the number of nodes on the interface between layers, we can make an observation that this number stays the same from the moment given layer has been refined. In other words, the number of nodes on the interface between any layers is equal to the number of nodes on the interface between layer $s_0$ and $s_1$:

$$|If(s_{i-1}, s_i)| = (p+1)^d - p^d \tag{4.6}$$

Similarly we can calculate the number of internal nodes, making the observation that it is equal to the number of nodes in a mesh of refinement level 1 ($G^d(1)$) excluding the nodes on the (hyper)faces not touching the corner with singularity and minus amount of nodes in a mesh of refinement level 0 ($G^d(0)$).

$$|If(s_i)| = (2^d-1)p^d - |If(s_{i-1}, s_i)| = (2^d-1)p^d + p^d - (p+1)^d = (2p)^d - (p+1)^d \tag{4.7}$$

Thus the cost of elimination of a single layer $s_i$, where $1 < i < r$, goes as follows:

$$\begin{aligned} T(s_i) &= |If(s_i)|(|If(s_i)| + 2|If(s_{i-1}, s_i)|)^2 + 4|If(s_{i-1}, s_i)|^3 \\ &= ((2p)^d - (p+1)^d)((2p)^d - (p+1)^d + (p+1)^d - p^d)^2 + 4((p+1)^d - p^d)^3 \\ &= ((2p)^d - (p+1)^d)((2^d-1)p^d)^2 + 4((p+1)^d - p^d)^3 \\ &\leq (2p)^d((2p)^d)^2 + 4(p+1)^{3d} = (2p)^{3d} + 4(p+1)^{3d} \\ &= \mathcal{O}(2p)^{3d} \end{aligned} \tag{4.8}$$

The computational cost of the whole mesh will then be as follows:

$$T(G^d(r)) = \mathcal{O}(r(2p)^{3d}) = \mathcal{O}((2p)^{2d} \cdot 2^d r p^d)\mathcal{O}(2^{2d}p^{2d}N) \tag{4.9}$$

With the assumption that the parameter $p$ is constant and considering that dimension $d$ is constant for a problem, we arrive at a linear time complexity ($\mathcal{O}(N)$) of the whole algorithm.

## 4.3. Quasi-optimal point singularity meshes

To generalize the analysis above, we will present a set of properties of the mesh that guarantee that it is possible to create an ordering that leads to linear solver execution time. We will call a mesh that fulfills those properties as *quasi-optimal point singularity mesh*. The properties of *quasi-optimal point singularity mesh* are the following:

1. The mesh can be split into no more than $Kr$ consecutive layers, where $r$ is the refinement level of the mesh and $K$ is some constant.
2. Each basis function can be assigned to one of the layers in such a way that a pair of basis functions will not have overlapping supports if they are more than $M$ layers apart, where $M$ is some constant;
3. Each layer will have no more than $L$ basis functions assigned, where $L$ is some constant;

If those properties are met, it is possible to create an ordering in which the nodes are eliminated layer by layer. Elimination of each consecutive layer will require no more than $C_r(L, L*(M+1)) = O(L^3M^2)$ subtractions and the whole solver will require no more than $\mathcal{O}(KrL^3M^2)$ operations. The constructed mesh will have $N = \mathcal{O}(KrL)$ nodes, so the time complexity of the solver can be stated as $\mathcal{O}(N(LM)^2)$. As $K$, $L$ and $M$ are constants, we can remove them from the big-$\mathcal{O}$ notation and the resulting solution cost will be $\mathcal{O}(N)$, i.e. it grows linearly with the number of nodes (which number in turn grows linearly with the refinement level).

## 4.4. Quasi-optimality of $h$-adaptive point singularity meshes

A final step of the proof presented in this section is to show that any $h$-adaptive point singularity mesh created with Algorithm 1 will be quasi-optimal. It is easy to notice that:

1. The elements can be grouped into $R$ layers by their refinement level and each basis function can be assigned to the layer of one of its elements.
2. Because of the 1-*irregularity rule*, two elements sharing a vertex cannot differ by more than two refinement levels, so a single basis function cannot span over two elements with the difference of more than two refinement levels.
3. There will be no more than $4^d$ elements of each refinement level. At the same time, each element will have no more than $(p+1)^d$ basis functions – both values are constant.

Those observations lead to the conclusion that any point singularity $h$-adaptive mesh will be quasi-optimal:

1. There are exactly $R$ layers.
2. Nodes that are more than two layers apart will never overlap.
3. There will be no more than $4^d(p+1)^d$ nodes at each layer.

This way, we have proven that it is possible to solve a problem on a mesh with point singularity in linear time complexity ($\mathcal{O}(N)$).

It is also easy to extend those observations and see that a mesh refined towards any finite number of point singularities is quasi-optimal in the same way (each layer will potentially have the number of basis functions multiplied by the number of singularities).

Chapter 5

# Computational complexity for multi-dimensional flat singularity meshes

This section analyzes the time complexity of direct solvers being run over adaptive meshes refined over singularities of simple shapes of higer dimensionality. In particular, we analyze meshes with singularities in shape of lines, planes/faces and hyperplanes/hyperfaces with three or more dimensions, depending on the dimensionality of the space. In this case, for the simplicity of derivation, we will ignore the polynomial order $p$ factor.

## 5.1. Structure of the mesh with singularity

In this section, we will analyze $d$-dimensional meshes refined towards $q$-dimensional singularities. Such mesh will be considered refined until refinement level $r$ towards that singularity if all elements that overlap any part of the singularity are refined to refinement level $r$. Such refinement can again be achieved using Algorithm 1. A process of such refinement is shown in Figure 5.1 and 5.2, respectively for edge and face singularity in 3-dimensional space. It is important to notice that regular meshes can also be analyzed as mesh refined toward $d$-dimensional singularity in $d$-dimensional space.

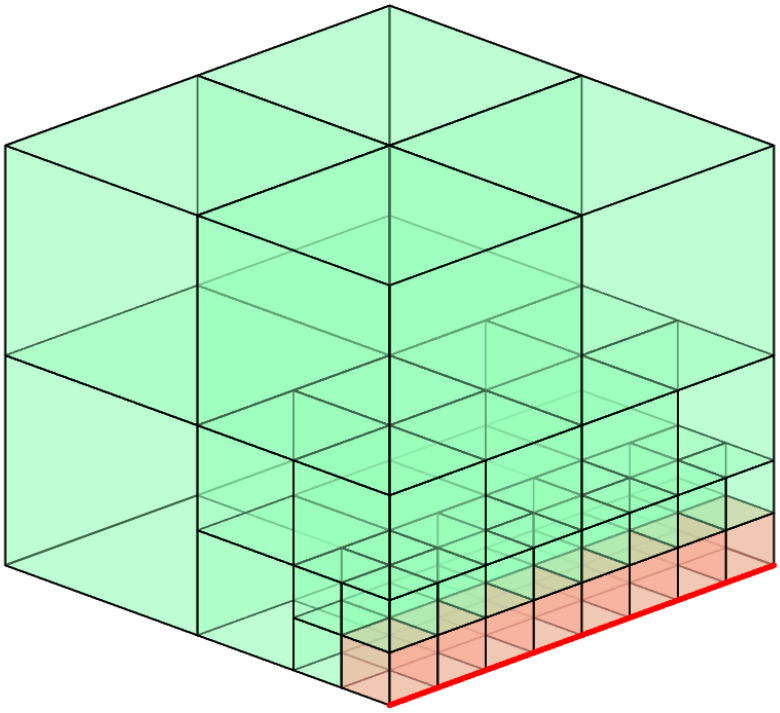


Figure 5.1: Example edge singularity mesh construction in 3D

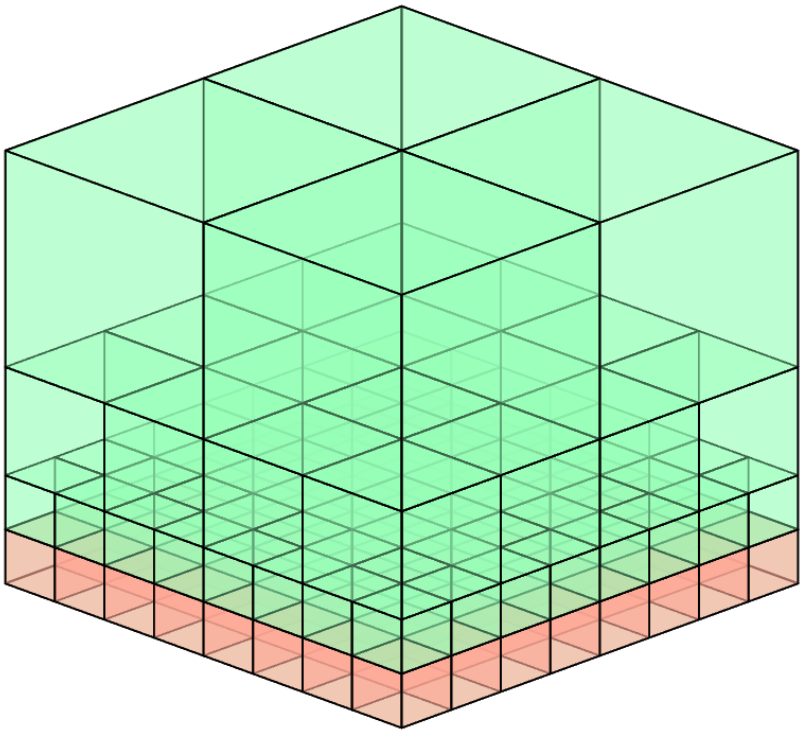


Figure 5.2: Example face singularity mesh construction in 3D

## 5.2. Analysis: singularity placed on the boundary of the mesh

For the sake of simplicity, we will start by analyzing the basic case of the singularity placed on the boundary of the whole $h$-adapted mesh. Let us denote the dimensionality of the mesh as $d$, the dimensionality of the singularity as $q$ and the singularity itself as $S_q$. The mesh will be denoted as $\mathcal{G}^d_{S_q}(r)$, where $r$ is its refinement level.

### 5.2.1. Properties of the mesh

The number of elements in such a singularity mesh can be calculated using the following recursive formula:

$$\begin{aligned} &N_e(\mathcal{G}^d_{S_q}(0)) = 1 \\ &N_e(\mathcal{G}^d_{S_q}(r)) = N_v(\mathcal{G}^d_{S_q}(r-1)) + 2^{q(r-1)} * (2^d - 1) \\ &N_e(\mathcal{G}^d_{S_q}(r)) = \begin{cases} (2^d-1)r + 1 & \text{if } q = 0 \\ \frac{(2^d-1)2^{rq} - (2^d - 2^q)}{2^q - 1} & \text{if } q \geq 1 \end{cases} \end{aligned} \tag{5.1}$$

The formula expands to the following values:

| | Point ($q=0$) | Edge ($q=1$) | Face ($q=2$) | Hyperface ($q=3$) |
|---|---|---|---|---|
| 1-D | $r+1$ | $2^r$ | | |
| 2-D | $3r+1$ | $3(2^r)-2$ | $4^r$ | |
| 3-D | $7r+1$ | $7(2^r)-6$ | $\frac{1}{3}(7(4^i)-4)$ | $8^r$ |
| 4-D | $15r+1$ | $15(2^r)-14$ | $\frac{1}{3}(15(4^i)-12)$ | $\frac{1}{7}(15(8^i)-8)$ |
| $d$-D | $(2^d-1)r+1$ | $(2^d-1)2^r-(2^d-2)$ | $\frac{1}{3}((2^d-1)4^i-(2^d-4))$ | $\frac{1}{7}((2^d-1)8^i-(2^d-8))$ |

Table 5.1: Number of elements for each mesh and singularity dimension

For $q \geq 1$ the number of elements can be approximated by the following lower and upper bounds:

$$2^{(d-1)}(2^q)^r \leq N_e(\mathcal{G}^d_{S_q}(r)) \leq 2^d(2^q)^r \tag{5.2}$$

The number of variables can be estimated to be between $p^d$ and $(p+1)^d$ per element, this leads us to the following approximation for $q \geq 1$:

$$2^{(d-1)}p^d(2^q)^r \leq N_v(\mathcal{G}^d_{S_q}(r)) \leq 2^d(p+1)^d(2^q)^r \tag{5.3}$$

For set $d$, $q \geq 1$ and $p$, the approximations lead to the following formulas:

$$N_e(\mathcal{G}^d_{S_q}(r)) = \Theta((2^q)^r) \tag{5.4}$$

$$N_v(\mathcal{G}^d_{S_q}(r)) = \Theta((2^q)^r) \tag{5.5}$$

In other words, both the number of elements and variables grow proportionally to $\mathcal{O}(2^{qr})$ and that this growth speed (understood in terms of $\mathcal{O}$-notation) depends only on the dimensionality of the singularity $q$, not the dimensionality of the mesh $d$.

### 5.2.2. Time complexity of a solution with singularity built on mesh boundary

To analyze the time complexity of the solver, we can again use the element partition tree approach. The element partition tree can be built using the following recursive procedure:

1. Create a root node of the element partition tree and attach all the elements to that node.
2. If there is just one element, finish the procedure, and the root node is the sole node of the returned tree.
3. Create a child node of the root node containing all the least refined elements.
4. Divide the other elements by $q$ parallel planes perpendicular to the singularity and parallel to the boundaries of the mesh (let us denote those planes as *dividing planes*), crossing the midpoint of the singularity. This will create $2^q$ submeshes.
5. For each submesh generated above, run this procedure recursively and attach the resulting trees as subtrees of the second child node.
6. Finish the procedure and return the tree stemming from the root node.

An example of such an element partition tree is shown in Figure 5.3. We denote by $s_i$ the element partition tree nodes from the refinement level $i$.

Even though the elimination tree nodes on each level have an analogous set of elements, the order of elimination will differ slightly for the nodes that contain elements on the boundary of the mesh other than the boundary containing the singularity. To simplify the analysis, we will modify the order of elimination slightly so that those nodes behave similarly to the others: the variables corresponding to the basis functions on the boundary of the mesh will be eliminated at the root node $s_r$. This change increases the computation time slightly. However, it has no impact on the time complexity.

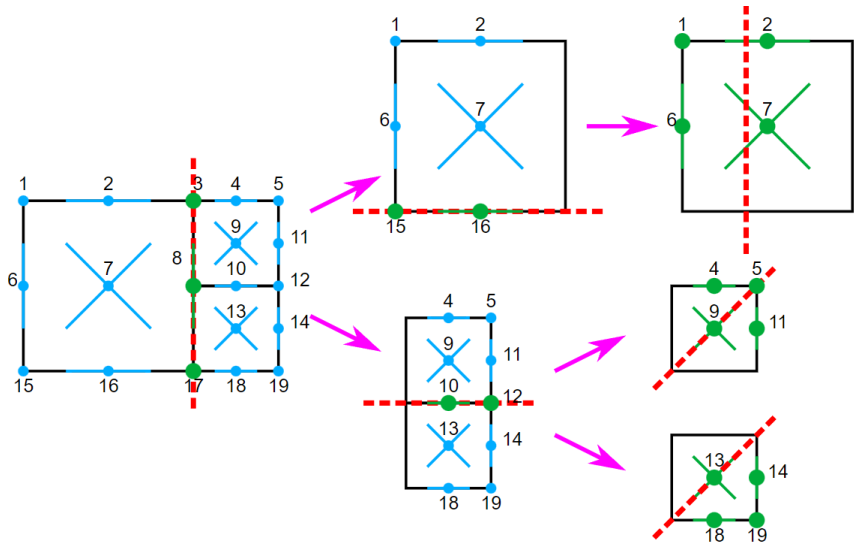



Figure 5.3: Example partition tree for one-dimensional boundary singularity in 2-D.

To calculate the computational cost of the solver using the ordering generated from that element partition tree, we will need to know two values for each element partition tree node:

– The number of variables removed in that element partition tree node, or $n_r$. For the $s_i$ nodes, this number is proportional to the number of elements of that node that are touching the dividing planes that are used to further divide the submesh.

– The total number of variables with support over the elements in this subtree, or $n_t$. For the $s_i$ nodes, this number is proportional to the number of elements that are on the dividing planes of the ancestral tree nodes.

It is not difficult to see that the cross-section of the mesh through the dividing planes looks like a mesh in space of one less dimension built over a singularity of dimensionality one less than the original one.

Thus for $q > 1$, the following relationships are true:

$$n_r(s_i) = \mathcal{O}(2^{(q-1)r}q) = \mathcal{O}(2^{(q-1)r}) \tag{5.6}$$
$$n_e(s_i) = \mathcal{O}(2^{(q-1)r}q) = \mathcal{O}(2^{(q-1)r}) \tag{5.7}$$

All the other nodes have a $\mathcal{O}(1)$ number of elements.

Thanks to those observation, we can calculate the time complexity of running the solver for $q > 1$ using the following equation:

$$\begin{aligned}
T(s_0) &= \mathcal{O}(1) \\
T(s_r) &= 2^q T(s_{r-1}) + \mathcal{O}(C_e(n_r, n_r + n_e)) + \mathcal{O}(1) \\
&= 2^q T(s_{r-1}) + \mathcal{O}(2^{3(q-1)r}) \\
&= \mathcal{O}(\sum_{h=0}^{r}(2^{q(r-h)} \cdot 2^{3(q-1)h})) \\
&= \mathcal{O}(\sum_{h=0}^{r}(2^{q(r-h)+3(q-1)h})) \\
&= \mathcal{O}(\sum_{h=0}^{r}(2^{q(r+2h)-3h}) \\
&= \mathcal{O}((2^{3(q-1)})^r) \qquad (5.8) \\
T(\mathcal{G}^d_{S_q}(r)) &= \mathcal{O}((2^{3(q-1)})^r) \qquad (5.9)
\end{aligned}$$

For $q = 1$, the analogous calculations go as follows:

$$n_r(s_i) = \mathcal{O}(r) \tag{5.10}$$
$$n_e(s_i) = \mathcal{O}(r) \tag{5.11}$$

And the time complexity of running the solver will follow this equation:

$$\begin{aligned}
T(s_0) &= \mathcal{O}(1) \\
T(s_r) &= 2T(s_{r-1}) + \mathcal{O}(C_e(n_r, n_r + n_e)) + \mathcal{O}(1) \\
&= 2T(s_{r-1}) + \mathcal{O}(r^3) \\
&= \mathcal{O}(\sum_{h=0}^{r}(2^{r-h} \cdot h^3)) \\
&= \mathcal{O}(2^r \cdot 1^3 + 2^{r-1} \cdot 2^3 + \cdots + 2^1 \cdot (r-1)^3 + 1 \cdot r^3) \\
&= \mathcal{O}(2^r) \\
T(\mathcal{G}^d_{S_1}(r)) &= \mathcal{O}(2^r) \qquad (5.12)
\end{aligned}$$

Considering that the number of variables $N_v(\mathcal{G}^d_{S_q}(r)) = \Theta((2^q)^r)$, we can calculate the time complexity as a fuction of number of variables for $q \geq 1$ to be:

$$T(\mathcal{G}^d_{S_q}(r)) = \Theta(N_v^{3\frac{q-1}{q}}) \tag{5.13}$$

We sum up the analysis in the table 5.2.

| | Variables | Operations | Operations in $N_v$ |
|---|---|---|---|
| Point singularity | $\Theta(r)$ | $\mathcal{O}(r)$ | $\mathcal{O}(N_v)$ |
| Edge singularity | $\Theta(2^r)$ | $\mathcal{O}(2^r)$ | $\mathcal{O}(N_v)$ |
| Face singularity | $\Theta(4^r)$ | $\mathcal{O}(8^r)$ | $\mathcal{O}({N_v}^{\frac{3}{2}})$ |
| Hyperface (3-D) singularity | $\Theta(8^r)$ | $\mathcal{O}(64^r)$ | $\mathcal{O}({N_v}^2)$ |
| 4-D singularity | $\Theta(16^r)$ | $\mathcal{O}(512^r)$ | $\mathcal{O}({N_v}^{2.25})$ |
| 5-D singularity | $\Theta(32^r)$ | $\mathcal{O}(4096^r)$ | $\mathcal{O}({N_v}^{2.4})$ |
| $q$-dimensional singularity | $\Theta((2^q)^r)$ | $\mathcal{O}((2^{3(q-1)})^r)$ | $\mathcal{O}({N_v}^{3\frac{(q-1)}{q}})$ |

Table 5.2: Number of elements for each mesh and singularity dimension

## 5.3. Quasi-optimal $q$-dimensional singularity meshes

To generalize the analysis from the previous section, we can observe that a broader class of meshes with singularities will follow the same time complexity – we will define those meshes as *quasi-optimal $q$-dimensional singularity meshes*.

A mesh will be a quasi-optimal $q$-dimensional singularity mesh if it has the following properties:

1. The basis functions of the mesh can be assigned to tree nodes of a full $K$-nary tree ($K \geq 2$) of height not larger than some $R = \lceil \log_K N \rceil + R'$, where $R'$ is some constant.
2. If a pair of basis functions have overlapping supports, they will be assigned to the same tree node, or one of them will be assigned to an ancestor of the tree node of the other one.
3. Each tree node will not have more than $Q \cdot K^{h\frac{q-1}{q}}$ (if $q \geq 2$) or more than $Q \cdot h$ (if $q \leq 1$) basis functions assigned, where $h$ is the height of the subtree that given node is the root of and $Q$ is some arbitrary positive constant. At the same time, the number of overlaps between basis functions belonging to that tree with basis functions of ancestor tree nodes will be limited by the same number.

If such a tree is created, we can use it to create an ordering that would follow the post-order traversal of that tree. If so, when $q \leq 1$, for $q$-dimensional singularity the cost of removing of all nodes belonging to a tree node $S_h$ with height $h$ is bounded by the following equation:

$$T(S_h) \leq C_r(Qh, 2Qh) = 4 \cdot Q^3 h^3 \tag{5.14}$$

The total cost of the execution of the solver will in turn be no more than:

$$\begin{aligned}
T(\mathcal{G}(r)) &\leq \sum_{h=1}^{R} K^{R-h} \cdot 4(Q^3 h^3) \\
&= 4Q^3 K^R \sum_{i=1}^{R} (K^{-h} h^3) \\
&= Q^3 K^R (\frac{1^3}{K^1} + \frac{2^3}{K^2} + \frac{3^3}{K^3} + \cdots + \frac{(R-1)^3}{K^(R-1)} + \frac{R^3}{K^R}) \\
&< Q^3 K^R \cdot 26 \\
T(\mathcal{G}(r)) &= \mathcal{O}(K^R)
\end{aligned}$$

$$= \mathcal{O}(N) \tag{5.15}$$

In the case of $q$-dimensional singularity with $q \geq 2$, the cost of removing of all nodes belonging to a tree node $S_h$ with height $h$ is bounded by the following equation:

$$T(S_h) \leq C_r(Q \cdot K^{h\frac{q-1}{q}}, 2Q \cdot K^{h\frac{q-1}{q}}) = 4Q^3 \cdot K^{3h\frac{q-1}{q}} \tag{5.16}$$

And the total cost of running the whole solver will be no more than:

$$\begin{aligned}
T(\mathcal{G}(r)) &= \sum_{h=1}^{R}(K^{R-h} \cdot 4Q^3(K^{3h\frac{q-1}{q}})) \\
&= 4Q^3 \sum_{h=1}^{R}(K^{R-h+3h\frac{q-1}{q}}) \\
&= 4Q^3 \sum_{h=1}^{R}(K^R K^{h\frac{3(q-1)-q}{q}}) \\
&= 4Q^3 \sum_{h=1}^{R}(K^R (K^{\frac{2q-3}{q}})^h) \\
&= 4Q^3 K^R \sum_{h=1}^{R}((K^{\frac{2q-3}{q}})^h) \\
&< 4Q^3 K^R (K^{\frac{2q-3}{q}})^{R+1} \\
T(\mathcal{G}(r)) &= \mathcal{O}(K^R (K^{\frac{2q-3}{q}})^R) \\
&= \mathcal{O}(K^{R+\frac{2q-3}{q}R}) \\
&= \mathcal{O}(K^{(1+\frac{2q-3}{q})R}) \\
&= \mathcal{O}(K^{3\frac{q-1}{q}R}) \\
&= \mathcal{O}(K^{3\frac{q-1}{q}\log_K N}) \\
&= \mathcal{O}((K^{\log_K N})^{3\frac{q-1}{q}}) \\
&= \mathcal{O}(N^{3\frac{q-1}{q}})
\end{aligned} \tag{5.17}$$

Together, both cases can be stated as:

$$T(\mathcal{G}(r)) = \mathcal{O}(N^{\min(3\frac{q-1}{q},1)}) \tag{5.18}$$

## 5.4. Quasi-optimality of $h$-adaptive meshes over singularities

To prove that every $h$-adaptive mesh adapted towards a flat singularity using the Algorithm 1 is quasi-optimal, we can propose the following tree generation algorithm:

1. Find a dividing plane that crosses the least amount of basis functions' supports out of all planes perpendicular to the singularity that divide the mesh in such a way that no more than half of all basis functions lay solely on either one of the sides of the plane. Create a tree node with all basis functions that the chosen plane crosses the support of and remove them from the mesh.
2. For each side of the plane take the basis functions on that side and recursively run the algorithm. The resulting trees become subtrees of the node created in the previous function.

3. Return the tree rooted in the node created in the first step.

Let us analyze if a tree generated that way proves the quasi-optimality of the mesh:

1. Every child of any node will have at most half of the basis functions of its parent. Because of that, the total height of the tree cannot be larger than $\lceil \log_2 N_v \rceil$ ($K = 2$).
2. There is no overlap between supports of basis functions of either side of the dividing plane.
3. Elements of refinement level $r$ cannot be produced farther than $2\sqrt{d}$ times the side of those elements. Thus the number of elements of a given refinement level that cross the dividing plane is limited.

   (a) In the case of $q = 1$, all the dividing planes are parallel to each other. The dividing plane of tree node $g$ nodes deep from the root will have a distance between the nearest planes of tree nodes higher in the gree of at most $2^{-g} \cdot L$, where $L$ is the length of the singularity. This means that no elements larger than $2^{-g} \cdot L$ will be eliminated at this node. In other words, while each dividing plane crosses at most some $Q$ elements of each refinement level ($Q$ is an arbitrary constant), the minimal refinement level of elements removed at given tree node increases by 1 every step down the tree. This means that the root node will have at most $Q \cdot R$ elements, and other nodes will have at most $Q \cdot h$ elements.

   (b) In case of $q \geq 2$, the limit of elements of refinement level $r$ crossed by a dividing plane is $\mathcal{O}(2^{r(q-1)} \cdot L)$, where $L$ is the length/area/volume of cross section between the singularity and the dividing plane (assuming that the whole mesh has side of length 1). Because the plane of the smallest cross-section is chosen, $L$ will decrease on average by $2^{\frac{q-1}{q}}$. The total number of elements crossed will be $\mathcal{O}(2^{r(q-1)} \cdot ((\frac{1}{2})^{\frac{q-1}{q}})^{R-h})$. Considering that $R = rq$ (up to a constant), this is equivalent to $\mathcal{O}(2^{r(q-1)} \cdot ((\frac{1}{2})^{\frac{q-1}{q}})^{rq-h}) = \mathcal{O}(2^{h\frac{q-1}{q}})$.

This shows that all meshes with flat singularities that are adapted using the Algorithm 1 are quasi-optimal.

Chapter 6

# Generalized computational complexity for arbitrary meshes

In previous chapters we have shown incremental approaches to the problem of finding the complexity of solvers for meshes with singularity. In particular, Chapter 4 covers simple case of point singularities and Chapter 5 extends the proofs for flat signularities. In this Chapter 6, we finally approach the problem of time complexity for $h$-adaptive mesh built over a closed singularity[1] of any well-behaved, non-fractal shape, using tools presented in previous Chapter 5. Examples of such meshes can be seen in Figure 6.2 and 6.1. In this Chapter, we consider only singularities of dimensionality $q \geq 1$, i.e. we omit point singularities, as those were covered in Chapter 4.

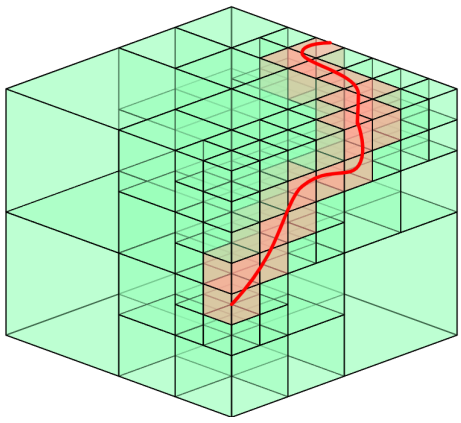


Figure 6.1: Three-dimensional mesh refined towards non-regular edge singularity ($q = 1, d = 3$).

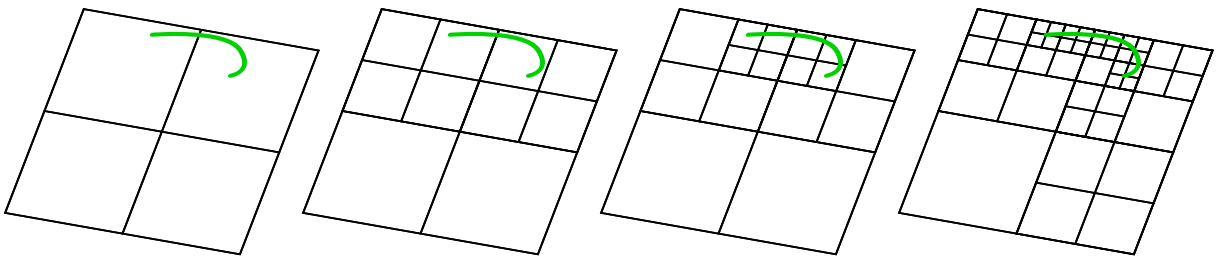


Figure 6.2: Two-dimensional mesh refined towards non-regular edge singularity ($q = 1, d = 2$).

[1] *closed singularity* – singularity that includes its boundary

## 6.1. Quasi-optimal $q$-dimensional singularity mesh properties

To prove that any proper non-fractal $h$-adaptive mesh built over singularity is quasi-optimal, let us first revisit the definition of *quasi-optimal $q$-dimensional singularity meshes* from Section 5.3. A mesh will be a quasi-optimal $q$-dimensional singularity mesh if it has the following properties:

1. *Logarithmic height.* The basis functions of the mesh can be assigned to tree nodes of a full $K$-nary tree ($K \geq 2$) of height not larger than some $R = \lceil \log_K N \rceil + R'$, where $R'$ is some constant.
2. *Proper elimination order.* If a pair of basis functions have overlapping supports, they will be assigned to the same tree node or one of them will be assigned to an ancestor of the tree node of the other one.
3. *Limited tree node size.* Each tree node will not have more than $Q \cdot K^{h\frac{q-1}{q}}$ (if $q \geq 2$) or more than $Q \cdot h$ (if $q \leq 1$) basis functions assigned, where $h$ is the height of the subtree that given node is the root of and $Q$ is some arbitrary positive constant. At the same time, the number of overlaps between basis functions belonging to that tree with basis fuctions of ancestor tree nodes will be limited by the same number.

Below, we describe an example method of constructing a tree that would meet the required properties and then we prove that the generated tree indeed matches the properties.

## 6.2. Quasi-optimal tree generation method

To create the tree as proscribed in the previous section, we can follow the following recursive procedure for a $q$-dimensional singularity in $d$-dimensional space.

1. *Choose a dividing plane.* Select a (hyper)plane that divides the singularity into equal halves in respect to their measure (i.e. length, area, volume, hypervolume, etc. or the count of points in case of point singularity). Of all possible planes, select one that has the smallest intersection with the singularity – that is minimizing the dimensionality of the intersection, and of all such intersections, selecting one with a minimal measure.
2. *Assign basis functions on the dividing plane.* Assign each unassigned basis function with support overlapping the dividing plane to a newly created tree node.
3. *Assign basis functions of unrefined elements.* In addition, assign to that tree node all basis functions with refinement level not higher than $\lfloor \frac{f}{q} \rfloor$, where $f$ is the current depth of the recursion (with $f = 0$ for the root execution of the procedure).
4. *Recurse.* If $\lfloor \frac{f}{q} \rfloor < R$, run the procedure for both parts of the space divided by the plane, taking into the account corresponding halves of the singularity and only the basis functions with supports in the parts of the space.
5. *Return a subtree.* Return a tree with the node created in Step 2 as root, and both trees returned from Step 4 as subtrees.

Please note, that the selection of cutting planes will be the same no matter the refinement level $R$.

## 6.3. Quasi-optimality of $h$-adaptive meshes

In this Section, we will prove that the generated tree fulfills the requirements from Section 6.1. The first two properties are relatively easy to prove, with the most effort required for the property of *limited node size*.

### 6.3.1. Logarithmic height

The recursion will end in $qR$ steps, thus the height of the tree is always $qR+1$. Because of the addition of Step 3, all basis functions corresponding to elements of refinement $R$ will be removed before the recursion is broken.

### 6.3.2. Proper elimination order

The property of proper elimination order is obvious, as long as the supports of each basis function are continuous (which is true in virtually every variant of Finite Element Method). In such a case, there can be no overlaps between basis functions that belong to different subtrees.

### 6.3.3. Limited tree node size

To prove this property, we need to analyze the properties of Algoritm 1. After each iteration $r$ of the main loop of the algorithm, the concatenated vector $G$ contains a proper mesh of refinement $r$. During refinement step $r$, only elements of refinement $r-1$ and $r-2$ can be refined. Elements of refinement $r-1$ will be refined if they overlap the singularity, and elements of refinement $r-2$ will be refined if they are neighbors of an element of refinement $r-1$ that is refined in that iteration. This allows for the following observation:

**Observation 6.3.1.** *An element of refinement $r$ ($0 < r < R$) in a correct mesh $h$-adapted over singularity $S$ can be created only if there is at least one point belonging to $S$ within a distance of two times the length of the diagonal of that element* (or $W$ times in the special case of $W$ divisions in each cardinal direction during each refinement).

From this observation, we can derive a second one:

**Observation 6.3.2.** *All elements of refinement $r$ ($0 < r < R$) in a correct mesh $h$-adapted over singularity $S$ will be contained within a distance of $3\sqrt{d}x$ (or $(W+1)\sqrt{d}x$ if $W \neq 2$) from $S$, where $x$ is the length of the edge of element of refinement $r$ and $d$ is the dimensionality of the space.*

This means that all elements of refinement level $r < R$ are contained within a thin shell around the singularity, thus their number will depend on the Kolmogorov dimension of the exterior of the singularity. A non-fractal shape will have exterior of Kolmogorov dimension one less than the Kolmogorov dimension of the shape. Thus:

**Observation 6.3.3.** *The number of elements of refinement $r$ ($r < R$) in a correct mesh $h$-adapted over non-fractal singularity $S$ is bounded by to $\mathcal{O}(2^{(q-1)r})$ (or $\mathcal{O}(W^{(q-1)r})$), where $q$ is the Kolmogorov dimension of the singularity $S$.*

By the definition of Kolmogorov dimension, we can make the following observations about elements of the highest refinement level $R$:

**Observation 6.3.4.** *The number of elements of highest refinement level $R$ is bounded by $\mathcal{O}(2^{qR})$ (or $\mathcal{O}(W^{qR})$), where $q$ is the Kolmogorov dimension of the singularity $S$.*

**Observation 6.3.5.** *A smallest cross-section that divides a singularity into two parts of equal measure will be proportional to the measure of the the singularity to the power of $\frac{q-1}{q}$, with a singularity of a shape of a (hyper-)ball being the worst case, as each considered cross-section then is equal in size. This means that the number of elements of highest refinement level $R$ on a dividing plane is bounded by $\mathcal{O}((2^{qR})^{\frac{q-1}{q}}) = \mathcal{O}(2^{(q-1)R})$, where $q$ is the Kolmogorov dimension of the singularity $S$.*

Considering those observations, we can derive some properties of a tree node $f$ steps deep in the tree ($f = 0$ being the tree root):

1. *For $q = 1$:*A dividing plane that is not parallel to the singularity will cross it in at most one point. For each refinement level only a limited and decreasing part of the plane will be within a radius of two element diagonals[2]. This means that there exists a constant, that for each refinement level and, the number of basis functions overlaping the dividing plane is guaranteed not to exceed it. Because the basis functions of low refinement level are removed in Step 3, the number of elements in each subtree will not exceed $Qh$, where $Q$ is some constant and $h$ is the height of that subtree.
2. *For $q > 1$:* The number of elements of refinement level $R$ on dividing plane is limited by $\mathcal{O}((2^{qR})^{\frac{q-1}{q}}) = \mathcal{O}(2^{(q-1)R})$ for the root node. The number of elements of lower refinement level $r$ cannot exceed $\mathcal{O}(2^{(q-1)r})$, thus the total number of elements is also bounded by $\mathcal{O}(\sum_{r=0}^{R} 2^{(q-1)r}) = \mathcal{O}(2^{(q-1)R})$ As the singularity is halved in each subtree, the number of elements will be limited by $\mathcal{O}(2^{qR-f})^{\frac{q-1}{q}}) = \mathcal{O}(2^{(q-1)R-f\frac{q-1}{q}})$ for all tree nodes. If we consider the height of the tree $H = qR + 1$, and height of each subtree $h = qR + 1 - f$, the upper bound of elements (and basis functions) can be stated as $\mathcal{O}(2^{(q-1)R-(qR+1-h)\frac{q-1}{q}}) = \mathcal{O}(2^{(q-1)R-(q-1)R+(h-1)\frac{q-1}{q}}) = \mathcal{O}(2^{(h-1)\frac{q-1}{q}})$.

This way, the *limited tree node size* property is met for each $q \geq 1$.

## 6.4. Practical conclusions

While this Chapter proves the time complexity of quasi-optimal solvers, the constants that can be neglected for the sake of theoretical time complexity, can be relatively large in practice. The method of generating elimination tree from this chapter will not produce the best results for most real life use cases and, most probably, existing solvers will be faster for most problems. There are a couple conclusions that should be taken into the account while constructing new methods though:

– If there are any singularity-like features in the problem, division of the mesh in the elimination process orthogonally (or near-orthogonally) to the singularity can significantly improve the running times. It might be worth to optimize elimination tree generation algorithms to recognize such features.

– Adaptation towards a singularity-like feature in an otherwise quite uniformly adapted mesh should have a significantly lower impact on the running time of the solver than an uniform adaptation. This means that given proper solver algorithm, for many simulation problems we could be more lax with the usage of the $h$-adaptation. In particular, it might be worth to consider adaptation strategies that generate more variables, but require lower solution cost.

[2]This would not be guaranteed if the singularity would not contain its boundary, as in that case infinitesimally small distances could be considered. Similar situation could happen if the singularity has any fractal, infinite or infinitesimal qualities.

Chapter 7

# Computational complexities of space–time formulations and time–marching schemes

We compare the computational cost of the direct solver executed over the $d$ dimensional space-time domain to the computational cost of the direct solver executed multiple times over $d$-1 dimensional meshes within the time-marching scheme. In Table 7.1 we list space-time adaptive meshes. We include the 4D space-time uniform mesh, 4D space-time mesh refined towards the hyperface singularity, 4D space-time mesh refined towards the face singularity, 4D space-time mesh refined towards edge singularity, as well as 3D space-time unfirm mesh, 3D space-time mesh refined towards face singularity, and 3D space-time mesh refined towards edge singularity. We compute dimensions $N$ of all these grids and express it by the number of refinement levels $r$. Additionally, we provide the estimates for the direct solver execution time. An interesting observation is that the computational cost of the direct solver executed for the space-time mesh refined towards an edge is linear $\mathcal{O}(N)$.

Next, in Table 7.2 we evaluate the computational cost for time-marching scheme. For each $d$-dimensional space-time refined toward the $q$-singularity, we construct a sequence of $2^r$ meshes. Each of the grids from the sequence is $d-1$ dimensional, and it is refined towards $q-1$ singularity. Such the sequence of grids provides the solution with similar accuracy to the space-time grid. We estimate the dimensions $n$ of the spatial meshes from the sequence and estimate the computational cost of the direct solver executed $2^r$ times, once for each grid from the sequence. An interesting observation is that the computational cost of the time-marching scheme corresponding to the space-time mesh refined towards an edge is $r$ times higher than the space-time cost. However, for all other space-time grids, the time marching scheme is cheaper.

| Space-time mesh | Space-time mesh size | Space-time mesh direct solver cost |
|---|---|---|
| 4D uniform | $N = \mathcal{O}(2^{4r})$ | $\mathcal{O}(N^{\frac{9}{4}}) = \mathcal{O}(2^{9r})$ |
| 4D hyperface | $N = \mathcal{O}(8^r) = \mathcal{O}(2^{3r})$ | $\mathcal{O}(N^2) = \mathcal{O}(2^{6r})$ |
| 4D face | $N = \mathcal{O}(4^r) = \mathcal{O}(2^{2r})$ | $\mathcal{O}(N^{\frac{3}{2}}) = \mathcal{O}(2^{3r})$ |
| 4D edge | $N = \mathcal{O}(2^r)$ | $\mathcal{O}(N) = \mathcal{O}(2^r)$ |
| 3D uniform | $N = \mathcal{O}(2^{3r})$ | $\mathcal{O}(N^2) = \mathcal{O}(2^{6r})$ |
| 3D face | $N = \mathcal{O}(4^r) = \mathcal{O}(2^{2r})$ | $\mathcal{O}(N^{\frac{3}{2}}) = \mathcal{O}(2^{3r})$ |
| 3D edge | $N = \mathcal{O}(2^r)$ | $\mathcal{O}(N) = \mathcal{O}(2^r)$ |

Table 7.1: Space-time formulations, mesh dimensions and direct solver costs

| Space-time mesh | Sequence of spatial meshes | Single spatial mesh size | Total direct solver cost for a sequence |
|---|---|---|---|
| 4D uniform $N = \mathcal{O}(2^{4r})$ | $2^r\times$ 3D uniform | $n = \mathcal{O}(2^{3r})$ | $2^r \times \mathcal{O}(n^2) = \mathcal{O}(2^{7r})$ |
| 4D hyperface $N = \mathcal{O}(2^{3r})$ | $2^r\times$ 3D face | $n = \mathcal{O}(2^{2r})$ | $2^r \times \mathcal{O}(n^{\frac{3}{2}}) = \mathcal{O}(2^{4r})$ |
| 4D face $N = \mathcal{O}(2^{2r})$ | $2^r\times$ 3D edge | $n = \mathcal{O}(2^r)$ | $2^r \times \mathcal{O}(n) = \mathcal{O}(2^{2r})$ |
| 4D edge $N = \mathcal{O}(2^r)$ | $2^r\times$ 3D point | $n = \mathcal{O}(r)$ | $2^r \times \mathcal{O}(r) = \mathcal{O}(r2^r)$ |
| 3D uniform $N = \mathcal{O}(2^{3r})$ | $2^r\times$ 2D uniform | $n = \mathcal{O}(2^{2r})$ | $2^r \times \mathcal{O}(n^{\frac{3}{2}}) = \mathcal{O}(2^{4r})$ |
| 3D face $N = \mathcal{O}(2^{2r})$ | $2^r\times$ 2D edge | $n = \mathcal{O}(2^r)$ | $2^r \times \mathcal{O}(2^r) = \mathcal{O}(2^{2r})$ |
| 3D edge $N = \mathcal{O}(2^r)$ | $2^r\times$ 2D point | $n = \mathcal{O}(r)$ | $2^r \times \mathcal{O}(r) = \mathcal{O}(r2^r)$ |

Table 7.2: Time-marching schemes corresponding to space-time formulations

## 7.1. Computational costs of iterative solvers for space-time formulations and time-marching schemes

Finally, we focus on the iterative solver. The computational costs of the iterative solver for both space-time grids and the time-marching scheme is estimated as in Table 7.3. The computational cost for the space-time grid is equal to $\mathcal{O}(N_{iter} \times N)$ where $N$ is the size of the space-time grid, and $N_{iter}$ is the number of iterations of the iterative solver for the space-time grid. The computational cost for the space-time grid is equal to $\mathcal{O}(2^r \times n_{iter} \times n)$ where $2^r$ is the number of steps of the time-marching scheme, $n$ is the size of the spatial grid from the sequence, and $n_{iter}$ is the number of iterations on the spatial grid. It is obvious to assume that $N_{iter} >> n_{iter}$, but the exact numbers of iterations are problem dependent. The computational cost of the iterative solver for space-time grids is higher than the computational cost of iterative solver for time-marching scheme. An interesting case is the space-time grid refined to the space-time edge, following the trajectory of the point object. In this case, the cost of iterative solver for space-time grid is $\mathcal{O}(N_{iter} \times N) = \mathcal{O}(N_{iter} \times 2^r)$ while the cost of the iterative solver for time marching grids is $\mathcal{O}(2^r \times n_{iter} \times n) = \mathcal{O}(n_{iter} \times r2^r)$. The iterative solver for space-time grid is cheaper than for the time marching scheme, if $N_{iter} < r \times n_{iter}$.

| Space-time mesh | Sequence of spatial meshes | Space-time iterative solver cost | Time-marching scheme total iterative solver cost |
|---|---|---|---|
| 4D uniform $N = \mathcal{O}(2^{4r})$ | $2^r\times$ 3D uniform | $\mathcal{O}(N_{iter} \times 2^{4r})$ | $\mathcal{O}(2^r \times n_{iter} \times 2^{3r}) = \mathcal{O}(n_{iter} \times 2^{4r})$ |
| 4D hyperface $N = \mathcal{O}(2^{3r})$ | $2^r\times$ 3D face | $\mathcal{O}(N_{iter} \times 2^{3r})$ | $\mathcal{O}(2^r \times n_{iter} \times 2^{2r}) = \mathcal{O}(n_{iter} \times 2^{3r})$ |
| 4D face $N = \mathcal{O}(2^{2r})$ | $2^r\times$ 3D edge | $\mathcal{O}(N_{iter} \times 2^{2r})$ | $\mathcal{O}(2^r \times n_{iter} \times 2^{r}) = \mathcal{O}(n_{iter} \times 2^{2r})$ |
| 4D edge $N = \mathcal{O}(2^{r})$ | $2^r\times$ 3D point | $\mathcal{O}(N_{iter} \times 2^{r})$ | $\mathcal{O}(2^r \times n_{iter} \times r) = \mathcal{O}(n_{iter} \times r2^{r})$ |
| 3D uniform $N = \mathcal{O}(2^{3r})$ | $2^r\times$ 2D uniform | $\mathcal{O}(N_{iter} \times 2^{3r})$ | $\mathcal{O}(2^r \times n_{iter} \times 2^{2r}) = \mathcal{O}(n_{iter} \times 2^{3r})$ |
| 3D face $N = \mathcal{O}(2^{2r})$ | $2^r\times$ 2D edge | $\mathcal{O}(N_{iter} \times 2^{2r})$ | $\mathcal{O}(2^r \times n_{iter} \times 2^{r}) = \mathcal{O}(n_{iter} \times 2^{2r})$ |
| 3D edge $N = \mathcal{O}(2^{r})$ | $2^r\times$ 2D point | $\mathcal{O}(N_{iter} \times 2^{r})$ | $\mathcal{O}(2^r \times n_{iter} \times r) = \mathcal{O}(n_{iter} \times r2^{r})$ |

Table 7.3: Comparison of costs of iterative solvers for space-time formulations and time-marching schemes.

Additionally, we compare the computational cost of iterative solver executed over the $d$ dimensional space-time domain to the computational cost of the iterative solver executed multiple times over the $d$-1 dimensional meshes during the time-marching scheme.

### 7.1.1. Impact of polynomial order of approximation

Notice that we haven't included the $p$-factor in the computational cost estimates for the arbitrary shape of singularity. However, we can easily estimate the computational cost of the static condensation performed at the beginning of the computations with a higher-order finite element method with hierarchical basis functions.

The computational cost of the static condensation over a single element is equal to $\mathcal{O}(p^{3d})$. We are eliminating the degrees of freedom from the interior of the element; we have $(p-1)^d$ degrees of freedom there; the matrix is dense, so the cost of elimination is the cube of the number of degrees of freedom. In other words, the cost of static condensation for a grid with $N$ elements is $\mathcal{O}(Np^{3d})$. With this observation in mind, we can estimate the costs of static condensations for space-time and time-marching grids. It is illustrated in Table 7.4. Clearly, the cost of static condensations is always higher for the space-time mesh.

| Space-time mesh | Sequence of spatial meshes | Space-time cost of static condensation | Time-marching scheme cost of static condensation |
|---|---|---|---|
| 4D uniform $N = \mathcal{O}(2^{4r})$ | $2^r \times$ 3D uniform | $\mathcal{O}(p^{12}2^{4r})$ | $\mathcal{O}(2^r \times p^9 2^{3r})$ |
| 4D hyperface $N = \mathcal{O}(2^{3r})$ | $2^r \times$ 3D face | $\mathcal{O}(p^{12}2^{3r})$ | $\mathcal{O}(2^r \times p^9 2^{2r})$ |
| 4D face $N = \mathcal{O}(2^{2r})$ | $2^r \times$ 3D edge | $\mathcal{O}(p^{12}2^{2r})$ | $\mathcal{O}(2^r \times p^9 2^r)$ |
| 4D edge $N = \mathcal{O}(2^r)$ | $2^r \times$ 3D point | $\mathcal{O}(p^{12}2^r)$ | $\mathcal{O}(2^r \times p^9 r)$ |
| 3D uniform $N = \mathcal{O}(2^{3r})$ | $2^r \times$ 2D uniform | $\mathcal{O}(p^9 2^{3r})$ | $\mathcal{O}(2^r \times p^6 2^{2r})$ |
| 3D face $N = \mathcal{O}(2^{2r})$ | $2^r \times$ 2D edge | $\mathcal{O}(p^9 2^{2r})$ | $\mathcal{O}(2^r \times p^6 2^r)$ |
| 3D edge $N = \mathcal{O}(2^r)$ | $2^r \times$ 2D point | $\mathcal{O}(p^9 2^r)$ | $\mathcal{O}(2^r \times p^6 r)$ |

Table 7.4: Comparison of costs of static condensations for space-time formulations and time-marching schemes.

## 7.2. Practical conclusions

We conclude that the total time complexity of the direct solver for time marching scheme is usually cheaper than for the corresponding space-time formulations. The exception is the case where we model point object traveling in space. This can be read from the fourth rows in Tables 7.2 and 7.3. The fourth row in Table 7.2 corresponds to the complexity of the single execution of the direct solver on the computational space-time mesh refined towards space-time edge singularity, representing point object traveling in time. The fourth row in Table 7.3 represents complexity of several executions of the direct solvers on a sequence of spatial grids refined towwards point singularities. They are obtained by ślicing"the space-time mesh. The space the formulation for this case is cheaper than time marching scheme $\mathcal{O}(2^{2r}) < \mathcal{O}(r2^{2r})$.

However, if in our space-time mesh we start refining towards a two-dimensional manifold, either caused by “flapping wings” of the point object, or by a boundary layer, the space-time formulation becomes more expensive. This is illustrated in the third row in Table 7.2 for the space-time formulation, and in the third row in Table 7.3 for time marching schemes. This time the space-time formulations is more expensive than time-marching scheme $\mathcal{O}(2^{3r}) > \mathcal{O}(2^{2r})$.

Thus, we advocate the usage of time-marching schemes.

In the case of direct solvers, we advocate the elimination of rows of the matrix (matrix permutations) based on the ordering of variables obtained by the post-order transition of the quasi-optimal element partition tree. The algorithm for generation of the quasi-optimal element partition tree is described in Section 6.2. The ordering of mesh variables from the element partition tree is explained in Section 3.4. The algorithm for sparse factorization is illustrated in Section 3.5

For the iterative solvers, the practical conclusion remains the same. This is illustrated in Table 7.3. It is a natural assumption that the number of iterations of the iterative solver grows with problem size. Namely $n_{iter} << N_{iter}$ where the first one corresponds to the number of iterations of an iterative solver run on spatial mesh from a sequence of meshes, and the former one corresponds to the number of iterations of an iterative solver run on the space-time mesh. This time, space-time formulations are also more expensive, unless for the traveling point objects (forth row in Table 7.3), where we have $N_{iter} < r \times n_{iter}$ where $r$ is the refinement level.

Chapter 8

# Numerical results

In this section we provide numerical experiments for verification of the theoretical findings summarized in Tables 7.1-7.3. We employ our Octave codes generating the structure of $d$-dimensional computational grids with $q$-dimensional singularity, generated using $r$ refinement levels. For simplicity of implementation, we generate our matrices by looking at relations between finite elements. Rows and columns in matrices correspond to finite elements. Non-zero entries in a row mean that two elements, one related to the row, and one related to the column, are adjacent through a $d-1$ dimensional face. This way of generating matrices influences the computational cost constant, ignoring the polynomial order of approximation, but the dependence on $N$ equal here the number of elements is of the same order as if we include all the relations of the basis functions.

We run experiments using Octave on a Linux cluster node equipped with 2.4GHz processor with 64 GB of RAM. We are not able to factorize more than eight refinements for the face singularity in four dimensions and nine refinement levels for the face singularity in three dimensions because of a lack of memory during the factorization process. The comparisons of execution times for four-dimensional face singularity versus a sequence of three-dimensional edge singularities are presented in Table 8.1. The comparisons of execution times for three-dimensional face singularity versus a sequence of two-dimensional edge singularities are presented in Table 8.2. We employ AMD ordering and multi-frontal solver as implemented in the Octave, e.g., for the face singularity with nine refinement levels in three dimensions, we run our matrix generation script:

```
F9_3=Face(9,3);
N = 87381
```

We compute the AMD permutation

```
p=amd(F9_3);
```

and we plug it into the LU factorization, measuring the execution time

```
tic; lu(F9_3(p,p)); toc
Elapsed time is 46.2016 seconds.
```

We are aware that this ordering is different from the ordering proposed in our paper. Nevertheless, the results show up to one order of magnitude times faster execution times of time-marching schemes with edge singularities in comparison to one call for the space-time domain with the face singularity.

The comparisons of execution times for four-dimensional edge singularity versus a sequence of three-dimensional point singularities are presented in Table 8.3. This time with Octave solver, we can perform 15 refinements over the space-time mesh. The space-time mesh with multi-frontal solver is, in this case, faster than the time marching scheme, up to the 13 refinement level. With 14 or 15 refinements, the cost of processing the space-time mesh is higher than the cost of processing the time-marching solver.

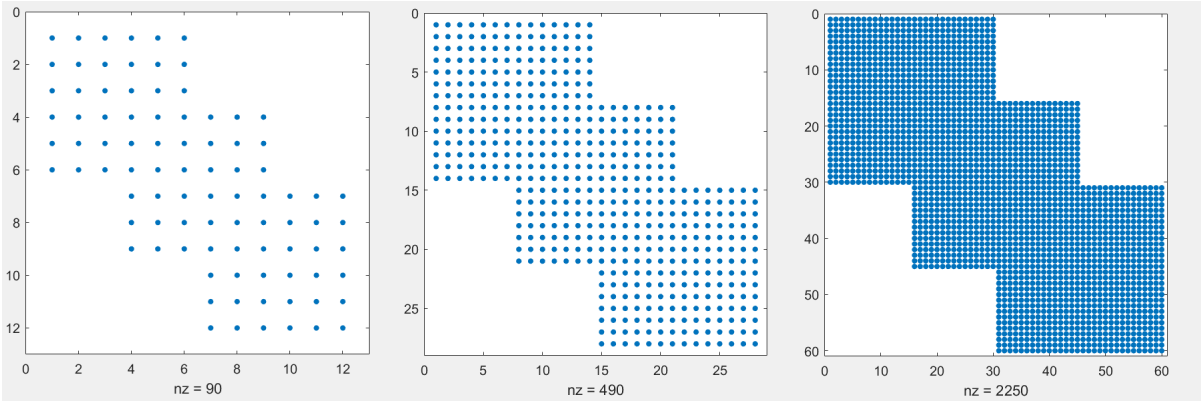



Figure 8.1: Structure of the matrix for two-, three-, and four- dimensional meshes refined towards point singularity with $r = 4$.

| Space-time mesh | r | N | Space-time solver time [s] | Sequence of spatial meshes | Time-marching scheme solver time [s] |
|---|---|---|---|---|---|
| 4D face | 6 | 4095 | 0.56 | $64\times$ 3D edge | $64 \times 0.0037 = 0.0236$ |
| 4D face | 7 | 16383 | 13.79 | $128\times$ 3D edge | $128 \times 0.024 = 3.072$ |
| 4D face | 8 | 65535 | 234 | $256\times$ 3D edge | $256 \times 0.063 = 16.128$ |

Table 8.1: Execution times for four dimensional mesh refined towards face singularity and the corresponding sequence of three-dimensional meshes refined towards edge singularities.

.

| Space-time mesh | r | N | Space-time solver time [s] | Sequence of spatial meshes | Time-marching scheme solver time [s] |
|---|---|---|---|---|---|
| 3D face | 6 | 1365 | 0.044 | $64\times$ 2D edge | $64 \times 0.0008 = 0.05$ |
| 3D face | 7 | 5461 | 0.28 | $128\times$ 2D edge | $128 \times 0.001 = 0.128$ |
| 3D face | 8 | 21845 | 2.55 | $256\times$ 2D edge | $256 \times 0.003 = 0.76$ |
| 3D face | 9 | 87381 | 46.20 | $512\times$ 2D edge | $512 \times 0.0074 = 3.78$ |

Table 8.2: Execution times for three dimensional mesh refined towards face singularity and the corresponding sequence of two-dimensional meshes refined towards edge singularities.

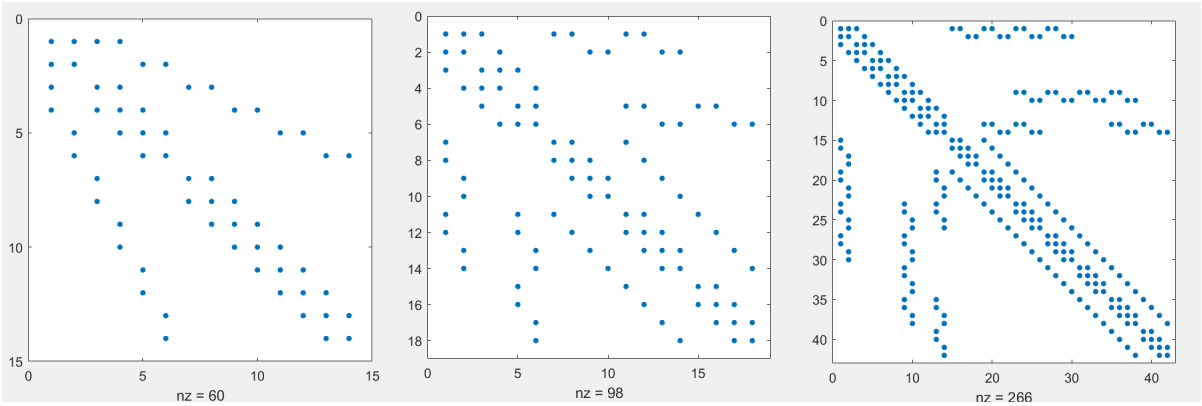



Figure 8.2: Structure of the matrix for two-, three-, and four- dimensional meshes refined towards edge singularity with $r = 3$.

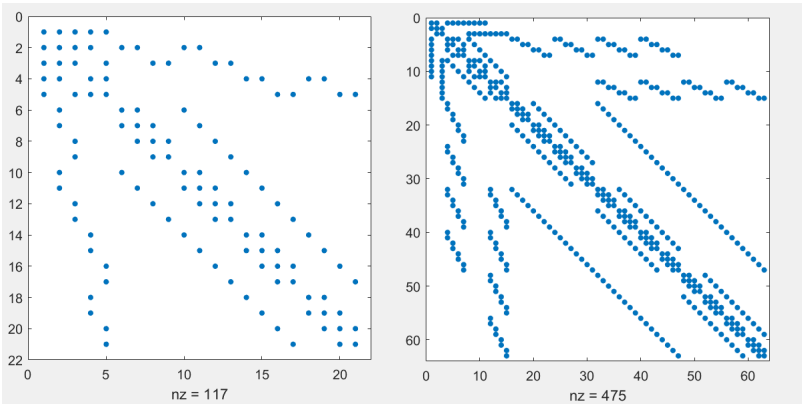



Figure 8.3: Structure of the matrix for three-, and four- dimensional mesh refined towards face singularity with $r = 3$.

| Space-time mesh | r | N | Space-time solver time [s] | Sequence of spatial meshes | Time-marching scheme solver time [s] |
|---|---|---|---|---|---|
| 4D edge | 6 | 434 | 0.008 | $64\times$ 3D point | $64 \times 0.0004 = 0.0256$ |
| 4D edge | 7 | 882 | 0.026 | $128\times$ 3D point | $128\times = 0004 = 0.0512$ |
| 4D edge | 8 | 1778 | 0.069 | $256\times$ 3D point | $256 \times 0.0005 = 0.128$ |
| 4D edge | 9 | 3570 | 0.15 | $512\times$ 3D point | $512 \times 0.0006 = 0.3$ |
| 4D edge | 10 | 7154 | 0.38 | $1024\times$ 3D point | $1024 \times 0.0006 = 0.61$ |
| 4D edge | 11 | 14332 | 1.01 | $2048\times$ 3D point | $2048 \times 0.0007 = 1.43$ |
| 4D edge | 12 | 14332 | 2.45 | $4096\times$ 3D point | $4096 \times 0.0008 = 3.27$ |
| 4D edge | 13 | 57330 | 5.52 | $8192\times$ 3D point | $8192 \times 0.0007 = 5.73$ |
| 4D edge | 14 | 114674 | 13.59 | $16384\times$ 3D point | $16384 \times 0.0007 = 11.46$ |
| 4D edge | 15 | 229362 | 38.96 | $32768\times$ 3D point | $32768 \times 0.0009 = 29.49$ |

Table 8.3: Execution times for four dimensional mesh refined towards edge singularity and the corresponding sequence of three-dimensional meshes refined towards point singularities.

.

Chapter 9

# Conclusions and future work

## 9.1. Computational complexities of direct solvers on hierarchically adapted grids

We have shown that for meshes hierarchically adapted towards singularities there exists an order of variable elimination that results in computational complexity of direct solvers not worse than $\mathcal{O}(\max(N, N^{3\frac{D-1}{D}}))$, where $N$ is the number of nodes and $D$ is the dimensionality of the singularity. This formula does not depend on the spatial dimensionality of the mesh. We have also shown the relationship between the time complexity and the Kolmogorov dimension of the singularity.

Additionally, we claim the following conjecture:

**Conjecture 9.1.1.** *For any set of points $S$ with Kolmogorov dimension $q \geq 1$ defined in Euclidean space with dimension $D$ and any point in that space $p$ there exists a cut of the space that divides the set $S$ into two parts of equal size, for which the intersection of that cut and the set $S$ has Kolmogorov dimension of $q-1$ or less.* Parts of equal size *for $q < D$ are to be meant intuivitely: as the size of covering boxes (as defined in the definition of Kolmogorov dimension) decreases to $0$, the difference between the number of boxes in both parts should decrease to $0$.*

It remains to be proven if the Conjecture 9.1.1 is true for well-behaved fractals. If so, then meshes built on well-behaved fractals of non-integer Kolmogorov dimension $q$ can be also solved with time complexity of $\mathcal{O}(N^{\max(3\frac{(q-1)}{q},1)})$. The proof of the conjecture is left for future work.

We can however illustrate the principle by the example of Sierpinski's triangle – a fractal of Kolmogorov dimension of $\frac{\log 3}{\log 2} = 1.58496250072116\ldots$. To build an elimination tree, we will divide the space into three roughly equal parts as shown in Figure 9.1. The Kolmogorov dimension of the boundary of such division is $0$, which is less than $\frac{\log 3}{\log 2} - 1$.

As the refinement level $R$ grows, the number of elements grows as $\mathcal{O}(2^{\frac{\log 3}{\log 2}R}) = \mathcal{O}(3^R)$ and as the elimination tree is ternary, the partition tree will have height of $\log_3 3^R + H_0 = \log_K N + H_0$. In addition, as the Kolmogorov dimension of the boundary of each elimination tree node is $0$, there are at most $\mathcal{O}(\log h)$ variables in each elimination tree node and on the overlap with nodes of ancestors is also limited by the same number. As this number is less than $\mathcal{O}(3^{h\frac{q-1}{q}})$, both conditions of *quasi-optimal mesh with q-dimensional singularity* are met, which means that it is possible to solve system build on a singularity of the shape of Sierpinski triangle in time not worse than $\mathcal{O}(N^{3\frac{\log 3/\log 2-1}{\log 3/\log 2}}) = \mathcal{O}(N^{1.10721\ldots})$.

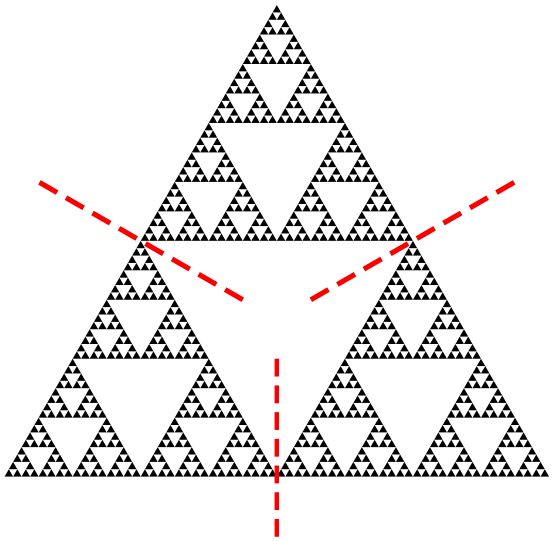


Figure 9.1: Division of the space for Sierpinski triangle.

## 9.2. Computational complexities of space-time formulations and time-marching schemes

To estimate the computational complexity of full space–time formulations and time–marching schemes for hypercubic elements, we simulate several possible scenarios for the resulting matrices when considering adaptivity toward singularities. In particular, we consider refinements towards the point, edge, face, and hyperface singularities over space–time mesh. In our idealized case, we refine all the elements that contain the prescribed point, edge, face or hyperface singularity. Thus, we obtain several representative refined $d$-dimensional computational meshes, where we assume that we perform refinements towards $q$-dimensional manifold ($q < d$) representing the singularities. For each of these representative meshes, we estimate the number of degrees of freedom (second column in Table 7.1), the computational complexity of the multi-frontal solver (third column in Table 7.1), the computational complexity of the iterative solver (third column in Table 7.3), and the computational complexity of the static condensation (third column in Table 7.4). On the other hand, we generated a sequence of refined $d-1$ dimensional computational meshes, representing the "cross-sectionsóf the $d$ dimensional space–time mesh. In this case, we performed refinements towards $q-1$-dimensional manifold representing the cross-section of the $q$-dimensional singularity.

*Our theoretical estimations and the numerical experiments imply that the time–marching scheme is usually faster than space-time formulations.*

This concerns the computational complexity of the multi-frontal solver (third column in Table 7.1 versus the fourth column in Table 7.2), the iterative solver (third column in Table 7.3 and fourth column in Table 7.3), and the static condensation (third and fourth column in Table 7.4). We also present numerical experiments, confirming the predicted theoretical behaviors.

We understand that our assumptions are the best possible idealistic scenarios. In the real life applications, the computational complexity of space–time formulation is more competitive due to

– Increased number of time steps in higher-order and accuracy time marching schemes, where the time-step size can be actually smaller than the temporal dimension of the smallest elements in the space–time mesh.

– Extensive parallelization of the computational process, where for example the static condensation for the space–time formulation can be performed fully in parallel, and the static condensation of the time–marching scheme has the limitation of the single size of the time-step mesh.

– Sequential nature of the time–marching scheme, where the iterative solver has to be executed in a sequence for each time-step mesh, and it cannot be parallelized once for the entire computational space–time mesh.

– The cost of generation of the refined computational meshes is in general ignored in our estimations (it is assumed to be linear), while in general, it is an iterative procedure that requires several solves, and in the space–time setup it can be performed once for the entire mesh, but in the time–marching scheme it has to be performed for each time-step mesh.

Nevertheless, our estimate constitutes the lower bounding case of the computational complexities for both space–time and time–marching schemes.

# List of figures

# List of tables

# Bibliography


[1] H. AbouEisha, V. M. Calo, K. Jopek, M. Moshkov, A. Paszyńska, M. Paszyński, Bisections-Weighted-by-Element-Size-and-Order Algorithm to Optimize Direct Solver Performance on 3D hp-adaptive Grids, Lecture Notes in Computer Science, 10861 (2018) 760-772

[2] P. R. Amestoy, T. A. Davis, I. S. Du, An Approximate Minimum Degree Ordering Algorithm, SIAM Journal of Matrix Analysis & Application, 17, 4 (1996) 886-905.

[3] Amestoy P. R., Duff I. S., L'Excellent J.-Y., 2000: Multifrontal parallel distributed symmetric and unsymmetric solvers. Computer Methods in Applied Mechanics and Engineering 184, 501-520

[4] Amestoy P. R., Duff I. S., Koster J. and L'Excellent J.-Y., 2001: A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM Journal of Matrix Analysis and Applications, 23, 1, 15-41

[5] Amestoy P. R., Guermouche A., L'Excellent J.-Y., Pralet S., 2006: Hybrid scheduling for the parallel solution of linear systems. Parallel Computing, 32, 2, 136-156

[6] D. N. Arnold, R. S. Falk, and R. Winther. Multigrid in h(div) and h(curl). Numerische Mathematik, 85:197–217, 2000.

[7] I. Babuška, B.Q. Guo. The h, p and h-p version of the finite element method: basis theory and applications. *Advances in Engineering Software*, Volume 15, Issue 3-4, 1992.

[8] Y. Bazilevs, C. Michler, V. M. Calo, and T. J. R. Hughes. Isogeometric variational multiscale modeling of wall-bounded turbulent flows with weakly enforced boundary conditions on unstretched meshes. Computer Methods in Applied Mechanics and Engineering, 199:780–790, 2010.

[9] A. Buffa, H. Harbrecht, A. Kunoth, and G. Sangalli. Bpx-preconditioning for isogeometric analysis. Computer Methods in Applied Mechanics and Engineering, 265:63–70, 2013.

[10] V. M. Calo, H. Gomez, Y. Bazilevs, G. Johnson, and T. J. R. Hughes. Simulation of engineering applications using isogeometric analysis. Proceedings of Tera Grid, 2008.

[11] N. Collier, L. Dalcin, D. Pardo, and V. M. Calo. The cost of continuity: Performance of iterative solvers on isogeometric finite elements. SIAM Journal on Scientific Computing, 35:A767–A784, 2013.

[12] V. M. Calo, N. Collier, D. Pardo, M. Paszyński, Computational complexity and memory usage for multi-frontal direct solvers used in p finite element analysis, Procedia Computer Science 4 (2011) 1854-1861

[13] O. Castillo-Reyes, A. Amor-Martin, A.Botella, P. Anquez, L. E. García-Castillo, Tailored meshing for parallel 3D electromagnetic modeling using high-order edge elements, Journal of Computational Science, 63 (2022) 101813

[14] O. Castillo-Reyes, D. Modesto, P. Queralt, A. Marcuello, J. Ledo, A. Amor-Martin,J. de la Puente, L. E. García-Castillo, 3D magnetotelluric modeling using high-order tetrahedral nédélec elements on massively parallel computing platforms Computers & Geosciences, 160 (2022) 105030

[15] L. Demkowicz, (2006). Computing with hp-Adaptive Finite Elements, Vol. I. Two Dimensional Elliptic and Maxwell Problems. Chapman and Hall/Crc Applied Mathematics and Nonlinear Science.

[16] Demkowicz, L., Kurtz, J., Pardo, D., Paszyński, M., Rachowicz, W., & Zdunek, A. (2007). Computing with hp-Adaptive Finite Elements, Vol. II. Frontiers. Three Dimensional Elliptic and Maxwell Problems with Applications. Chapman and Hall/Crc Applied Mathematics and Nonlinear Science.

[17] L. Demkowicz, A. Buffa, $H1$, $H(curl)$ and $H(div)$-conforming projection-based interpolation in three dimensions: Quasi-optimal p-interpolation estimates, Computer Methods in Applied Mechanics and Engineering 194(2-5) (2005) 267-296.

[18] L. Demkowicz, J. Gopalakrishnan, S. Nagaraj, P. Sepulveda, A spacetime DPG method for the Schrodinger equation, SIAM Journal on Numerical Analysis 55 (4) 1740–1759 (2017)

[19] W. Dorfler, A convergent adaptive algorithm for Poisson's equation, SIAM Journal on Numerical Analysis 33 (3) 1106–1124 (1996)

[20] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

[21] I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear equations. *SIAM Journal on Scientific and Statistical Computing*, 5:633–641, 1984

[22] J. Ernesti, C. Wieners, A space-time discontinuous Petrov–Galerkin method for acoustic waves, De Gruyter, Berlin, Boston, Chapter 3 89–116 (2019)

[23] G.W. Flake, R.E. Tarjan, K. Tsioutsiouliklis, Graph clustering and minimum cut trees, Internet Mathematics 1 (2003), 385-408.

[24] T. Fuhrer, M. Karkulik, Space-time least-squares finite elements for parabolic equations, Computers & Mathematics with Applications 9281 27–36 (2021)

[25] L. Gao and V. M. Calo. Preconditioners based on the alternating-direction-implicit algorithm for the 2d steady-state diffusion equation with orthotropic heterogeneous coefficients. Journal of Computational and Applied Mathematics, 273:274–295, 2015.

[26] P. Geng, J. T. Oden, and R. A. van de Geijn. A parallel multifrontal algorithm and its implementation. *Computer Methods in Applied Mechanics and Engineering*, 149:289–301, 1997.

[27] A. George. Nested Dissection of a Regular Finite Element Mesh. *SIAM Journal on Numerical Analysis*, 10(2), 345–363, 1973.

[28] J. Gopalakrishnan, P. Sepulvedas, A space-time DPG method for the wave equation in multiple dimensions, De Gruyter, Berlin, Boston, Chapter 4 117–140 (2019)

[29] J. Hartmanis and R. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[30] P. Heggernes, S.C. Eisenstat, G. Kumfert, A. Pothen, The Computational Complexity of the Minimum Degree Algorithm, ICASE Report No. 2001-42, (2001).

[31] R. Hiptmair. Multigrid method for maxwell's equations. SIAM Journal of Numerical Analysis, 36:204–225, 1998

[32] B. M. Irons. A frontal solution program for finite-element analysis. *International Journal for Numerical Methods in Engineering*, 2:5–32, 1970.

[33] G. Karypis and V. Kumar. METIS—unstructured graph partitioning and sparse matrix ordering system, version 2.0. *Tech. Rep.*, 1995.

[34] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[35] U. Langer and O. Steinbach (Eds.). Space-Time Methods: Applications to Partial Differential Equations (Vol. 25). Walter de Gruyter GmbH & Co KG. (2019)

[36] J.W.H. Liu, The multifrontal method for sparse matrix solution: theory and practice, SIAM Review 34 (1992), 82-109.

[37] A. El Maliki, M. Fortin, N. Tardieu, and A. Fortin. Iterative solvers for 3d linear and nonlinear elasticity problems: Displacement and mixed formulations. International Journal for Numerical Methods in Engineering, 83:1780–1802, 2010.

[38] C. A. de Moura, C. S. Kubrusly, The Courant-Friedrichs-Lewy (CFL) Condition: 80 Years After Its Discovery, Birkhauser, (2013).

[39] Neumuller, M.: Space-time methods: Fast solvers and applications, Ph.D. thesis, Graz University of Technology, Institute of Applied Mathematics (2013).

[40] A. Paszyńska, Volume and neighbors algorithm for finding elimination trees for three dimensional h-adaptive grids, Computers & Mathematics with Applications, 68(10) (2014) 1467-1478

[41] A. Paszyńska, K. Jopek, K. Banaś, M. Paszyński, P. Gurgul, A. Lenerth, D. Nguyen, K. Pingali, L. Dalcin, V. Calo, Telescopic hybrid fast solver for 3D elliptic problems with point singularities, Procedia Computer Science, 51 (2015) 2744-2748

[42] A. Paszyńska, M. Paszyński, K. Jopek, M. Woźniak, D. Goik, P. Gurgul, H. AbouEisha, M. Moshkov, V. M. Calo, A. Lenharth, D. Nguyen, K. Pingali, Quasi-optimal elimination trees for 2D grids with singularities, Scientific Programming, Article ID 303024 (2015), 1-18

[43] M. Paszyński, V. M. Calo, D. Pardo, A direct solver with reutilization of previously-computed LU factorizations for h-adaptive finite element grids with point singularities, Computers & Mathematics with Applications, 65(8) (2013) 1140-1151

[44] A. Schafelner, P. S. Vassilevski, Numerical results for adaptive (negative norm) constrained first order system least squares formulations. In: Recent Advances in Least-Squares and Discontinuous Petrov–Galerkin Finite Element Methods, Computers & Mathematics with Applications 95, 256–270, (2021)

[45] J. Schulze, Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods, BIT, 41, 4 (2001) 800.

[46] M. Skotniczny, M. Paszyński, A. Paszyńska, Bisection weighted by element size ordering algorithm for multi-frontal solver executed over 3D $h$-refined grids, Computer Methods in Materials Science, 16(1) (2016) 54-61

[47] O. Steinbach, H. Yang, Comparison of algebraic multigrid methods for an adaptive space–time finite-element discretization of the heat equation in 3d and 4d, Numerical Linear Algebra with Applications 25 (3) e2143 (2018)

[48] V. Thomée, Galerkin finite element methods for parabolic problems (Vol. 25). Springer Science & Business Media (2007).

[49] Voronin, K., Lee, C. S., Neumuller, M., Sepulveda, P., Vassilevski, P. S.: Space-time discretizations using constrained first-order system least squares (CFOSLS), Journal of Computational Physics 373 863–876 (2018)

[50] M. Yannakakis, Computing the minimum fill-in is NP-complete, SIAM Journal on Algebraic Discrete Methods, 2 (1981) 77-79

[51] O.C. Zienkiewicz, R. L. Taylor, J.Z. Zhu, The Finite Element Method: Its Basis and Fundamentals, Butterworth-Heinemann, 7th Edition (2013)

# Publications

My scientific results are summarized in the following papers:

[1] Marcin Skotniczny, Maciej Paszyński, Anna Paszyńska, *Bisection weighted by element size ordering algorithm for multi-frontal solver executed over 3D* h *refined grids*, **Computer Methods in Materials Science** 2016 vol. 16 no. 1, p. 54–61

[2] Marcin Skotniczny, *Computational complexity of hierarchically adapted meshes*, **Lecture Notes in Computer Science** 2020 vol. 12139. S. 226–239 *(International conference on Computational Science ICCS 2020)* [CORE A conference]

[3] Hassan AbouEisha, Victor Manuel Calo, Konrad Jopek, Mikhail Moshkov, Anna Paszyńska, Maciej Paszyński, Marcin Skotniczny, *Element partition trees for h-refined meshes to optimize direct solver performance. P. 1, Dynamic programming*, **International Journal of Applied Mathematics and Computer Science** 2017 vol. 27 no. 2, s. 351–365. [2 kwartyl, IF: 2.157]

[4] Piotr Gurgul, Marcin Sieniek, Krzysztof Magiera, Marcin Skotniczny, *Application of multi-agent paradigm to hp-adaptive projection-based interpolation operator*, **Journal of Computational Science**, 2013 vol. 4 iss. 3, p. 164-169 [1 kwartyl, IF: 4.97]

[5] Piotr Gurgul, Marcin Sieniek, Marcin Skotniczny, Krzysztof Magiera, Maciej Paszynski, *Agent-oriented image processing with the hp-adaptive projection-based interpolation operator*, **Procedia Computer Science** 2011 vol. 4 s.1844-1853 *(International conference on Computational Science ICCS 2020)* [CORE A conference]

[6] Marcin Skotniczny, Anna Paszyńska, Maciej Paszyński, *Algorithm for fast simulations of space-time finite element method* , **VII European Congress on Computational Methods in Applied Sciences and Engineering**, January 2016 (abstract)

[7] Marcin Skotniczny, Anna Paszyńska, Sergio Rojas, Maciej Paszyński, *Complexity of direct and iterative solvers on space-time formulations versus time–marching schemes for h-refined grids towards singularities*, https://arxiv.org/abs/2110.05804 (2022) (submitted to Q1 journal, under review)