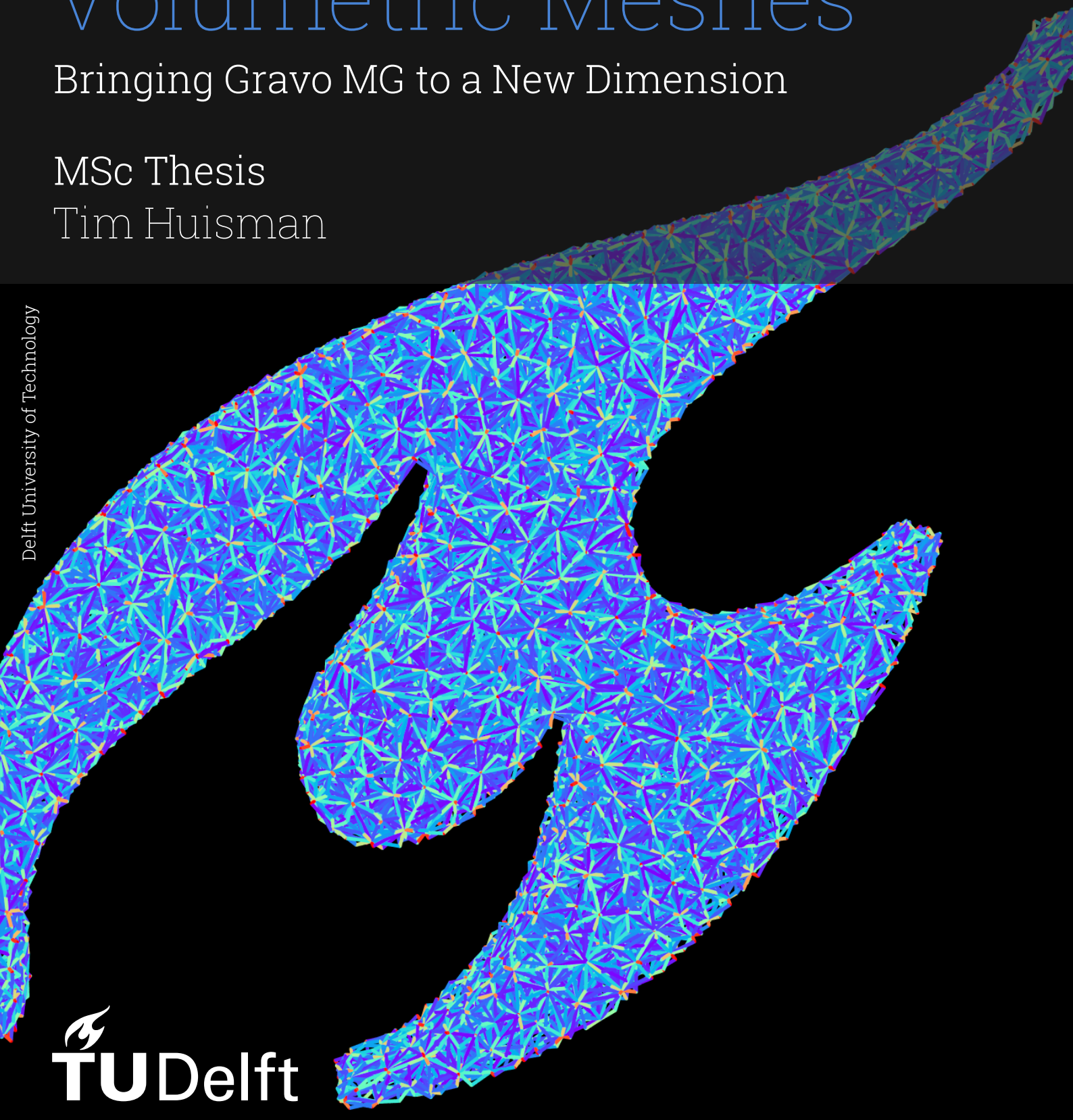
A Fast Geometric Multigrid Method for Volumetric Meshes

Bringing Gravo MG to a New Dimension

MSc Thesis

Tim Huisman



Delft University of Technology

TU Delft

A Fast Geometric Multigrid Method for Volumetric Meshes

Bringing Gravo MG to a New Dimension

by

Tim Huisman

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday, January 23, 2025, at 10:15 AM.

Student number:	5235499	
Project duration:	November 23, 2024 – January 23, 2026	
Thesis committee:	Prof. K. Hildebrandt,	TU Delft, supervisor
	Prof. M. de Weerdt,	TU Delft
	MSc. J. Campolattaro,	TU Delft

Cover:	TU Delft Flame by FlipFlop94 under CC BY 4.0 (Modified)
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

Linear systems are ubiquitous in the field of numerical simulation, and can be used to describe or approximate physical processes on arbitrary objects. These objects can be represented using volumetric meshes, often made out of tetrahedra. If the mesh contains many vertices, the corresponding linear system becomes too large to solve using conventional methods. Multigrid methods are a scalable alternative to iteratively approximate solutions for such large problems, but before they can be employed, a multigrid hierarchy that corresponds to the mesh must first be defined. A bad hierarchy can lead to bad convergence when solving the problem, but creating a more accurate hierarchy can be computationally expensive itself. We present an adaptation of Gravo MG, an algorithm to define such hierarchies for triangular meshes. This adaptation translates the same design principles for the hierarchy construction from the triangular to the tetrahedral case. Furthermore, we address some new challenges that arise when dealing with volumetric meshes, particularly the preservation of the boundary when coarsening the mesh.

Preface

I will not sugarcoat it; working on this thesis has been the most difficult thing I have ever had to do. From an intellectual perspective, entering an unfamiliar and yet well-established research field with the task of coming up with something new, the task could be considered challenging. From an emotional perspective, it often felt like an impossible trial. Whereas the course-based structure of the first few years of my education has felt like having to climb ergonomically designed ladders as fast as possible, this final challenge could be better compared to a rocky wall, with no clear path up or even a predefined target. Needless to say, trying to treat a wall as a ladder leads to a lot of falling, and many lessons were learned in the creation of this document, not all of them about multigrid methods. These were important lessons to learn, though, and if I had the chance to start over, I would have confidently made the same mistakes again. Doing a Master's thesis is a humbling experience, one that has maybe made me a little more intelligent, and if not that, then at least a lot wiser.

Even though doing a thesis can be a lonely endeavor, it would be ridiculous to imply I could have done it alone. Many people have supported me in wildly different ways, and while it will certainly not be enough to compensate for everything they have done for me, the least I can do is express some of my gratitude in the form of text.

First, I of course need to thank my supervisors, Professor Klaus Hildebrandt and Jackson Campolattaro, who pitched this project to me and have been my guides through the vast field of geometry processing. Whenever I was lost, you gave me valuable directions that led to a new fixation, and when I fixated too much, you dragged me out of it by pointing me to the things that mattered most. I know my process has not been exemplary; I would repeatedly shoot off on wild tangents, only to return to you far too late, more often than not with questionable results. Your patience with my tomfoolery is admirable, and the optimism and faith that you have shown in our meetings have helped me to stay on my feet just as much as your advice. I hope my work is valuable to you, directly or not, or at least that I have not been too much of a bother.

My next set of thanks goes out to my parents. It is easy to take for granted the care you receive at home. Growing up, you have guided me throughout everything I have done, with great care and the freedom to learn my own lessons. Aside from the obvious materialistic support that you have continuously provided throughout my upbringing, you have always given me a safe haven to return to when I wanted or desperately needed it. I am proud of the things I have achieved so far, but I know I would never have been able to if you had not had my back during the big moments and the many small moments in between. Thank you for always supporting me throughout all my adventures, for giving me the space to forge my own paths, and for picking me up every time I figuratively ran head-first into the wall, something that would happen less if only I would listen to you more often...

Next, I want to thank all of my fellow students I have had the pleasure to meet throughout this years-long journey. My thanks go out to Jegor, who was always there to accompany me as we slugged along on our separate journeys, and has served as a rock and a needed constant throughout this hard time. My thanks go out to all the friends I have made and shared experiences with, ones that I will cherish for decades to come. My thanks go out to my entire cohort, who challenged and helped me to reach my goals in the most formative years of my life, and to whom I hope to have reciprocated the favor. My thanks go out to every single student in the subsequent cohorts with whom I have had the chance to discuss the things I loved the most, to the point where I would learn about and love them more myself. I will miss the educational environment within the university and the community that comes with it, and I hope to find it back again someday, wherever I will end up next.

And finally, I must thank all the professionals at TU Delft who have guided me throughout my education and have given me the chance to grow in ways beyond the intellectual. Of particular importance is Stefan Hugtenburg: my teacher, my mentor, and the best boss I will ever have. Your impact on my life cannot be overstated. Eight years ago, you convinced me during an orientation day that Computer

Science was the future for me, but at that point, I could never have guessed just how much you would contribute to my professional and personal development later in my studies. Thank you for always being an example of a good educator and a good person, and thank you for granting me the rare opportunity to try and follow in that example myself. I owe more to you than you can imagine, and I am the person I am today because of you.

Tim Huisman
Delft, January 2026

Contents

1 Introduction

Systems of linear equations are ubiquitous in the field of numerical simulation. They can be used to find or help in finding solutions to partial differential equations in discrete domains, which are both handy tools to describe problems with all kinds of applications and infuriating puzzles that are still studied in great detail. Wherever a partial differential equation arises, a linear system is bound to follow, either to describe a solution or a close approximation of one. The introduction of computing devices a number of decades ago has made the task of simulating large systems feasible, but much research is still ongoing in improving scalability, in order to be able to push the limits of simulation further and further.

Many algorithms already exist for finding exact solutions to a given linear system, some even predating computing technology. Modern algorithms that directly solve such systems tend to make use of factorization and concurrency to ensure that finding solutions remains feasible at a large scale. Nevertheless, the time complexity of direct solvers hinders scalability severely, requiring other approaches to be applied instead. Iterative solvers provide an alternative approach, by iteratively improving on a given solution in linear time, allowing approximate solutions to be provided after a fixed amount of time or up to a fixed tolerated error. Solvers relying on iteration are useful, as long as their solution converges rapidly towards a “true solution”. Otherwise, the solving process will be a fruitless endeavor.

Multigrid methods are a revolutionary approach to iteratively solving linear systems on spatial domains. By considering the problem that requires a solution at varying levels of detail simultaneously, it is possible to reduce both the small errors and the large errors in the solution at the same time, leading to fast convergence. The importance of multigrid methods in numerical simulation cannot be understated. However, multigrid methods require a “good” multigrid hierarchy of domains to be defined, and the construction of such a hierarchy on irregular domains is far from a simple task; trade-offs always need to be made between accuracy and performance. Various branches of research have spawned to determine how best to define such a hierarchy under different conditions and circumstances.

Defining a proper multigrid hierarchy becomes more complicated as the focus is shifted from surface meshes towards *volumetric* meshes, i.e. three-dimensional meshes of which not just the surface is of relevance, but the interior of the mesh as well. The number of vertices in these meshes tends to be higher, and vertices also tend to have more neighboring vertices, thus affecting the density of the corresponding systems. Some research into volumetric multigrid hierarchies exists ([BKS11; SK11]), but these tend to prioritize the quality of the hierarchy over the efficiency of constructing the hierarchies. There is still progress to be made when it comes to fast-to-construct multigrid methods that can coarsen arbitrary irregular volumetric meshes.

Recently, [Wie+23] provided a new insight into constructing multigrid hierarchies for irregular triangular domains in 3D space. By relying on graph coarsening and Voronoi cells, their algorithm, *Gravo MG*, manages to quickly coarsen a geometric domain and define piecewise linear prolongation operators for it using a barycentric weighting scheme. The surprisingly good balance between preparation and execution raises questions about whether this approach can also be used under different circumstances, for example, in volumetric bodies consisting of tetrahedra.

Our work aims to extend the Gravo MG algorithm from [Wie+23] to support tetrahedral elements, and to explore and address the implications induced by this new ground. The extension of Gravo MG can be done quite naturally by considering that the fundamental building blocks of Gravo are not limited to a particular number of dimensions. Tetrahedra can be included by nesting some instructions and increasing the dimensionality of some parts of the algorithm.

The three-dimensional context brings rise to new challenges, which cause coarsened versions to degrade quite fast. Particularly around the boundary, anomalies can occur more easily in tetrahedral meshes. We propose several techniques to try and preserve the shape of the boundary as well as possible, using some heuristic info about what vertices belong to the boundary and which do not. These techniques include changes in the subsampling density, in the subsampling order, in how to keep the surface connected, and in how to smooth out the surface without degrading it.

We list all the required changes necessary for the extension and their implementation in detail. Furthermore, we provide a thorough quantitative evaluation of the algorithm, benchmarking its parameters against each other and providing a comparison between Gravo MG and some state-of-the-art algorithms that are similarly suitable for the task at hand, demonstrating that it constructs multigrid hierarchies of high quality in a time comparable to that of simple techniques.

2

Related Work

Multigrid Solvers

Multigrid solvers [Bra77; Wes04; BL11] are among the most efficient classes of iterative solvers. After their introduction in the previous century, they have become the default option to apply to large-scale linear systems. Multigrid methods operate by iteratively solving the linear system on multiple different resolutions of the domain simultaneously, with communication of systems and their solutions happening between adjacent resolutions. Two common types of multigrid solvers exist: *geometric* and *algebraic*.

Geometric multigrid (GMG) methods are distinguished by having their hierarchy of domains be determined solely by the geometric information of the initially provided domain, without relying on any information about the linear system that will be solved on it. For regular grids, the hierarchy usually consists of multiple versions of the same domain, where on each level higher up is a refinement of the one below, with new points located halfway between every pair of adjacent points [Bra77]. Multilinear interpolation can be used for transferring vectors between grids in these scenarios. The grids in question can be generated by globally refining every cell in the grid, leading to an increasingly finer grid, until the desired precision is obtained. To save resources, another common strategy is to refine the grid adaptively, only putting more detail in locations where detail is expected to matter, leaving the less important regions as coarse [BS99; SNI91].

Algebraic multigrid (AMG) methods [Bra86] find themselves at the other side of the coin, having no explicit information on the geometry of the domain and basing their hierarchy of linear systems only on a provided linear system. This linear system should represent the problem at the finest level of the hierarchy. Once this is provided, AMG generates the coarser matrices of the hierarchy independently. This is commonly done using either “classical” AMG approaches, such as Ruge-Stüben [RS87], or smoothed aggregation AMG [VMB96]. Since no geometric information is needed to construct these solvers, they can be applied in any context, as long as a matrix equation is provided. This makes AMG methods a popular choice when the need to solve sparse systems spontaneously arises. However, due to their ignorance of the geometric domain, the resulting hierarchy might prove to lead to slower convergence in comparison to hierarchies produced by most GMG methods.

Geometric Multigrid Solvers for Unstructured Grids

Geometric multigrid can also be applied to unstructured grids and meshes, in which the positions of the points do not follow a regular pattern. Two-dimensional grids are most often represented as triangular or quadrilateral meshes, which provide more freedom over the shape of the domain. The lack of structure does come with disadvantages, as geometric multigrid is not always as straightforward to apply to grids that do not follow a predictable structure. Refining coarse meshes is still easy, and many libraries allow the construction of hierarchies through global or adaptive refinement [Arn+21; Bas+21; And+21]. These operations will remain doable, as every refined cell in the coarse mesh can serve as its own small regular grid. Generating a hierarchy in the opposite direction, i.e. starting with a fine mesh and progressively coarsening it, is more complicated, as coarsening detailed geometry requires simplifying

the shape in a non-trivial manner. Plenty of algorithms exist that each coarsen and prolongate meshes in their own way. Most often, a proper subset of the vertex set, by deciding which points to remove from the original mesh, and then performing edge collapses or half-edge collapses to reconnect the removed node's neighbors [Hop96; Kob+98; RL03; NGH04; AKS03; BKS11]. These operations are generally executed such that some measure of the mesh quality is maximized or preserved, such as the aspect ratio of triangles [BKS11]. [Shi+06] does not rely on edge collapse, but rather treats the meshes as point clouds, where first a complete subset of vertices is decided, and only afterwards they are reconnected. This method makes the construction of such a hierarchy remarkably fast, but due to the lack of triangular information, the defined prolongation operators lead to worse convergence rates, thus marking a clear trade-off in the majority of use cases. [Liu+21] takes yet another approach, and uses a parametrization of the surface that is updated at every vertex removal to define the prolongation operators, which, while costly and time-consuming, has proven to be an effective strategy allowing for fast convergence. A little more recently, [Wie+23] designed an algorithm similar to [Shi+06], which subsamples point clouds, reconnects them, and then retriangulates the mesh, allowing for prolongation operators to be defined using barycentric interpolation weights. The construction of such a hierarchy is fast, while outperforming [Shi+06] by a wide margin in most scenarios.

Geometric Multigrid Solvers for Volumetric Meshes

Aside from 2D meshes, such as meshes located in the 2D space or describing surfaces in 3D space, there are many use cases for *volumetric* meshes as well, particularly in the realm of physics simulations. They can be used as the domain of interest in problems such as, for example, heat transfer [CC02], fluid simulation [MST10], and elastic deformation [JW00]. Volumetric meshes can be represented using regular grids, tetrahedra, and/or unstructured hexahedra, among other methods. For each of these representations, multigrid methods have been studied extensively [PPM92; Hem95; DGW11]. While some of the methods and approaches that apply to 2D meshes can still be applied to these domains, volumetric meshes introduce new challenges, both on a conceptual level and a performance level [Jac15]. The main reason for the drop in performance is the higher average number of connections per point. For regular triangular meshes, this average approximates three edges per vertex, whereas for tetrahedral meshes this statistic is closer to *seven* edges per vertex, causing most linear systems defined on tetrahedra to be far more dense than their two-dimensional equivalent. An example of an algorithm that extends from triangular to tetrahedral meshes is [BKS11], which ports its definitions from 2D into 3D, relying on notions of tetrahedral Delaunay conditions, discrete curvature, and "ridges" on the boundary. [SK11] instead relies on the observation that tetrahedral meshes have a defining curved surface on which regular coarsening techniques can be applied first. Afterwards, the coarse surfaces can be filled with tetrahedra to recreate a volumetric domain.

Both of the discussed procedures give certain guarantees about the quality of the meshes in the hierarchy, by carefully deciding what part of the mesh to coarsen and how to do so. While the output quality speaks for itself, these approaches take quite a bit of time to execute, since hierarchy construction has been turned into such an arduous task. In contrast, the work put forward in this thesis prioritizes fast construction times over mesh quality, in the hope that good convergence can still be achieved without having to spend minutes on preparing the multigrid.

3

Background

This chapter discusses the context and the preliminaries to the thesis. Its goal is to lay a foundation for the motivation of the research, while also educating the reader on the topics that are essential in interpreting it correctly. The following text has been written such that a person with a Computer Science background, but no prior experience with computer graphics or the mathematics that accompanies it, is able to understand the thesis in full. The chapter covers, among other things, partial differential equations, iterative techniques for approximating solutions to them, the basics of multigrid methods, and the inner workings of the *Gravo MG* algorithm.

3.1. Partial Differential Equations

A *real-valued function of real variables*, commonly referred to as a **real function**, is a function that maps a sequence of real values to another sequence of real values. Formally, it can be described as a function of the type $\mathbb{R}^m \to \mathbb{R}^n$. An example would be the function

$$f(x_1, x_2, x_3) = x_1^2 + x_2^2 + x_3^2,$$

of which the function type is denoted as $f : \mathbb{R}^3 \to \mathbb{R}^1$, or rather $f : \mathbb{R}^3 \to \mathbb{R}$. From this point onward, the reader may assume that any introduced function is of the type $\mathbb{R}^m \to \mathbb{R}^n$, unless otherwise stated. To simplify notation, a sequence of values $(x_1, \ldots, x_n)$ will in the future often be referred to as a vector $\mathbf{x}$, recognizable by its boldface typesetting.

The **derivative** of a real function describes how much the output of the function changes if we slightly tweak its input. In secondary education, it is usually taught as being the ”slope” of a function at a certain point, which could serve as useful intuition in the spatial setting. In terms of changes happening over time, one could instead think of it as a velocity or a rate of change.
For functions that have more than one input, each of the variables can have a different effect on the output. We can then define multiple derivatives for the function, where each of them is the derivative over just one of the given variables, under the assumption that the other variables are fixed and act merely as constants. These derivatives are referred to as **partial derivatives**, and are denoted as $\frac{\partial f}{\partial x}$, where f is the function and x is the variable with respect to which we take the derivative. As an example, the function

$$f(x_1, x_2) = 3x_1 + x_1^2 x_2$$

has the partial derivatives

$$\frac{\partial f}{\partial x_1} = 3 + 2x_1 x_2,$$

$$\frac{\partial f}{\partial x_2} = x_1^2.$$

A **partial differential equation**, commonly abbreviated as **PDE**, is an equation that describes equalities between expressions involving a function f and its partial derivatives. A *solution* to a PDE or a collection

of PDEs is an expression for the function f in question, such that the PDEs hold over the entire domain on which they are defined. Finding a solution is often quite hard, which motivates the entirety of this thesis.

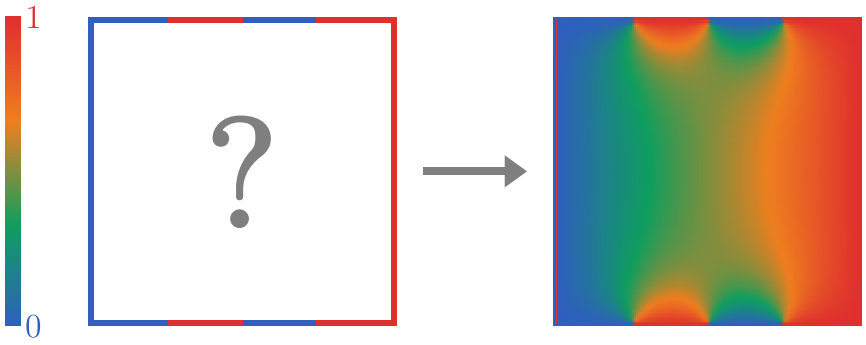


Figure 3.1: A Dirichlet problem being solved on a continuous square domain.

A well-known example of a relatively simple PDE is the *Dirichlet problem*, which will be introduced here and will routinely be referred back to to demonstrate new concepts in the future. The **Dirichlet problem** (see Figure 3.1) is a problem in which one must find a "harmonic function" on a domain with defined boundary values. That is, for a domain Ω with boundary $\partial\Omega$, and a known boundary function $g : \partial\Omega \to \mathbb{R}$, the goal is to find a function $f : \Omega \to \mathbb{R}$ such that

$$f = g, \text{ for all } \mathbf{x} \in \partial\Omega, \text{ and}$$

$$\Delta f = 0, \text{ for all } \mathbf{x} \in \Omega,$$

where Δ represents the Laplace operator

$$\Delta f = \sum_i \frac{\partial^2 f}{\partial x_i^2}.$$

An intuitive analogy for this problem could be how heat propagates through a room if the walls emit certain constant temperatures from various locations. The Dirichlet problem will recur on multiple occasions throughout this chapter, due to its simplicity in comparison to the vast number of PDEs that are also commonly used in practice.

3.2. Discretization

PDEs are notoriously difficult to solve. Even "easy" PDEs tend to have complicated analytical solutions, which are challenging to derive. For more complex PDEs, which might very well just be simple PDEs with just a slight twist applied to them, solutions are often infeasible to determine. For this reason, the search for analytical and exact solutions to PDEs is often not attempted in practical research, and instead *approximations* to such solutions are made through numerical means. These numerical approximations often involve the efforts of a computer, which can do billions of computations in a matter of seconds.
A common pitfall with computers, however, is that they do not deal well with concepts such as "continuity" or "infinitesimals", which are fundamental to calculus and the theory behind derivatives. To translate the mathematics to a format the computer can understand, sacrifices must be made, and the continuity must be replaced with discrete and finite parts, a process described as **discretization**.

The biggest and most fundamental change of discretization can be observed in the *representation* of a solution to a system of PDEs. Unlike in mathematics, where functions are usually symbolically defined using expressions and symbols, the computer will have to resort to another way of representing functions to do its approximations. Since computers can deal well with large amounts of information, they can give a reasonably accurate definition of a function by defining what its values are on a set of predetermined points in the domain, and implicitly defining the rest of the function using interpolation or extrapolation. One could imagine these points as "tent poles", attempting to give shape to a continuous

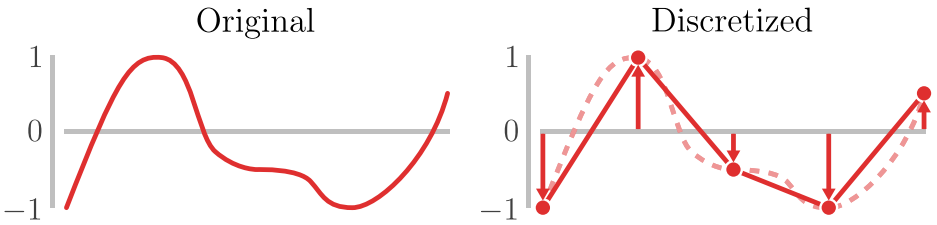


Figure 3.2: A continuous function on a one-dimensional domain, and a discretized representation of it made up of five points. The function values in between the given points can be determined through any kind of interpolation, e.g. linear interpolation as shown here.

function f by setting up a few supporting points, and allowing everyone else to take guesses at the values in between these points through interpolation. This way, it is possible to reasonably accurately represent continuous functions using discrete data (see Figure 3.2).

A few types of discretization are used quite commonly. For example, in physics simulations, three-dimensional spaces are often discretized using **regular grids**, usually *Cartesian* grids, which subdivide a cuboid representing the domain into a collection of smaller cuboids of equal size (see Figure 3.3). At each of the corners of the cuboids, a vertex can be placed, indicating a point at which our function f is evaluated and defined.

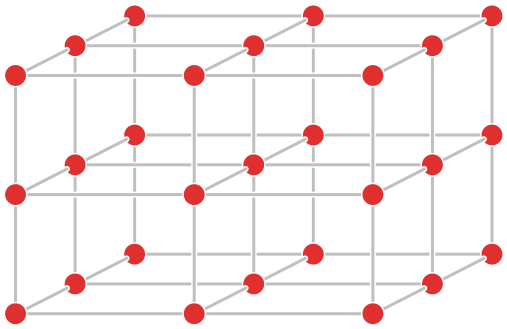


Figure 3.3: A small three-dimensional Cartesian grid.

While not as intuitive to visualize, the *temporal* dimension can also be discretized similarly. It can either be treated the same as any spatial dimension in the problem, or it can be simulated by taking a solution for a fixed timestamp, and using it to generate a new solution at a time shortly in the future, with a small but non-zero duration separating them. Both approaches would yield a discretization of time, subdivided into intervals of equal length.

Whenever a real-life problem or process is discretized, much information is lost, and hence whatever solution is obtained from the problem that remains, will at best be a close approximation of the original problem that needed solving. Accuracy can be improved by using a finer discretization, i.e. one that contains more points that are closer to each other, leaving the original domain slightly more intact. However, this comes at a great computing cost and can quickly result in instances that reach far outside the realm of what is computationally feasible. A tradeoff must always be made between performance and accuracy when it comes to numerical simulations such as these.

On the positive side, an advantage of discretization is that many intangible concepts and incomputable operators originally used to define PDEs take on a new form, usually one that is much, *much* easier to reason about. Partial derivatives and integrals are now replaced with simple operations such as addition and subtraction, or multiplication and division. For instance, for a function f that is discretized and defined at a series of uniformly distributed points $x_1, \ldots, x_n$ with a consistent spacing of h, i.e. $x_{i+1} - x_i = h$, the derivative $\frac{df}{dx}$ evaluated close to point x_i can be approximated using a simple division of subtractions,

$$\frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}} = \frac{f(x_i) - f(x_{i-1})}{h},$$

and its second derivative follows naturally as the “derivative” of “derivatives”, along the lines of

$$\frac{\left(\frac{f(x_{i+1}) - f(x_i)}{h}\right) - \left(\frac{f(x_i) - f(x_{i-1})}{h}\right)}{h} = \frac{1}{h^2} f(x_{i+1}) - \frac{2}{h^2} f(x_i) + \frac{1}{h^2} f(x_{i-1}).$$

The Dirichlet problem seen before, if applied to a regular one-dimensional grid, would contain many

equations that resemble

$$f(x_{i+1}) - 2f(x_i) + f(x_{i-1}) = 0,$$

which computationally looks a lot simpler than the continuous version seen before, which incorporated the Laplace operator Δ.

Considering the fact that the evaluations of $f(x_i)$ are the unknowns that should be solved for, it appears that PDEs, after being discretized, tend to present themselves as equations that are *linear* in nature, in the sense that they consist only of terms of either constants or constant multiples of variables. The act of solving PDEs approximately can therefore be reduced to the act of solving linear equations, which is an important insight, as it bridges the gap between the non-computable and the easily-computable. At this point, we will completely abandon the context of partial differential equations, and continue our endeavors in the context of *systems of linear equations*.

3.3. Linear Systems

Linear equations are equations that can be represented in the form

$$a_1 x_1 + \cdots + a_m x_m = b,$$

where b and all a_is are *constants* in $\mathbb{R}$, and all x_is are *variables* in $\mathbb{R}$. A linear equation can also be written using a dot product, as $\mathbf{a} \cdot \mathbf{x} = b$ or $\mathbf{a}^\intercal \mathbf{x} = b$. **Systems of linear equations**, also referred to as **linear systems**, are collections of linear equations $\mathbf{a}_i^\intercal \mathbf{x} = b_i$ that all reason simultaneously about the same variables $x_1, \ldots, x_m$. A linear system with n equations can be represented as one equation, by turning it into what is known as a **matrix equation**. Namely, we can define

$$A = \begin{bmatrix} a_{1,1} & \cdots & a_{1,m} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,m} \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix},$$

and consequently write our system as

$$A\mathbf{x} = \mathbf{b}.$$

This is a standard form often seen in literature. A solution to a matrix equation, and thus to a linear system, is given by any vector $\mathbf{x}$ for which the above equation would hold.

Linear systems are ubiquitous in numerical simulations, as complex systems can surprisingly often be approximated using linear relations between unknowns. This is a phenomenon we have already seen in Section 3.2 with regards to partial differential equations; even though we had only really observed one linear equation for a single point before, one can imagine that as we try to satisfy similar equations for a *collection* of points in a spatial region at the same time, a linear system is bound to present itself. Since each PDE tends to only reason about a small neighborhood around a point within the domain, many of the discrete equations only involve a few of the many variables included in the problem. The consequence is that A is a **sparse matrix**, a matrix in which the majority of the values are 0s. This will be significant later on, as sparse matrices can be encoded and worked with more efficiently, which will have a great impact on our solving algorithms later on.

Linear systems are not necessarily guaranteed to have just one solution. Depending on the provided matrix A, there might exist just one $\mathbf{x}$ satisfying the equation, or there might exist no $\mathbf{x}$s or even an infinite number of $\mathbf{x}$s. The discussion of these phenomena becomes significantly simpler if we consider A to be a **square matrix**, i.e. a matrix of which the number of rows and the number of columns are equal. From here on out, unless stated otherwise, the reader may presume that A refers to a *square* matrix whenever it is mentioned.
Square matrices A have exactly one solution if they are **invertible**, i.e. there exists a square matrix A^{-1} such that the multiplication AA^{-1} gives the identity matrix. This property has many different equivalent meanings: A has an inverse, A has a non-zero determinant, As columns are linearly independent; the list goes on and on. Matrix equations involving a *non*-invertible A are bound to have either zero solutions or an infinite number of solutions, depending on the chosen vector $\mathbf{b}$ to accompany A in the equation, whereas equations with an invertible A always have exactly *one* solution.

Due to their prominence, much work has gone into finding ways to efficiently solve linear systems. Some common solving techniques are discussed in the following sections, first casting a light on some background on *direct solvers* (Section 3.3.1), and from there continuing with *iterative solvers* (Section 3.3.2), the latter of which has particular relevance with regards to the topics that have yet to appear.

3.3.1. Direct Solvers

Direct solvers are solvers for matrix equations that are guaranteed to find an exact solution in a finite number of computations. For small systems, they can provide such a solution quickly enough for them to be the preferred option. While understanding specific techniques is not strictly necessary for understanding the thesis, two approaches are briefly discussed to create a contrast with the iterative solvers later on.

One of the oldest techniques for finding solutions to linear systems is **Gaussian elimination**, named after historical mathematician Carl Friedrich Gauss. The core idea is to reduce A to *row echelon form* by repeatedly applying the *elementary row operations* to it. An invertible square matrix is in row echelon form if it's an upper triangular matrix of which the diagonal consists of only 1s. Visually, this means A should look like

$$\begin{bmatrix} 1 & * & \cdots & * \\ 0 & 1 & \cdots & * \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix},$$

where $*$ represents any value, possibly 0. The elementary row operations are:

- swapping two rows;
- scaling a row with a non-zero factor;
- subtracting a multiple of a row from another.

Each of these operations preserves the solution space of the linear system; any solution remains a solution, and anything that is not will not suddenly become a solution. When these operations are applied to rows of A, they should simultaneously be applied to the respective entries of $\mathbf{b}$, to maintain consistency between the left-hand side and the right-hand side of the equation. Once the row echelon form has been achieved, all entries of $\mathbf{x}$ can be determined in reverse order by solving the equations from the bottom to the top.

While Gaussian elimination is simple and interpretable, one of its disadvantages is that it has a runtime complexity of $O(n^3)$, with n defined as the size of the vector $\mathbf{x}$. For practical use cases, which nowadays involve a large number of variables, Gaussian elimination is not a viable option. On top of that, the execution of this algorithm requires a lot of multiplication and division of numbers. If these numbers are encoded in a floating-point representation, their precision will degrade rapidly, and direct solving algorithms do not account for such numerical instability, leading to possibly huge errors in the final result.

Direct solving techniques exist that perform better on both fronts. For example, an improvement can be made by constructing a **Cholesky decomposition** or **LDL decomposition** on the matrix A, and using those to find a solution. The details of this approach can be found in [GV96]. This kind of decomposition can be a beneficial alternative if A is **symmetric**, meaning $A = A^\intercal$, and **positive definite**, meaning that for every non-zero vector $\mathbf{x}$, it holds that $\mathbf{x}^\intercal A\mathbf{x} > 0$. For sparse systems, Cholesky decomposition has been found to have superior performance over most other direct solving techniques [GSH07], but it will still fall short once the size of the problem becomes too large. In these cases, alternative solving techniques must be considered, which will be discussed next.

3.3.2. Iterative Solvers

As discussed previously, direct solvers are not always practical. Their lack of scalability and numerical stability makes them less viable to apply to large systems, even though they, on a theoretical level, could provide the exact solution to the system. For most purposes, however, an exact solution is not always necessary; in most use cases, it would be sufficient to have an algorithm that can estimate a solution that is correct up to a certain error. This leads us to the idea of *iterative solvers*.

Iterative solvers are solvers that start with an initial guess at a solution $\mathbf{x}$, often referred to as $\mathbf{x}^{(0)}$, and then try to deduce a *more accurate* "solution" $\mathbf{x}^{(1)}$ from it, by adjusting it to be a slightly closer approximation of the actual solution to the system $A\mathbf{x} = \mathbf{b}$. Afterwards, the same process can be followed with $\mathbf{x}^{(1)}$, leading to a new candidate solution $\mathbf{x}^{(2)}$, which should be even better than the previous two. This step can be iterated over and over again, each time providing us with a solution that should be at least a little more accurate than the previous one. At any point, when the solution is deemed "good enough" or time has run out, the solver can decide to terminate its optimization and return the last computed candidate solution $\mathbf{x}^{(n)}$.

To evaluate the quality of a candidate solution, we need to have a notion of an *error*, a metric to evaluate how close a candidate is to the actual solution. A first instinct might be to just define the error as the Euclidean distance from the candidate to the solution $\mathbf{x}^*$, i.e. as $\mathrm{Err}^{(i)} = \left\|\mathbf{x}^* - \mathbf{x}^{(i)}\right\|$. The issue with this definition is that it assumes that $\mathbf{x}^*$ is known, which it obviously is not; finding $\mathbf{x}^*$ is precisely what we are trying to do! So instead, the error is commonly defined as the **residual error** of the solution, which is the norm of the **residual vector** $\mathbf{r}^{(i)} = A\mathbf{x}^* - A\mathbf{x}^{(i)} = \mathbf{b} - A\mathbf{x}^{(i)}$, leading to the formula

$$\mathrm{Err}^{(i)} = \left\|\mathbf{b} - A\mathbf{x}^{(i)}\right\|.$$

To evaluate this error, knowledge about $\mathbf{x}^*$ is not required; only knowledge about A and $\mathbf{b}$ is, which is already a known part of the problem.

To be able to deduce a new candidate solution from an old one, an algorithm is required that can optimize a given solution. Two options for such an algorithm are the Jacobi method and the Gauss-Seidel method, which will be examined in Sections 3.3.2.1 and 3.3.2.2, respectively.

3.3.2.1. Jacobi Method

The **Jacobi method** is a technique to improve a candidate solution to a linear system, i.e. to reduce its residual error. The idea supporting the Jacobi method is quite simple. In the matrix equations that we have discussed so far, the i^{th} equation describes what value the i^{th} point should have. As an example, in the Dirichlet problem, the i^{th} equation dictates that the i^{th} point must be the average of its neighbors. A straightforward idea would then be to compute this average for each of the points and change their values accordingly.

To phrase this idea in a mathematical and more general way, suppose that our system $A\mathbf{x} = \mathbf{b}$ is given by

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix}, \text{ and } \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix},$$

and that a candidate solution $\mathbf{x}^{(i)}$ is given. The Jacobi method will then compute a new vector $\mathbf{x}^{(i+1)}$ such that for each $j \in \{1, \ldots, n\}$, the value of $x_j^{(i+1)}$ is set such that

$$(a_{j,1}x_1^{(i)} + \cdots + a_{j,j-1}x_{j-1}^{(i)}) + a_{j,j}x_j^{(i+1)} + (a_{j,j+1}x_{j+1}^{(i)} + \cdots + a_{j,n}x_n^{(i)}) = b_j.$$

Consequently, the following update rule arises:

$$x_j^{(i+1)} = \frac{b_j - \sum_{k \neq j} a_{j,k}x_k^{(i)}}{a_{j,j}}$$

A visual example of the Jacobi method in action can be observed in Figure 3.4. The demonstrated problem is a Dirichlet problem on a uniform one-dimensional domain, where the two values at the extreme ends of the domain are fixed to particular values. The goal is thus to satisfy the equation $-\frac{1}{2}x_{j-1} + x_j - \frac{1}{2}x_{j+1} = 0$ for each internal point, leading to the update rule

$$x_j^{(i+1)} = \frac{1}{2}x_{j-1}^{(i)} + \frac{1}{2}x_{j+1}^{(i)}.$$

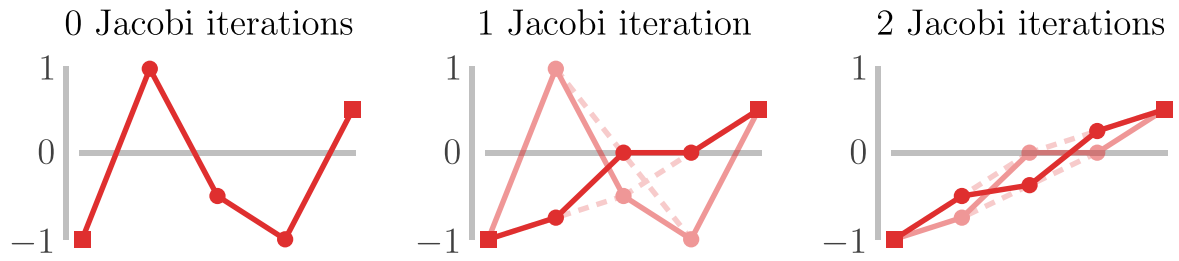


Figure 3.4: Multiple numbers of iterations of the Jacobi method applied to a one-dimensional Dirichlet problem. After each iteration, every point's value is adjusted to be the average of its neighbors' previous values.

The effect of the Jacobi method is that high outliers in the function are iteratively "smoothed out", at least for the Dirichlet problem.

One can also write the update rule as a single matrix expression applied to all values at once, by considering a decomposition of A into three matrices

$$L = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{2,1} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & 0 \end{bmatrix}, \quad D = \begin{bmatrix} a_{1,1} & 0 & \cdots & 0 \\ 0 & a_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n,n} \end{bmatrix}, \quad U = \begin{bmatrix} 0 & a_{1,2} & \cdots & a_{1,n} \\ 0 & 0 & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix},$$

such that

$$A = L + D + U.$$

Given a candidate vector $\mathbf{x}^{(i)}$, the Jacobi method is able to obtain a better candidate $\mathbf{x}^{(i+1)}$ by evaluating

$$\mathbf{x}^{(i+1)} = D^{-1}\left(\mathbf{b} - (L+U)\mathbf{x}^{(i)}\right).$$

Algorithm 1 One iteration of the Jacobi method.

```
function jacobi(A, b, x)
    x' ← 0
    for i ∈ {1, ..., n} do
        rowSum ← 0
        for i ∈ {1, ..., n} do                ▷ Sum up non-diagonal entries
            if i ≠ j then
                rowSum ← rowSum + A_{i,j} x_j
            end if
        end for
        x'_i ← (b_i − rowSum)/A_{i,i}
    end for
    return x'
end function
```

An element-wise algorithm that applies this technique once to a solution $\mathbf{x}^{(i)}$ is shown in Algorithm 1.

Under the assumption that A is a sparse matrix encoded with a sparse data structure, one iteration can be run in $O(\text{nonzero}(A))$ time, where $\text{nonzero}(A)$ denotes the number of non-zero entries in A.

Convergence to a solution is not guaranteed for all linear systems. The Jacobi method is known to converge if A is **strictly diagonally dominant**, meaning that for each row in A, the magnitude of the value on the diagonal is *strictly* greater than the sum of the magnitudes of all other values in the row. This does not hold for all systems of interest, which motivates the usage of a *variation* of this method, which is discussed next.

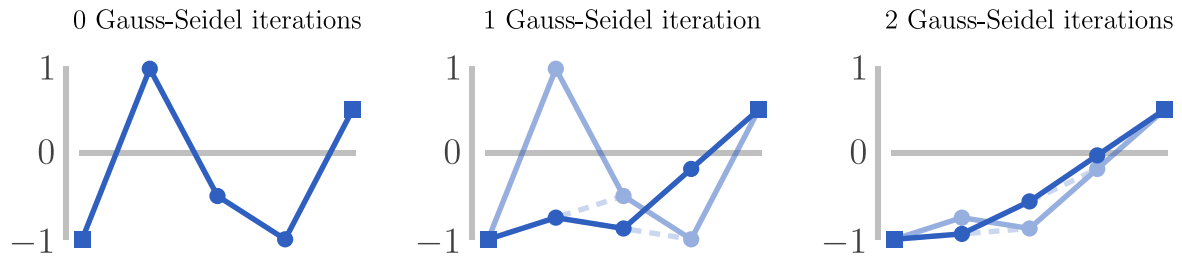


Figure 3.5: Multiple numbers of iterations of the Gauss-Seidel method applied to the same one-dimensional Dirichlet problem as in Figure 3.4. After each iteration, every point's value is adjusted to be the average of its left neighbor's new value and its right neighbor's old value.

3.3.2.2. Gauss-Seidel Method

The **Gauss-Seidel method** is yet another iterative algorithm for optimizing an estimate $\mathbf{x}^{(i)}$. It works very similarly to the Jacobi method, but slightly adapts the technique by immediately reusing newly computed entries of $\mathbf{x}^{(i+1)}$ in the computation for later entries within the same vector. This means that the Gauss-Seidel method computes each $x_j^{(i+1)}$ in some order, usually lexical order, each time trying to satisfy the equation

$$(a_{j,1}x_1^{(i+1)} + \cdots + a_{j,j-1}x_{j-1}^{(i+1)}) + a_{j,j}x_j^{(i+1)} + (a_{j,j+1}x_{j+1}^{(i)} + \cdots + a_{j,n}x_n^{(i)}) = b_i,$$

which leads us to the update rule

$$x_j^{(i+1)} = \frac{b_j - \sum_{k<j} a_{j,k}x_k^{(i+1)} - \sum_{k>j} a_{j,k}x_k^{(i)}}{a_{j,j}}.$$

With the Gauss-Seidel method, the way in which we order the variables in the linear system has a small impact on how fast information propagates and how quickly solutions converge. However, since any order is guaranteed to converge anyway, an arbitrary order may be used.

The Gauss-Seidel method is demonstrated visually in Figure 3.5. Similar behavior as with the Jacobi method can be observed, though due to the immediate usage of the new values, the function tends to "wiggle around" less.

Just like the Jacobi method, the update rule of the Gauss-Seidel method can be rewritten as a matrix equation, relying on the same decomposition of A into L, D, and U. The difference is that the performed iteration is instead given by

$$\mathbf{x}^{(i+1)} = (L + D)^{-1}\left(\mathbf{b} - U\mathbf{x}^{(i)}\right).$$

Algorithm 2 One iteration of the Gauss-Seidel method.

```
function gaussSeidel(A, b, x)
    for i ∈ {1, ..., n} do
        rowSum ← 0
        for i ∈ {1, ..., n} do                ▷ Sum up non-diagonal entries
            if i ≠ j then
                rowSum ← rowSum + A_{i,j} x_j
            end if
        end for
        x_i ← (b_i − rowSum)/A_{i,i}
    end for
    return x
end function
```

An element-wise algorithm can also be constructed for the Gauss-Seidel method, as shown in Algorithm 2. Unlike in the algorithm for the Jacobi method, the values from $\mathbf{x}^{(i)}$ and $\mathbf{x}^{(i+1)}$ are no longer separated

into different data structures, and instead the given vector is modified in-place. The consequence is that newly-computed entries of $\mathbf{x}^{(i+1)}$ are immediately used in the computation of later entries of $\mathbf{x}^{(i+1)}$.

A concrete benefit of the Gauss-Seidel method in comparison to the Jacobi method is that its convergence is not only guaranteed for strictly diagonally dominant systems, but also for matrices that are *positive semi-definite*. A matrix A is **positive semi-definite** if it is symmetric and if for every vector $\mathbf{x}$, it holds that $\mathbf{x}^\intercal A\mathbf{x} \geq 0$. This is a bit different from the definition of *positive definite* given earlier, for which the *strict* inequality $\mathbf{x}^\intercal A\mathbf{x} > 0$ was defined. The new convergence conditions broaden the domain of solvable problems significantly and allow us to apply iterative techniques to a couple of very practical systems that will be introduced in a later chapter.

3.4. Multigrid Methods

This section will introduce and discuss the general idea of *multigrid methods*, starting with a motivating example for their relevance in numerical optimization in Section 3.4.1, and then outlining the framework and highlighting all elements involved in Section 3.4.2.

3.4.1. Motivation

Iterative solvers, when applied to the right problems, will often eventually converge to an approximately right answer, if given enough time. Unfortunately, iterative solvers might take an *unreasonable* amount of time until they resemble a usable solution. One of the causes of this can be demystified by considering how the Jacobi method would operate on a Dirichlet problem with a large domain.

Imagine a finite one-dimensional domain Ω as displayed in Figure 3.6, regularly discretized using 33 points, with the two extreme points forming the boundary $\partial\Omega = \{p_0, p_{32}\}$. We fix $f(p_0) = -1$ and $f(p_{32}) = 1$, and initialize the 31 points in between with $f^{(0)}(p_i) = 0$. In its attempt to solve the Dirichlet problem, the Jacobi method will iterate over each variable point p_i and compute

$$f^{(j+1)}(p_i) = \frac{f^{(j)}(p_{i-1}) + f^{(j)}(p_{i+1})}{2}.$$

After one iteration, $f^{(1)}(p_1) = -0.5$ and $f^{(1)}(p_{31}) = 0.5$ are obtained, but for all other points in between, $f^{(1)} = 0$ still, as the repercussions of the boundary values at p_0 and p_{32} have not been able to reach them yet after just one iteration. In the next iteration, $f^{(2)}(p_2)$ and $f^{(2)}(p_{30})$ become non-zero as well, but for *all* the relevant points to become non-zero, at least 15 iterations would be needed, which is an arguably high number for a still relatively small domain. The number of iterations required for f to actually *converge* with an error lower than some desired small value would be significantly higher. Suppose now that we did not have around thirty variable points, but around thirty *thousand* instead! The process would be agonizingly slow, and it would be unreasonable to sit and wait for the computer to finish its job, while it slowly, seemingly perpetually, tweaks its solution by the tiniest amounts each time.

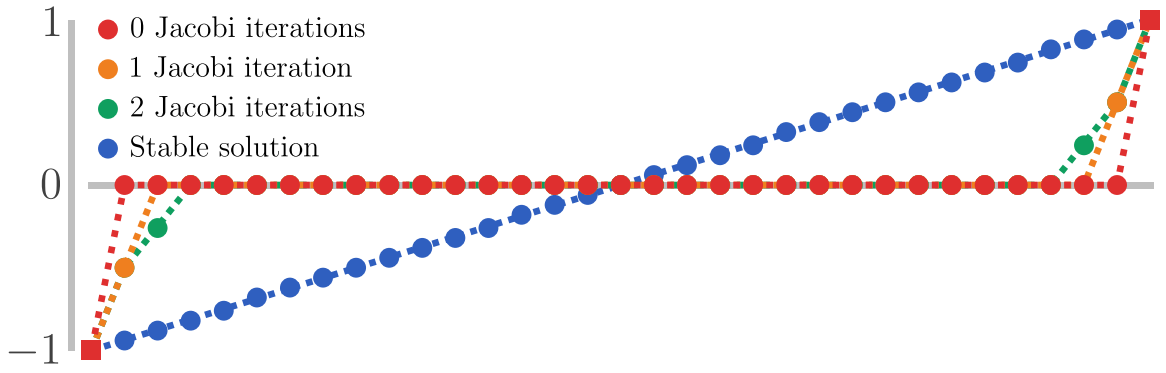


Figure 3.6: A demonstration of the convergence of the Jacobi method on a large one-dimensional Dirichlet problem. Applying Jacobi iterations will let us slowly approach the stable solution, but it might take a long time for this to happen.

To phrase the situation differently, the issue with the Jacobi method, as shown just now, is that *it is unable to see the big picture of a problem*. It only applies local corrections based on the state of a small neighborhood around each point, smoothing out the fine irregularities of the solution, but never drastically reshaping it to better resemble the broad curves seen in the expected end product. The effects of distant updates take many iterations to propagate through the entire domain, making convergence difficult. The Gauss-Seidel method propagates these changes slightly faster than the Jacobi method due to its in-place updates, but it still hopelessly falls short in terms of performance for the very same reason. What is needed is a new approach with which huge gaps can be bridged, and relations between points with great distances between them can be both represented and taken advantage of. This leads us to the concept of *multigrid methods*.

3.4.2. Overview

A **multigrid method** (occasionally abbreviated as an *MG method*) is a hierarchical approach to iteratively solving linear systems. In a nutshell, it achieves fast convergence by optimizing a solution on multiple discretizations of a domain simultaneously, each discretization having been made at a different resolution. By observing the problem at multiple different scales, multigrid methods can optimize both on a detailed level and on a grand level at the same time.

3.4.2.1. Hierarchy

Before the actual solving algorithm can be explained, it is important to clarify what "multiple different scales" means in practice. For this, consider again a one-dimensional domain Ω_0 discretized with 33 points (see Figure 3.7), similar to the one introduced in Section 3.4.1. To generate a new discretization Ω_1 with a lower level of detail, we can choose to remove every second point in Ω_0, yielding a new regular discretization with twice the spacing in between its points. The general process of representing a domain with a new, similar domain consisting of fewer points is called **coarsening**, and the two domains are referred to as the **fine domain** and the **coarse domain**, respectively, with respect to each other. If the Jacobi method were to be applied to Ω_1 instead of Ω_0, information would be transferred through space more quickly, as the point's neighbors are located at a larger distance, meaning that the information travels further in one iteration. Of course, since the coarse domain contains fewer points, the obtained solution might not have the desired level of detail; we still wish to find a solution on the domain Ω_0. In Section 3.4.2.3, this problem will be addressed.

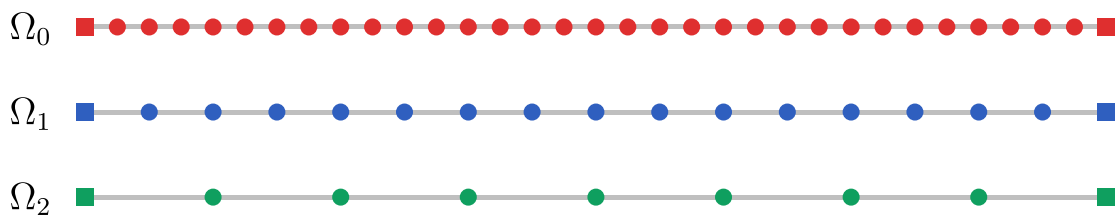


Figure 3.7: Multiple discretizations of a regular one-dimensional domain, at different resolutions.

We can similarly coarsen Ω_1 to get a domain Ω_2, and repeat this process any number of times, providing us with a **hierarchy** of domains $\Omega_0, \ldots, \Omega_k$ where every level contains significantly fewer points than the previous one. Generating such a hierarchy is the first step required to pave the way for multigrid methods, but just coarsening the domain is not enough, as we would still need to translate the posed problem into this new context as well. This begs the question: how do we coarsen a linear system, i.e. its *matrices* and *vectors*?

3.4.2.2. Restriction and Prolongation

Restriction and **prolongation** are essential components for MG methods to work, and must be defined for each newly created domain in the hierarchy. Their purpose is to map linear systems and possible solutions from one level in the hierarchy to a neighboring one. Translating something from a fine domain down to the next coarser domain is referred to as **restricting**, and the opposite upwards motion from a coarse to a finer domain is referred to as **prolongating**.

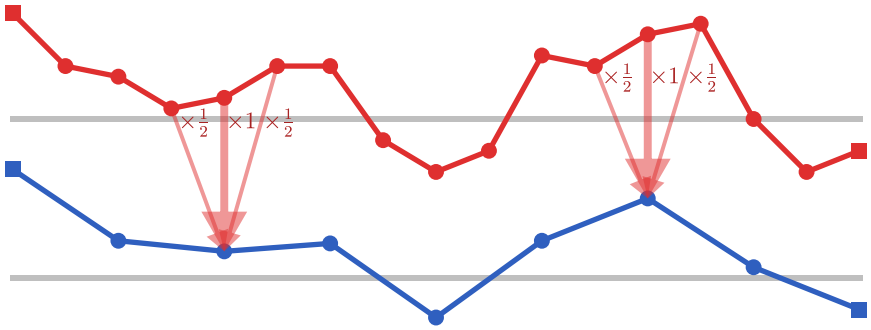


Figure 3.8: Restriction from a fine domain (red) to a coarser one (blue). Two weighted-average calculations are drawn out with arrows. The plots are not drawn to scale; the blue plot is generally twice is in fact generally twice as large vertically as the red plot.

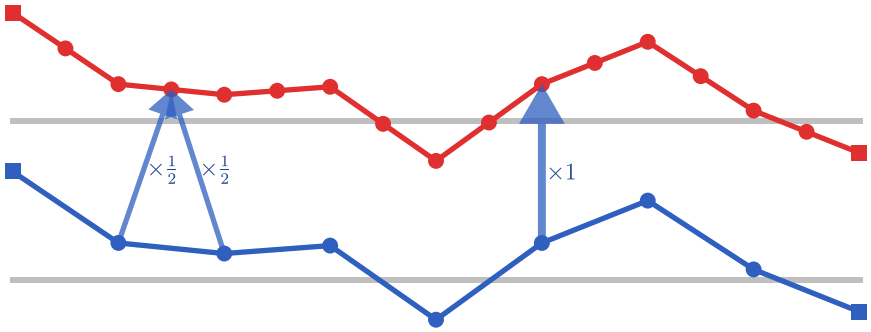


Figure 3.9: Prolongation from a coarse domain (blue) to a finer one (red). Two weighted-average calculations are drawn out with arrows. Note that the fine solution appears as a smoothed version of the one from Figure 3.8.

These two operations are commonly defined by associating points to spatially close points in adjacent levels, and defining their values to be weighted averages of the values of these "neighboring" points. A visualization of this process can be seen in both Figure 3.8 and Figure 3.9. Since both operators are weighted averages defined for each point, they can be represented as a matrix multiplication. This has many benefits, the most important one being that as we restrict a full linear system $A_0\mathbf{x}_0 = \mathbf{b}_0$ to lower levels, it will remain linear on every level, allowing us to still use the same methods to solve the translated problem. We will refer to our restriction operators as matrices $R_1^0, R_2^1, \ldots, R_k^{k-1}$ and to our prolongation operators as $P_{k-1}^k, \ldots, P_1^2, P_0^1$. The prolongation matrices must be chosen such that if we have a solution $\mathbf{x}_1$ on the domain Ω_1, it can be transformed to a similarly-shaped solution $\mathbf{x}_0$ on the domain Ω_0 using just the matrix multiplication

$$\mathbf{x}_0 = P_0^1\mathbf{x}_1.$$

Consequently, P_0^1 must be a matrix of size $|\Omega_0| \times |\Omega_1|$ in order for this multiplication to be well-defined. Similarly, R_1^0 must be a matrix of size $|\Omega_1| \times |\Omega_0|$.

It should be noted that even though restriction and prolongation are considered to be "opposing" operations, the operations are not each other inverses, i.e. it does *not* hold that $R_1^0P_0^1$ or $P_0^1R_1^0$ is equal to the identity matrix. Effectively, this means that restricting a candidate $\mathbf{x}_k$ and then prolongating it back, or vice versa, will not yield $\mathbf{x}_k$ again, as can be observed in the examples in Figures 3.8 and 3.9. However, this serves no problem, as the way in which these operators will be used does not require them to be each other's inverse. In fact, the operators that will be defined later on in the thesis will scale any candidate roughly by a factor of $|\Omega_k|/|\Omega_{k+1}|$, when subsequently restricted and prolongated. The way these operators are going to be used, this will be no issue.

To transfer not the solutions $\mathbf{x}^{(i)}$ but a linear system defined by A and $\mathbf{b}$ to different levels, two things can be done. A first approach would be to rediscretize the PDE from which the original system was

derived, this time on each of the coarser domains Ω_i. This requires knowledge about the original continuous problem from which the linear system was defined, which is not always available, well-defined, or easy to reason about.
The second approach is easier to execute, as it does not rely on information about the original continuous problem. The idea is to use the restriction and prolongation operators to "project" the matrix A and the vector $\mathbf{b}$ onto the coarser domains. Specifically, the calculations to apply are

$$A_{k+1} = R_{k+1}^k A_k P_k^{k+1},$$

$$\mathbf{b}_{k+1} = R_{k+1}^k \mathbf{b}_k.$$

One can see that the multiplication of $\mathbf{b}_k$ happens analogously to the multiplication of an arbitrary solution $\mathbf{x}_k^{(i)}$. To justify the computation of A_{k+1}, one can ponder about the meaning of multiplying the matrix with a vector $\mathbf{x}_{k+1}$; this multiplication can namely be expressed as

$$A_{k+1}\mathbf{x}_{k+1} = R_{k+1}^k A_k P_k^{k+1}\mathbf{x}_{k+1},$$

which can be interpreted as prolongating the solution to the finer level, applying the original matrix A_k, and then restricting the result back to the coarser level. This way of reducing linear systems down the hierarchy is called the **Galerkin method**, and it saves us from the necessity to encode the original continuous problem somehow and to communicate it to the solver, thus extending its applicability to problems where this information is not available.

Since the previously mentioned algorithms generally rely on A_k being symmetric, it can be useful to define

$$R_{k+1}^k = (P_k^{k+1})^\intercal.$$

With this definition, if A_k is symmetric, A_{k+1} will be too. On top of that, this definition also provides the benefit of needing to define only *one* matrix for transfers in both directions; if prolongation is defined, restriction is granted for free with it.

To demonstrate a concrete instance of prolongation, consider once more the hierarchy of regular domains in Figure 3.7. A good prolongation operator for these hierarchies is not hard to come up with. To create a solution on Ω_0 based on a solution on Ω_1, one could transfer the values at each point over to their respective points on the finer level and interpolate to obtain values for the points in between. Effectively, it would mean multiplying a solution $\mathbf{x}_1$ by the matrix

$$P_0^1 = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix}$$

in order to obtain a translated solution $\mathbf{x}_0$. Note that this is the same prolongation operator as demonstrated in Figures 3.8 and 3.9. Similar operators can be defined for each level, and can be reused for restriction as well, allowing any vector or matrix to be transferred to any level in the hierarchy of domains.

3.4.2.3. Multigrid Solver

With the hierarchy of domains and the prolongation operators defined, the solving method can finally be discussed.

Multigrid solvers are similar to iterative solvers in the sense that they rely on multiple subsequent executions of an algorithm, each of them improving a candidate solution until it converges. Unlike the iterations from before, however, these new "iterations" do not just consist of single calls to Jacobi or Gauss-Seidel; instead, they consist of recursive operations called **multigrid cycles**. Multiple different types of multigrid cycles exist, the most common being the *V-cycle*, the *F-cycle*, and the *W-cycle*. To

Algorithm 3 Multigrid V-Cycle with Gauss-Seidel.

function multigridVCycle($k, \mathbf{b}_k, \mathbf{x}_k$)
  if Ω_k is the coarsest layer **then**
    return solve$(A_k, \mathbf{b}_k)$ ▷ Base case, direct solve
  end if
  repeat $preIters$ **times**
    $\mathbf{x}_k \leftarrow$ gaussSeidel$(A_k, \mathbf{b}_k, \mathbf{x}_k)$ ▷ Pre-smoothing
  end
  $\mathbf{r}_k \leftarrow \mathbf{b}_k - A_k\mathbf{x}_k$ ▷ Residual calculation
  $\mathbf{r}_{k+1} \leftarrow (P_k^{k+1})^\intercal \mathbf{r}_k$ ▷ Restriction
  $\mathbf{e}_{k+1} \leftarrow$ multigridVCycle$(k+1, \mathbf{r}_{k+1}, \mathbf{0})$ ▷ Recursive call
  $\mathbf{e}_k \leftarrow P_k^{k+1}\mathbf{e}_{k+1}$ ▷ Prolongation
  $\mathbf{x}_k \leftarrow \mathbf{x}_k + \mathbf{e}_k$ ▷ Error correction
  repeat $postIters$ **times**
    $\mathbf{x}_k \leftarrow$ gaussSeidel$(A_k, \mathbf{b}_k, \mathbf{x}_k)$ ▷ Post-smoothing
  end
  return $\mathbf{x}_k$
end function

demonstrate the structure of a cycle, the V-cycle is explained first. The F-cycle and the W-cycle follow easily from the definition of the V-cycle, as they are simple variations thereof.

The algorithm for the **V-cycle** is described in Algorithm 3. In the usual scenario, the algorithm starts with a couple of iterations of e.g. the Gauss-Seidel method on the solution $\mathbf{x}_k$, mimicking our initial approach. This process is referred to as **pre-smoothing**. After a fixed number of iterations of this, it calculates the *residual vector* $\mathbf{r}_k$, which indicates roughly in what direction $\mathbf{x}_k$ must be moved to become a better solution. Next, it restricts $\mathbf{r}_k$ to $\mathbf{r}_{k+1}$ on the domain Ω_{k+1}, and attempts to solve the **residual equation** $A_{k+1}\mathbf{e}_{k+1} = \mathbf{r}_{k+1}$ on the next coarser domain, by calling the algorithm recursively. The solution $\mathbf{e}_{k+1}$ is then restricted and added to the solution $\mathbf{x}_k$, in an attempt to compensate for the lower-frequency errors still occurring in $\mathbf{x}_k$ even after having been pre-smoothed. Finally, some **post-smoothing** is done to account for any errors introduced by the addition of $\mathbf{e}_k$, and the solution is returned as the new candidate.
Once the coarsest layer is reached, the algorithm no longer relies on this iterative step-by-step plan and instead uses a direct solver to solve the system; since the coarsest layer contains significantly fewer points than the finer layers, the use of a direct solver will not lead to problems at this resolution.

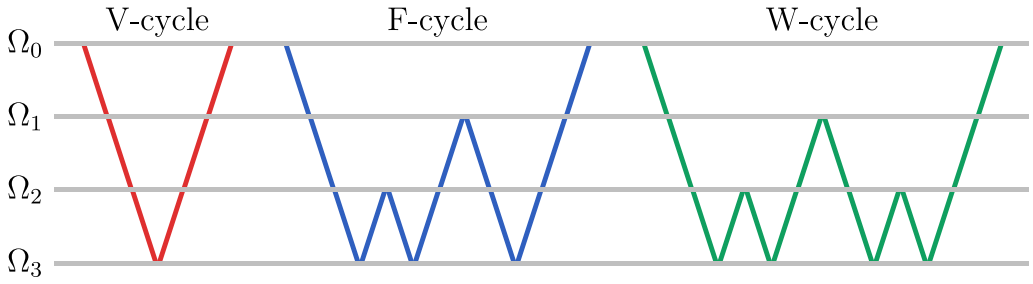


Figure 3.10: A visualization of the transfers between the layers in the hierarchy, as performed by the V-cycle, F-cycle, and W-cycle.

The reason why this structure is called a *V-cycle* becomes clear when visualizing its recursive pattern, as is done in Figure 3.10. Downwards movement represents the restriction of residuals, upwards movement represents the prolongation of error corrections. At the bottom, direct solves are executed, and in any layer above, a few smoothing iterations take place.
Using the same figure, the *F-cycle* and the *W-cycle* can also be explained. The same operations are used, but the pattern in constructing and solving residual equations is different. The **F-cycle** initially

restricts to the coarsest level, and then executes V-cycles of increasingly greater size. The **W-cycle** restricts and prolongates once and recursively executes two smaller W-cycles in between until it reaches the bottom. Both methods apply more smoothing per cycle, gaining in accuracy obtained per cycle, while also increasing the amount of time spent on executing one. As a result, the different cycles each perform differently from one problem to another, and one might want to opt for one type of multigrid cycle over another, depending on what problem and what context is being dealt with.

3.5. Unstructured Meshes

In previous sections, we covered multigrid methods that execute over regular domains. These domains were defined as Cartesian grids with any number of dimensions, which are subdivisions of cuboids. Even our one-dimensional domains fall in this category. If the number of cuboids in each dimension is a particular power of two, coarsening the domain and defining prolongation operators are trivial matters. When dealing with *unstructured* meshes, however, the story becomes *very* different.

An **unstructured mesh** is a mesh without any regular patterns in terms of the placement of the vertices or connectivity between them. An example is that of the surface of an irregular shape, such as a face or a map of a country. Because of the lack of structure, many algorithms and methods that were simple to define for regular meshes are harder to generalize and apply to this new type of mesh. In the context of multigrid methods, the issues arise when considering how to define mesh coarsening and how to construct the prolongation operators, which have now become non-trivial tasks.

3D triangular meshes are meshes that are described by two collections: a list V, containing vectors in $\mathbb{R}^3$ that define positions of its vertices, and a set F, containing triplets of vertices between which a triangle (also referred to as a *face*) is drawn. Usually, the triplets of F consist of the indices at which said vertices are located in V, meaning that F is populated by triplets of integers from 1 up to and including $|V|$ (see Figure 3.11, top). With this definition, the surface of any continuous three-dimensional shape can be approximated up to any level of detail (see Figure 3.12).

3D tetrahedral meshes are similar to triangular meshes, but instead of using a set of faces, they are defined using a set of *tetrahedra* T. These tetrahedra are defined using *quadruplets* of vertex indices, in the same way that faces were defined before (see Figure 3.11, bottom). The added benefit of tetrahedral meshes, in contrast to triangular meshes, is that they allow us to encode and represent *volumes* within shapes, instead of just the shape's outer surface. Particularly for processes in physics simulations, such as heat diffusion or elastic deformation, just reasoning about the outside of a shape is simply not enough to provide a full view of the entire system at play; one must be able to talk about what happens *inside* of a mesh to accurately model these phenomena, and tetrahedra are one way to support this.

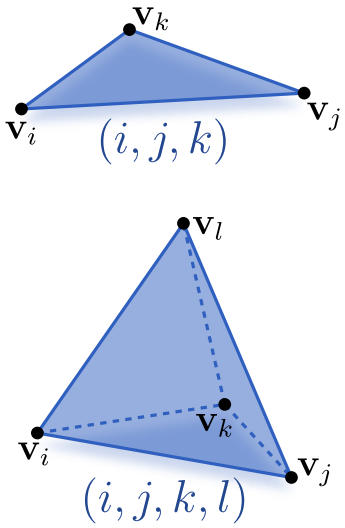


Figure 3.11: A triangle (top) and a tetrahedron (bottom).

When referring to meshes, often a distinction is made between *topology* and *geometry*. We refer to the **topology** of a mesh when talking about the structure defined by E, F, and T. The positions of the vertices in V are not considered in this definition, and thus the actual physical shape of the mesh is disregarded when talking about its topology. The **geometry** of a mesh *does* incorporate the spatial data included by V, and thus allows for the discussion of lengths, areas, angles, and more.

Due to the irregular topology and geometry of unstructured meshes, many problems have to be redefined to remain applicable. For example, our Dirichlet problem, which used to enforce all values to be equal to the average of its neighbors, will now have to work with *weighted* averages instead. These weights will depend on the topology and geometry of the mesh, more specifically one the distribution of mass and the stiffness of the mesh, which is a topic to be further discussed in 5.1.2.

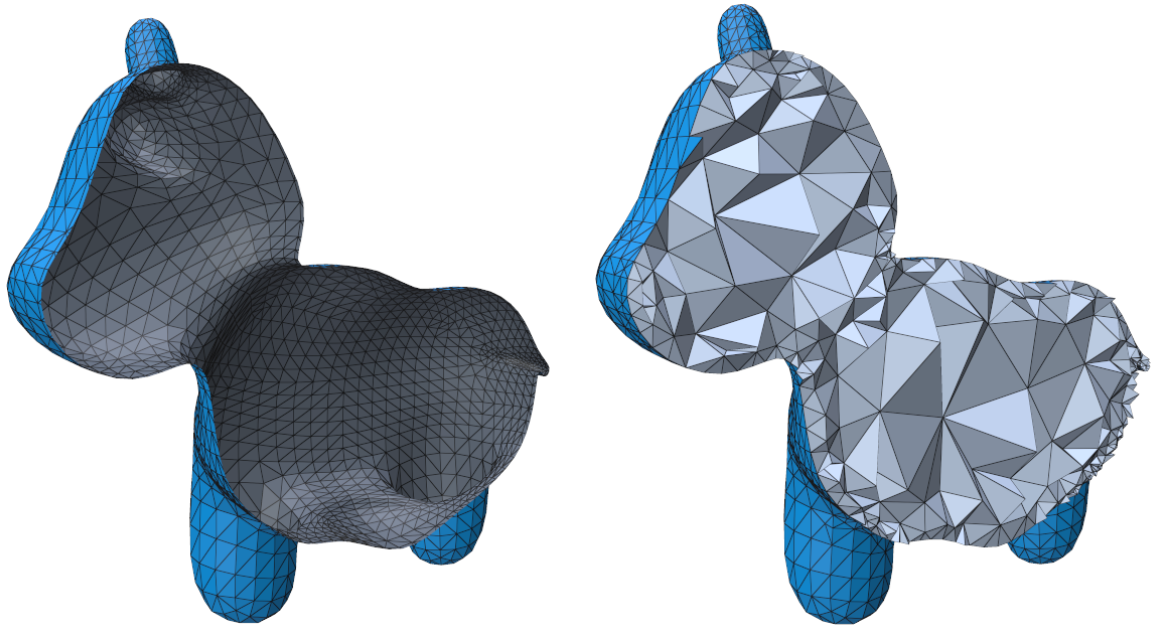


Figure 3.12: A sliced triangular mesh (left) and a sliced tetrahedral mesh (right) with the same contour. `spot` mesh obtained from [CPS13], visualized using *Polyscope* [Sha+19].

3.6. Gravo MG

Defining a geometric multigrid scheme through coarsening on irregular meshes is significantly more complicated than it is on regular meshes. Whereas before we could trivially coarsen the domain by omitting every other point and define prolongation through multilinear interpolation, we can no longer rely on such simple techniques, due to the absence of a regular grid. Nonetheless, geometric multigrid methods on irregular meshes can still be defined by approximating the shape of the original mesh with fewer points and defining prolongation and restriction by matching proximate points between layers. This just requires more thought and consideration than before.

When constructing a multigrid scheme, there are at least two things that are of great importance: the time needed to construct the system for an arbitrary domain, and the time needed to solve an arbitrary equation on the domain. A tradeoff must often be made between these two, as simple and unsophisticated construction techniques could lead to slow convergence, and higher convergence usually demands a coarsening technique that is more expensive to compute. Numerous studies and techniques have been developed to apply multigrid schemes to irregular meshes, often leading to techniques that strike a particular balance between the two relevant durations.

Recently, a new technique has been developed to generate multigrid hierarchies on irregular surface meshes, called **Gravo MG** [Wie+23] (occasionally referred to simply as “Gravo”). In a nutshell, it creates a multigrid hierarchy by recursively coarsening a graph representation of the mesh and constructs a graph Voronoi diagram to quickly triangulate the meshes, well enough to define decently accurate prolongation weights. Gravo MG serves as the “main character” in this thesis, and thus we will spend the rest of this chapter discussing its inner workings in detail.

3.6.1. Algorithm

The Gravo MG algorithm initially starts with a collection of vertices V_0, often referred to as a *point cloud*, and a set of edges E_0 connecting these vertices. A set of triangular faces F_0 may also be available, but it will only serve to determine the set of edges E_0. The output of the algorithm is a sequence of prolongation matrices $P_1^0, \dots, P_k^{k-1}$, which can be plugged into any multigrid solver.

In a nutshell, the algorithm iteratively performs the following five steps to generate a new level k from an existing level $k-1$:

1. Subsample the point cloud V_{k-1} to obtain a point cloud V_k, such that $|V_k| \ll |V_{k-1}|$.
2. Cluster the points in V_{k-1} into clusters C_i, each associated to a vertex $\mathbf{v}_i$ from V_k.
3. Connect the points in V_k with an edge set E_k based on the clusters C_i.
4. Triangulate V_k using E_k to create a set F_k.
5. Prolongate V_{k-1} by construct a prolongation operator P_{k-1}^k based on F_k, E_k and V_k.

This process is repeated until V_k is sufficiently small, which we consider to be the case once it contains at most 1000 vertices.

Each of these steps will be discussed in further detail in Sections 3.6.1.1, 3.6.1.2, and 3.6.1.3.

3.6.1.1. Subsampling

Gravo MG attempts to subsample a point cloud V_k from V_{k-1} by applying variation on the **Maximum Delta-Independent Set** sampling strategy, abbreviated as **MδIS**. The algorithm that finds such an independent set follows a few steps. First, the average length $\hat{e}$ of all edges in E_{k-1} is calculated. Next, a predefined ratio $0 < \phi < 1$ is used alongside it to compute a value δ using the formula

$$\delta = \phi^{-\frac{1}{3}} \hat{e}.$$

The algorithm will treat δ as a minimum distance to maintain between the points in V_k. Then, the algorithm will repeatedly select a random available point from V_{k-1}, add it to V_k, and eliminate all its direct neighbors (so the vertices located in its **one-ring**, seen in Figure 3.14) for which the Euclidean distance to the sampled points is smaller than δ. This continues until all points are either sampled or eliminated. The result is a new point cloud V_k containing significantly fewer points than V_{k-1}.

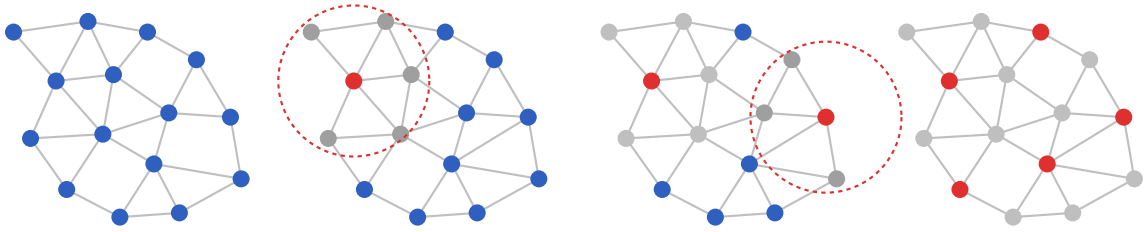


Figure 3.13: The MδIS-1 algorithm, visualized. The red points will form the new domain.
From left to right: 0 steps, 1 step, 2 steps, and a possible end result.

Unlike the *Maximum Independent Set* (*MIS*) problem, which often occurs in more algorithmic settings, Maximum Delta-Independent Set intentionally ignores connections with a length of at least δ. By doing this, it incorporates information about the geometry of the mesh to ensure that the subsampling remains locally approximately uniform. After all, imperfect tetrahedralizations could contain particularly long edges, which would cause conflicts between vertices that are not remotely close to each other and thus should not mutually exclude each other. For this reason, MδIS is made use of here instead of MIS.

[Wie+23] applies an adaptation of MδIS, where not only direct neighbors in close proximity are eliminated, but also the neighbors of those neighbors, i.e. vertices in its **two-ring** (see Figure 3.14). By limiting the search to the two-ring, the runtime is not increased too much, while still eliminating significantly more points within the desired radius, achieving uniformity faster. When distinction is needed, we will refer to this adaptation as **MδIS-2**, while referring to the original one-ring version as **MδIS-1**.

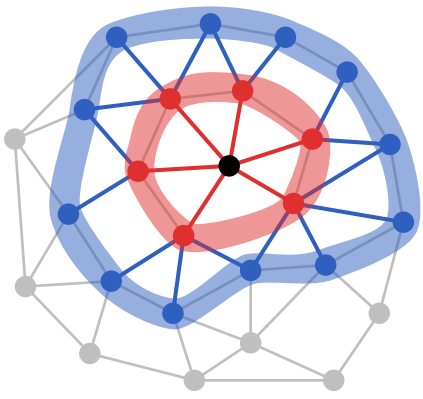


Figure 3.14: The one-ring (red) and the two-ring (blue) of a vertex (black).

It should be noted that the ratio between the sizes $|V_k|$ and $|V_{k-1}|$ is likely not equal or even close to the ratio ϕ, since δ is computed heuristically, and does not guarantee that the desired number of vertices is sampled. In practice, MδIS-1 often samples significantly more points, and MδIS-2 significantly fewer. This phenomenon will be revisited in Section 4.2.3.1 in more detail.

3.6.1.2. Clustering, Connecting and Triangulating

The output of subsampling a point cloud V_{k-1} is a new point cloud V_k, devoid of any edges of triangles connecting the points. Gravo MG will attempt to construct a triangular mesh based on the new point cloud and the previous mesh in three steps. The whole process is visualized in Figure 3.15.

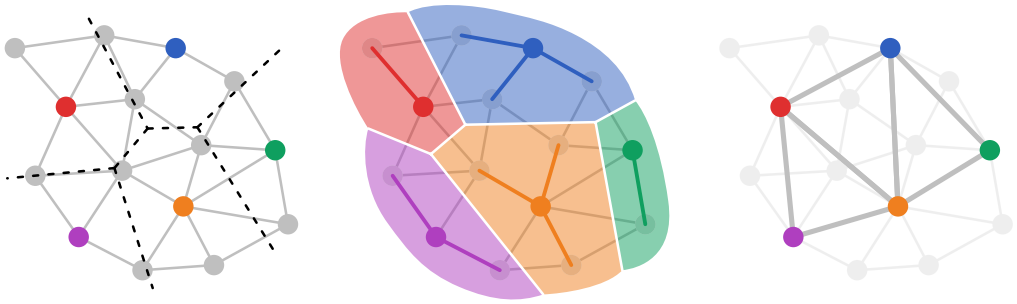


Figure 3.15: The triangulation process demonstrated on the mesh from Figure 3.13.
Left: The Voronoi cells that are approximated using Graph Voronoi. The coarse points (colored) are used as centroids.
Center: The clustering of all fine points, based on their proximity to the selected coarse points.
Right: Coarse edges drawn between coarse points whose clusters are connected by at least one fine edge, and the triangles found within these coarse edges.

First, it will cluster the points in V_{k-1} over different clusters C_i, where each cluster is associated with a particular point $\mathbf{v}_i$ from the point cloud V_k. The remaining not-sampled points, those in $V_{k-1} \setminus V_k$, are divided over these clusters based on a **graph Voronoi diagram**. In essence, this means that a fine point is assigned to the cluster of the coarse point closest to it, where the distance between two points is defined as the length of the shortest edge path connecting them, measured as the sum of the lengths of the edges in the path. This assignment, along with the relevant closest distances, can be computed for every point using a multi-source variation of Dijkstra's algorithm, starting from each coarse point.

Second, after the clusters have been created, edges are drawn between every pair of coarse points $v_i, v_j \in V_k$ if there exists an edge in E_{k-1} that connects a point in C_i to a point in C_j. Phrased differently, the points are connected if their clusters are direct neighbors. The drawn edges will form the edge set E_k that connects the point cloud V_k.

Algorithm 4 Triangulating a point cloud using a set of vertices V and a set of edges E.

function triangulate(V, E)
  $N_1 \leftarrow \emptyset, \ldots, N_{|V|} \leftarrow \emptyset$
  for $(i, j) \in E$ **do** ▷ Construct sets of neighboring vertices
    $N_i \leftarrow N_i \cup \{\mathbf{v}_j\}$
    $N_j \leftarrow N_j \cup \{\mathbf{v}_i\}$
  end for
  $F \leftarrow \emptyset$
  for $\mathbf{v}_i \leftarrow V$ **do**
    for $\mathbf{v}_j \leftarrow \{\mathbf{v}_j \in N(\mathbf{v}_i) \mid i < j\}$ **do** ▷ Inequality included as symmetry break
      for $\mathbf{v}_k \leftarrow \{\mathbf{v}_k \in N(\mathbf{v}_i) \mid j < k \wedge \mathbf{v}_k \in N(\mathbf{v}_j)\}$ **do**
        $F \leftarrow F \cup \{(i, j, k)\}$ ▷ Add face to set
      end for
    end for
  end for
  return F
end function

Finally, the mesh is triangulated by searching for triplets of unique vertices in V_k such that each pair of vertices has an edge between them. An efficient implementation of this process is outlined in Algorithm 4. All triangles are then stored in the set F_k, and will be used in the prolongation step discussed in Section 3.6.1.3. It should be noted that it is extremely likely that the resulting triangulation is not watertight and contains holes. This, however, will not cause any significant problems in later stages of the algorithm, since prolongation operators can still be defined for these meshes, even if the transfer is a bit less accurate than it would be for a watertight mesh.

3.6.1.3. Prolongating

After having executed the preceding steps, Gravo MG should have definitions for V_{k-1}, V_k, E_k, and F_k at its disposal. Using these, it will define the prolongation from level k to level $k-1$. The value of each point from V_{k-1} will be determined by a weighted average of the values of at most three points from V_k. For each fine point $\mathbf{v} \in V_{k-1}$, it decides on those points and their corresponding weights by first determining the coarse point $\mathbf{v}_c \in V_k$ representing the cluster C_c that $\mathbf{v}$ is in, and then traversing the following checklist:

1. Find the closest triangle $f \in F_k$ containing $\mathbf{v}_c$ such that $\mathbf{v}'$, which is the projection of $\mathbf{v}$ onto the plane spanning f, falls onto f (see Figure 3.16). Prolongate using the three vertices of f, and use the barycentric coordinates of $\mathbf{v}'$ inside f as the prolongation weights. If no such triangle exists, continue to Step 2.
2. Find the closest edge $e \in E_k$ containing $\mathbf{v}_c$ such that $\mathbf{v}'$, which is the projection of $\mathbf{v}$ onto the line spanning e, falls onto e (see Figure 3.16). Prolongate using the two endpoints of e, and use the linear interpolation weights of $\mathbf{v}'$ inside e as the prolongation weights. If no such edge exists, continue to Step 3.
3. Select the three points in V_k that are closest to $\mathbf{v}$. Prolongate using these points, and use weights that are proportional to their inverse distance to $\mathbf{v}$.

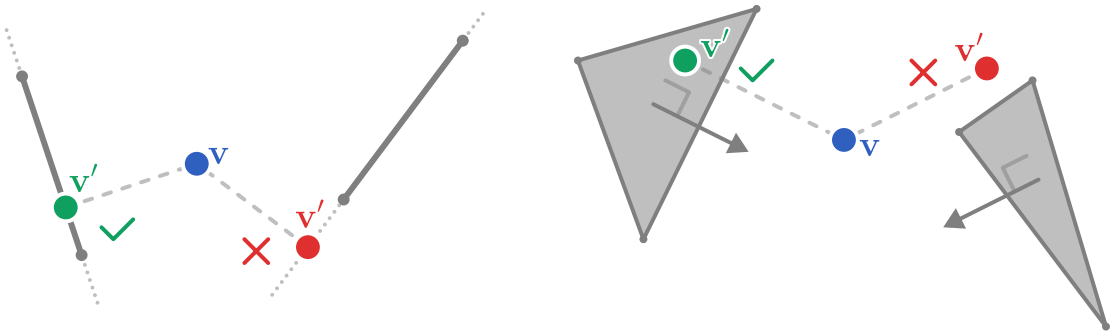


Figure 3.16: The projection of a point $\mathbf{v}$ onto edges and triangles. If the projection falls inside the primitive, the primitive can be used for prolongation with linear or barycentric weights.

To determine whether $\mathbf{v}'$ is inside an edge e or not, one can determine whether all the linear interpolation weights of $\mathbf{v}$ inside e are non-negative. The computation of the barycentric coordinates inside edges is outlined in Algorithm 5. Similarly, for triangles, one can determine whether the barycentric coordinates of $\mathbf{v}$ inside f are all non-negative, and these coordinates can be calculated using Algorithm 6.

By only considering triangles and edges that connect to $\mathbf{v}_c$, the computation is sped up significantly, as it saves us from having to iterate over the entire mesh. Instead, the search is limited to a small neighborhood surrounding the coarse point.

The triangulation of the point cloud at level k allows Gravo MG to prolongate using *barycentric coordinates* for the majority of the fine points. This generally leads to quicker convergence than prolongation using inverse distance weights, which can be computed with only a point cloud, but do not serve as an accurate parametrization of the surface to use for interpolation.

Algorithm 5 Calculating the linear interpolation weights of a point $\mathbf{v}$ inside an edge spanning vertices $\mathbf{v}_0$ and $\mathbf{v}_1$, if projected onto the edge's line.

function linearWeightsEdge($\mathbf{v}, \mathbf{v}_0, \mathbf{v}_1$)
  $\lambda_0 \leftarrow \frac{(\mathbf{v}-\mathbf{v}_0)\cdot(\mathbf{v}_1-\mathbf{v}_0)}{\|\mathbf{v}_1-\mathbf{v}_0\|^2}$
  $\lambda_1 \leftarrow 1-\lambda_0$ ▷ Compute remaining weight
  return (λ_0, λ_1)
end function

Algorithm 6 Calculating the barycentric coordinates of a point $\mathbf{v}$ inside a triangle spanning vertices $\mathbf{v}_0$, $\mathbf{v}_1$ and $\mathbf{v}_2$, if projected onto the triangle's plane.

function baryCoordsTri($\mathbf{v}, \mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$)
  $\mathbf{n} \leftarrow (\mathbf{v}_1-\mathbf{v}_0)\times(\mathbf{v}_2-\mathbf{v}_0)$ ▷ Triangle normal
  $a \leftarrow \|\mathbf{n}\|$ ▷ Triangle surface area times two
  $\hat{\mathbf{n}} \leftarrow \mathbf{n}/a$ ▷ Triangle normal of unit length
  $d \leftarrow (\mathbf{v}-\mathbf{v}_0)\cdot\hat{\mathbf{n}}$ ▷ Distance from $\mathbf{v}$ to triangle
  $\mathbf{v}' \leftarrow \mathbf{v}-d\hat{\mathbf{n}}$ ▷ Projection of $\mathbf{v}$ onto triangle
  $\lambda_0 \leftarrow \frac{1}{a}((\mathbf{v}_2-\mathbf{v}_1)\times(\mathbf{v}'-\mathbf{v}_1))\cdot\mathbf{n}$
  $\lambda_1 \leftarrow \frac{1}{a}((\mathbf{v}_0-\mathbf{v}_2)\times(\mathbf{v}'-\mathbf{v}_2))\cdot\mathbf{n}$
  $\lambda_2 \leftarrow 1-\lambda_0-\lambda_1$ ▷ Compute remaining coordinate
  return $(\lambda_0, \lambda_1, \lambda_2)$
end function

3.6.2. Conclusion

By constructing domains using primarily point cloud data, followed by a quick triangulation of these point clouds using graph Voronoi for the purpose of defining the prolongation operators, Gravo MG achieves a representative and useful multigrid hierarchy in a relatively short time. It hence solves systems quicker than approaches that operate and prolongate solely on point cloud data, such as [Shi+06], and takes less time to set up than approaches that carefully maintain a triangulation throughout the entire coarsening process, such as [Liu+21].

4

Methods

In this chapter, the main contribution of the thesis is provided, which is a description of the implemented changes to the original Gravo MG algorithm from [Wie+23] in order to support tetrahedral meshes. In Section 4.1, the strictly necessary changes for transitioning from the triangular to the tetrahedral context are outlined, Section 4.2 explores ideas on how to preserve the boundary of the domains throughout the hierarchy, and 4.3 briefly discusses the parallelization employed to improve runtime.

4.1. The Essentials

When considering each of the individual steps of the original Gravo MG algorithm, it turns out that many of them do not rely on the assumption that the given point cloud represents a triangular mesh, not a tetrahedral one. Stages such as subsampling, clustering, and connecting each remain unchanged and untouched. Even though the two remaining steps, triangulating and prolongation, can in theory also be left as they are without causing blocking problems, it would be nonsensical to look for triangles in a tetrahedral mesh and use those for prolongation. If prolongation for fine points depends on triangles and not on tetrahedra, a lot of spatial information about the mesh is disregarded, which affects convergence negatively. These two stages of the algorithm thus need to be adapted. The required changes for both are discussed in Sections 4.1.1 and 4.1.2, respectively.

4.1.1. Tetrahedralizing

The first essential change is that instead of *triangulating* the point cloud formed by V_k and E_k, it needs to be *tetrahedralized*. The tetrahedralization process is similar to the triangulation process discussed in Section 3.6.1.2, with the difference being that we are looking for a group of *four* connected vertices, instead of three. These groups will form a set of tetrahedra T_k, which give shape to the mesh on the next layer.
Alongside the tetrahedralization T_k, the triangulation F_k is still generated as well, since the triangle data can still be used as backup in the prolongation step, if prolongation based on tetrahedra fails. This is likely to happen close to the boundary of the mesh, for example. The triangulation can be easily computed simultaneously with the tetrahedralization. The pseudocode for generating both a triangulation and a tetrahedralization can be found in Algorithm 7.

The tetrahedralizations that Gravo MG generates are far from perfect. An example can be seen in Figure 4.1. Even if the input mesh is watertight, has no overlaps, and appears to perfectly describe a certain shape, the subsampling and the rapid coarse reconstruction of this shape will lead to many anomalies in the domain. These anomalies include visible pits in the surface, gaps on the interior of the mesh, and tetrahedra that partially intersect and thus both occupy the same small space within the domain. Particularly, the intersection of tetrahedra is something that would never be seen in a mesh that was carefully prepared to perform physics simulation on, since this would already cause ambiguity issues in the definition of the problem, which usually relies on the assumption that the mesh can be embedded in space.

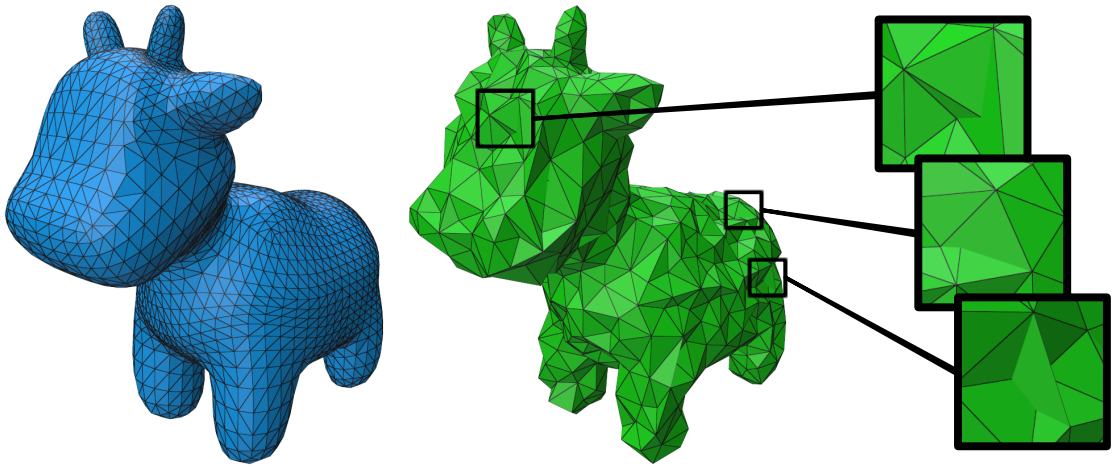


Figure 4.1: The finest domain in a multigrid hierarchy (left) and the next coarser domain, tetrahedralized with Gravo MG. Some visible tetrahedron intersections are highlighted, though many more exist inside the mesh, not visible from the outside.

While it might be tempting to blame the occurrence of intersections on the usage of graph Voronoi methods over actual Voronoi methods, it should be pointed out that using perfect Voronoi cells would not mitigate the issue. To this end, imagine a subsampled point cloud of five coarse points in 3D space, positioned as seen in Figure 4.2, each with an implied Voronoi cell acting as the point's cluster. In three dimensions, it is possible for these five clusters to all directly neighbor each other. The result is a *clique* of five points in which edges are added between every single pair. As a result, five different tetrahedra will be generated, even though only two or three of these tetrahedra are needed to fill the space. This is just one example of how intersecting tetrahedra can be created. In practice, they can also be generated under different circumstances, which are harder to predict and even harder to prevent.

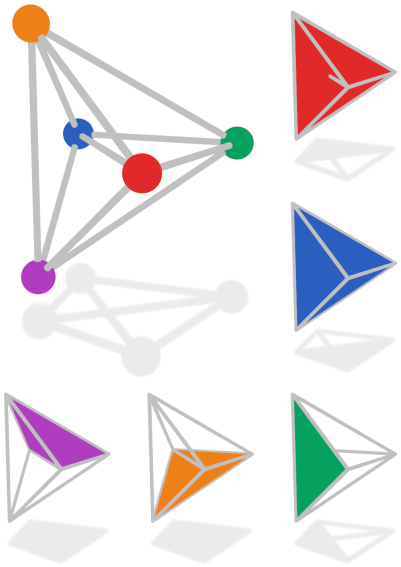


Figure 4.2: A five-point clique naturally occurring when connecting a point cloud. This simple structure will cause the generation of five tetrahedra, each of which overlaps with some of the others.

Even though intersecting tetrahedra may sound problematic, they turn out not to form a real issue for our use case. First of all, these intersections only appear in the intermediate coarser representations, which are solely used for computing, and not in any kind of visible final output. Second of all, the tetrahedra themselves are only used for defining the prolongation operators (as will be further discussed in Section 4.1.2), and overlaps will just lead to multiple options for this definition for some of the points, at which point an arbitrary selection can be made. Since the effort needed to cleanly retetrahedralize the mesh is not trivial and would significantly increase Gravo's low construction time, one of its most valuable qualities, these "bad" tetrahedralization are not improved and are used as they are given.

4.1.2. Prolongating

The prolongation operators for the adapted Gravo algorithm are defined similarly as in Section 3.6.1.3. A checklist is still traversed to find the best possible prolongation for a given fine point. To support tetrahedral meshes, however, a new step is added at the very start, which is to be checked first before continuing with the default cases. In the stepwise algorithm below, Step 1 is the newly added step, and the remaining steps have been unchanged. As a reminder, $\mathbf{v}$ refers to the fine point for which to define the prolongation, $\mathbf{v}_c$ refers to the coarse point that represents $\mathbf{v}$'s cluster, and V_k, E_k, F_k, and T_k define the mesh.

Algorithm 7 Tetrahedralizing a connected point cloud using a set of vertices V and a set of edges E.

function tetrahedralize(V, E)
  $N_1 \leftarrow \emptyset, \ldots, N_{|V|} \leftarrow \emptyset$
  for $(i, j) \in E$ **do** ▷ Construct sets of neighboring vertices
    $N_i \leftarrow N_i \cup \{\mathbf{v}_j\}$
    $N_j \leftarrow N_j \cup \{\mathbf{v}_i\}$
  end for
  $F \leftarrow \emptyset$
  $T \leftarrow \emptyset$
  for $\mathbf{v}_i \leftarrow V$ **do**
    for $\mathbf{v}_j \leftarrow \{\mathbf{v}_j \in N(\mathbf{v}_i) \mid i < j\}$ **do** ▷ Inequality included as symmetry break
      for $\mathbf{v}_k \leftarrow \{\mathbf{v}_k \in N(\mathbf{v}_i) \mid j < k \wedge \mathbf{v}_k \in N(\mathbf{v}_j)\}$ **do**
        $F \leftarrow F \cup \{(i, j, k)\}$ ▷ Add face to set
        for $\mathbf{v}_l \leftarrow \{\mathbf{v}_l \in N(\mathbf{v}_i) \mid k < l \wedge \mathbf{v}_l \in N(\mathbf{v}_j) \wedge \mathbf{v}_l \in N(\mathbf{v}_k)\}$ **do**
          $T \leftarrow T \cup \{(i, j, k, l)\}$ ▷ Add tetrahedron to set
        end for
      end for
    end for
  end for
  return (F, T)
end function

1. Find a tetrahedron $t \in T_k$ containing $\mathbf{v}_c$ such that $\mathbf{v}$ lies within t. In case multiple tetrahedra contain $\mathbf{v}$, arbitrarily pick one of them. Prolongate using the four vertices of f, and use the barycentric coordinates of $\mathbf{v}$ inside t as the prolongation weights. If no such tetrahedron exists, continue to Step 2.
2. Find the closest triangle $f \in F_k$ containing $\mathbf{v}_c$ such that $\mathbf{v}'$, which is the projection of $\mathbf{v}$ onto the plane spanning f, falls onto f. Prolongate using the three vertices of f, and use the barycentric coordinates of $\mathbf{v}'$ inside f as the prolongation weights. If no such triangle exists, continue to Step 3.
3. Find the closest edge $e \in E_k$ containing $\mathbf{v}_c$ such that $\mathbf{v}'$, which is the projection of $\mathbf{v}$ onto the line spanning e, falls onto e. Prolongate using the two endpoints of e, and use the linear interpolation weights of $\mathbf{v}'$ inside e as the prolongation weights. If no such edge exists, continue to Step 4.
4. Select the three points in V_k that are closest to $\mathbf{v}$. Prolongate using these points, and use weights that are proportional to their inverse distance to $\mathbf{v}$.

Again, to determine whether $\mathbf{v}$ is inside t or not, one can determine whether all the barycentric coordinates of $\mathbf{v}$ inside t are non-negative. The computation of the barycentric coordinates inside tetrahedra is outlined in Algorithm 8.

Algorithm 8 Calculating the barycentric coordinates of a point $\mathbf{v}$ inside a tetrahedron spanning vertices $\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{v}_3$.

function baryCoordsTet($\mathbf{v}, \mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$)
  for $(a, b, c, d) \in \{(0, 1, 2, 3), (1, 2, 3, 0), (2, 3, 0, 1)\}$ **do** ▷ Cycle over ordered quadruplets of vertices
    $\mathbf{n} \leftarrow (\mathbf{v}_c - \mathbf{v}_b) \times (\mathbf{v}_d - \mathbf{v}_b)$ ▷ Triangle normal
    $\hat{\mathbf{n}} \leftarrow \frac{\mathbf{n}}{\|\mathbf{n}\|}$ ▷ Triangle normal of unit length
    $\lambda_a \leftarrow \frac{\hat{\mathbf{n}} \cdot (\mathbf{v} - \mathbf{v}_b)}{\hat{\mathbf{n}} \cdot (\mathbf{v}_a - \mathbf{v}_b)}$
  end for
  $\lambda_3 \leftarrow 1 - \lambda_0 - \lambda_1 - \lambda_2$ ▷ Compute remaining coordinate
  return $(\lambda_0, \lambda_1, \lambda_2, \lambda_3)$
end function

4.2. Boundary Preservation

In this section, a few techniques are proposed to better preserve the shape of the domain through coarsening. The techniques all rely on a notion of the *boundary*. Section 4.2.1 will give a precise description of this term, 4.2.2 will describe our approach to detecting the boundary for an arbitrary hierarchy, and 4.2.3 will show some techniques that use the boundary information to construct better hierarchies.

4.2.1. Concept

The **boundary** of a triangular mesh is a region on the mesh at which the mesh appears to simply stop, or be "cut" in a way. Examples can be a hole that is punched through the mesh or the silhouette of a flat sheet. More formally, a boundary can be identified as a collection of edges that are adjacent to just *one* triangle in the mesh, instead of being located in between two of them. It is possible for a triangular mesh to not have a boundary at all. A simple cube, to take a concrete example, only has edges that reside between two triangles, and thus, no boundary exists according to the definition. In fact, the surface of any physical object around us represents a mesh without a boundary. Because of this fact, often watertight meshes, i.e. meshes without a boundary, are preferred in physical simulation or realistic rendering, meaning that a notion of a boundary is not that important.

Tetrahedral meshes also have their own definition of a boundary; it is commonly referred to as the **surface** of the mesh. A tetrahedral mesh *always* has a boundary, since, unlike 2D surfaces embedded in a 3D space, which can curve back to itself, 3D volumes in a 3D Euclidean space must "end" somewhere. There is guaranteed to be a certain nonempty subset of vertices that lie on the outer surface of the shape, and because these vertices are not surrounded by vertices on all sides, they tend to behave differently from the interior vertices when doing various computations on them. For example, when iteratively adjusting the positions of vertices to be the average of their neighbors, a region of interior vertices might shuffle in any direction, and gradually attempt to converge to an equilibrium, whereas a vertex on the surface would tend to gravitate further towards the inside of the mesh, inadvertently shrinking it. The fundamentally different behavior of boundary points gives reason to determine what the general shape of the boundary is.

We make a distinction between three different types of boundary points, which will be called *surface points* (or 1^{st} rank boundary points), *ridge points* (2^{nd} rank boundary points), and *corner points* (3^{rd} rank boundary points). **Surface points** are points that simply lie on the boundary and are exposed to "negative space". Most can be seen on the outside of the mesh, but they could also occur in a cavity within it. **Ridge points** are surface points that lie on a particularly curved part of the surface, where the mesh appears to be folded by a significant angle. In a sense, a ridge could be considered "a boundary of a boundary" of sorts. **Corner points** are surface points that find themselves in a particularly pointy part of the surface, and could be considered "a boundary of a ridge". Each of these types of boundary points is visualized in Figure 4.3.

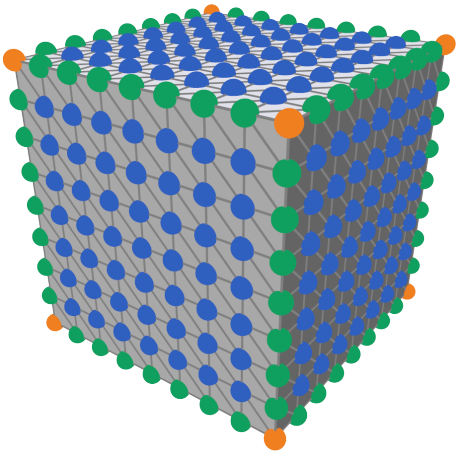


Figure 4.3: Boundary vertices on a cube mesh of different ranks. Blue points are 1^{st} rank boundary points, green points are 2^{nd} rank, and orange points are 3^{rd} rank.

4.2.2. Computation

This section will describe the process to determine the boundary for a given mesh. This process has a few different stages and aspects to it, each of which is described in the following subsections.

4.2.2.1. Initialization

Gravo MG does not come with a baked-in algorithm for determining what points are surface points, ridge points, or corner points in the input mesh. Instead, this information is left for the user to provide as additional context for the mesh. Gravo can still function without this information, although the techniques described in Section 4.2.3 can then no longer be executed.

The decision to leave the boundary definition for the input mesh to the user was made for several reasons. First of all, Gravo is, by design, unaware of the surface of the mesh, as it only has information regarding the vertex positions defined by V and their pairwise connections defined by E. Gravo can at best make a guess at what the boundary is supposed to be, and this might lead to estimations of the boundary that are less than ideal. By giving this control to the user, they can decide to manually paint the surface with ridges and corners to prioritize some details of the mesh over others, or even delegate the work to a *different* algorithm, with more sophisticated detection methods beyond the scope of this work.

However, in the event that the original mesh is known, including the tetrahedra and the faces that form the surface, we can still propose a brief method to determine the boundary of the input that is provided to Gravo. What will be described next is the detection method used in our *experiments* to determine the boundary of an arbitrary manifold tetrahedral mesh. These methods are executed as a preprocessing step before Gravo is run itself, and give a good enough estimation of the boundary in the majority of cases.

Surface points can be detected using just the topology of the domain. Consider a tetrahedralization of a shape in which no points share the same location, no tetrahedra overlap with each other, every point is used to define at least one tetrahedron, and all tetrahedra are non-degenerate, i.e. their points are not coplanar and thus their volume is non-zero. For these kinds of "good" tetrahedralizations, there exists a straightforward topological approach to determine what points lie on the surface of the shape it defines. The key is to first find the *faces* that lie on the surface, which are distinguished by the fact that they only lie on *one* tetrahedron instead of two, and to mark every point that lies on any of the found faces. All marked points together form the set of surface points. A formal algorithm is described in Algorithm 9.

Algorithm 9 Given a set of tetrahedra T that describe the topology of a mesh (V, T), determines a set of surface point indices $\mathrm{Surf} \subseteq \{0, \ldots, |V| - 1\}$.

function determineSurface(T)
  $cnt \leftarrow \{\}$ ▷ Counter data structure, initializes everything with 0
  for $(i, j, k, l) \in T$ **do**
    Increment $cnt[\{i, j, k\}]$
    Increment $cnt[\{i, j, l\}]$
    Increment $cnt[\{i, k, l\}]$
    Increment $cnt[\{j, k, l\}]$
  end for
  $F^{\mathrm{Surf}} \leftarrow \{f \in cnt \mid cnt[f] = 1\}$ ▷ Faces on the surface would only be encountered once
  $\mathrm{Surf} \leftarrow \emptyset$
  for $\{i, j, k\} \in F^{\mathrm{Surf}}$ **do**
    $\mathrm{Surf} \leftarrow \mathrm{Surf} \cup \{i, j, k\}$
  end for
  return Surf
end function

Ridge points we detect by iterating over each of the edges that connect two surface triangles, and determining the angle that the two triangles make with each other. To this end, their triangle normals are computed, and the angle between these two normals is measured. If this angle is bigger than our selected value $\frac{1}{3}\pi$, the two endpoints of the edge are marked as ridge points. Otherwise, they are not.

Finally, *corner points* are detected by computing the *Gaussian curvature* of each of the surface points, which can be used as a measure of how pointy a mesh is at a certain point. The process of computing the Gaussian curvature is described in Section 4.2.2.3. A point is considered a corner point if its Gaussian curvature exceeds our selected value $\frac{1}{3}\pi$.

4.2.2.2. Propagation

The boundary initialization algorithms designed to detect boundary features, as described in Section 4.2.2.1, work well for properly tetrahedralized meshes with a manifold surface and without self-intersections, and can thus be applied to find the boundary points in the original domain at the finest

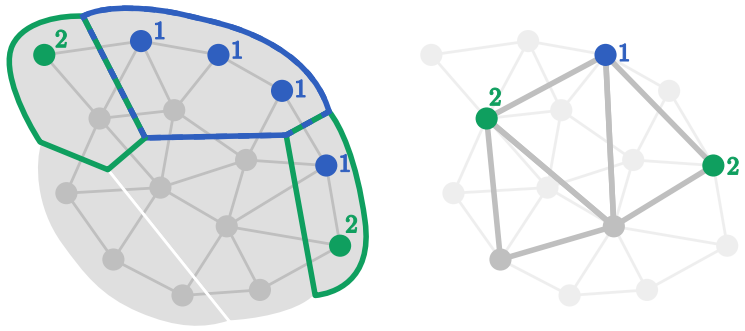
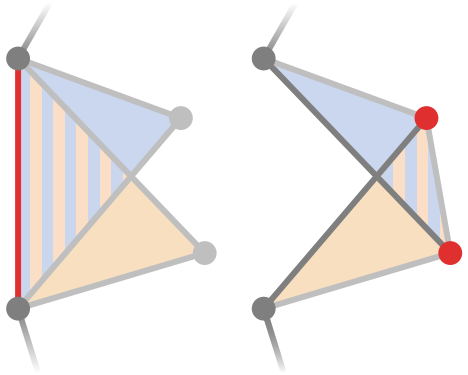


Figure 4.4: A coarse vertex is assigned to be a boundary point of the highest rank that can be found within its cluster.

level as a preprocessing step. However, for the coarser and messier domains in the multigrid hierarchy, these algorithms are thrown out of the window in favor of a much simpler approach. On the lower levels of the hierarchy, we consider a coarse point to be a boundary point of *at least* a certain rank if there is at least *one* fine point in the cluster associated with that same boundary rank. The coarse boundary thus directly depends on the finer boundary. A visualization is shown in Figure 4.4.

There are two substantial benefits to the propagational technique introduced here. First, it sidesteps edge cases that would occur from performing all the above algorithms on the poor tetrahedralizations that might arise. An example of such an edge case would be two intersecting tetrahedra that share a face on the surface, as demonstrated in Figure 4.5a. While these edge cases can be detected and solved by analyzing the geometry of the mesh as well as its topology, this would cause a lot of computational overhead, ultimately making it not worth the effort. Second, using a recursive definition means that the boundary information depends completely on the boundary given at the finest level, which the user gets to define themselves. This gives them almost complete control over the definition of the boundary on every level in the hierarchy, by just providing it at the highest level. Additionally, this approach completely removes the dependence the algorithm would have on the set of tetrahedra in the original domain. This independence would mean that Gravo MG could be run on just a point cloud, similarly to the original Gravo MG algorithm from [Wie+23], as long as a guess at what the boundary should be is provided.



(a) A side view of two intersecting tetrahedra (blue, orange) sharing a triangle (red). This triangle would topologically not be classified as a surface triangle.

(b) A side view of two intersecting tetrahedra (blue, orange) with intersecting faces, causing ambiguity about whether the red points are located on the surface.

It should be noted that due to the poor coarse tetrahedralizations, points that reside a bit below the surface and are arguably *interior* points, could still be marked as a *boundary* point instead, for example, in the scenario depicted in Figure 4.5b. This error has been observed to cause even deeper “boundary” points in subsequent coarser domains, making the computation of the surface a bit unreliable. However, all the operations that will be described in Section 4.2.3 use the boundary solely for heuristic purposes, and never as a cornerstone for any crucial operation. Slight mistakes or ambiguities in the boundary thus have no disastrous effects for the execution of the algorithm as a whole.

4.2.2.3. Promotion

While coarsening a mesh, it might happen that a curved section that did not get marked as a corner point initially gets sharper after points are removed. In this event, it would be good if a point is “promoted” to a corner point, even if there was no fine corner point in its cluster, so that the newly-developed sharp feature of the mesh is preserved. To figure out whether a point should be promoted, an estimation of the *Gaussian curvature* of the mesh around that point must be made.

The **Gaussian curvature** of a point on a clean and manifold mesh describes how curved the surface around that point is. In discrete settings, it is often defined as the difference between a complete angle ($360°$ or 2π) and the sum of internal angles around the vertex. If the region surrounding the point is perfectly flat, these two will be equal, and the curvature will be 0. An extruded point will have a positive curvature, and a point located in a saddle will have a negative curvature. While this definition of curvature is useful, it relies on a well-triangulated exterior, which is not available to Gravo on the coarse levels. The messy tetrahedralization and triangulation make this measure unreliable. However, using just information about the vertices and about the connections between them, an accurate estimation of the curvature can still be made.

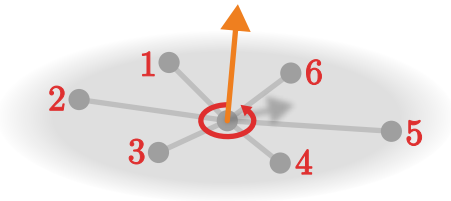


Figure 4.6: Neighbors sorted around the normal.

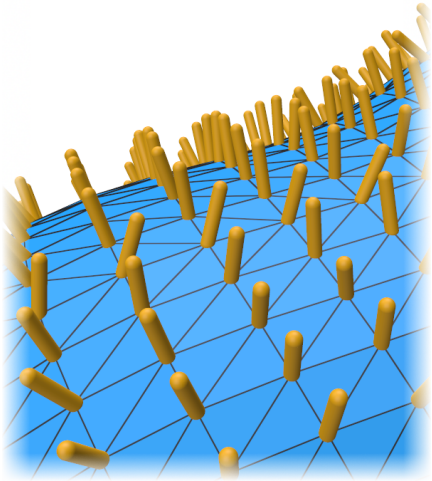


Figure 4.7: The estimated normals sticking out of the surface, not always quite perpendicularly.

Suppose that we have a set of points and edges connecting them, and for a given point, we need to estimate its Gaussian curvature. Let the *edge vector* corresponding to a neighboring point be the vector pointing from the given point to that neighbor. The estimation process of the curvature for a given surface point can then be described with the following steps:

1. Estimate a rough surface normal at the given point.
2. Sort the edge vectors of its neighboring surface points in clockwise or counterclockwise order relative to this normal.
3. Compute the angles between each pair of adjacent edge vectors.
4. Compute the curvature based on these angles.

Estimating a surface normal is done by summing up all the edge vectors of the given point, including the ones pointing to its internal neighbors, and then taking the unit vector pointing in the exact opposite direction. This is an inaccurate way of defining the normal, and will lead to visibly crooked vectors that stick out of the surface in a quite slanted manner, as can be seen in Figure 4.7. However, this is not a problem for the following steps, as the normal will just be used for ordering the neighboring surface points, a process that is quite robust to large deviations of the normal in question.

Ordering the edge vectors around the normal (see Figure 4.6) is done by first setting up an arbitrary orthonormal basis with the normal acting as its third basis vector and constructing a matrix from these three basis vectors. Multiplying the transpose of this matrix with each of the edge vectors gives a representation of these edge vectors relative to the normal. In other words, the vectors are rotated such that the normal points up. The z-coordinate of the rotated edge vectors is then ignored, and their angles on the xy-plane are computed with $\mathrm{atan2}(y, x)$. The edge vectors are then sorted increasingly on this angle.

Next, to compute the internal angles between neighboring edge vectors, the standard cosine-formula is used, namely

$$\theta_{ij} = \arccos \frac{e_i \cdot e_j}{|e_i| \cdot |e_j|},$$

and finally, the computed angles are used to compute the Gaussian curvature, that being

$$\text{Curvature} = 2\pi - \sum \theta_{ij}.$$

If this value is higher than our selected value $\frac{1}{3}\pi$, the point is promoted to a corner point, else it is left untouched.

The estimation of the curvature can be executed without any explicit triangulation, and only relies on knowledge about the positions of vertices, their neighbors, and whether they lie on the surface or not, each of which is known information. Furthermore, since the computation only needs to be done for surface points, it barely contributes to the runtime of Gravo for quasi-uniform meshes. Furthermore, it can be trivially parallelized for each point individually, as further discussed in Section 4.3, making the computation almost negligible in practice.

While the estimation of the curvature corresponds almost perfectly to the true discrete curvature for most of the points, there are certain cases in which the estimation fails. Causes of this can be an erroneous normal estimation due to unevenly-distributed edge vectors, or the occurrence of two boundary points being connected by an edge that passes through the mesh instead of remaining on its surface. In these cases, either more internal angles end up being added, or the incorrect ordering will cause internal angles to overlap. In either case, the sum of internal angles increases, and thus the curvature drops far below the intended value, as can be seen in Figure 4.8. This happens occasionally, but infrequently enough not to be considered a major issue. Most of the curvature estimations are still accurate, so if the curvature is only used for heuristic purposes, the outliers should not have much impact on the result. Furthermore, the negative bias of the estimation will not cause points to be promoted even though they should not be, since promotion only happens when the curvature is positive. False positive cases are thus not expected to occur in practice.

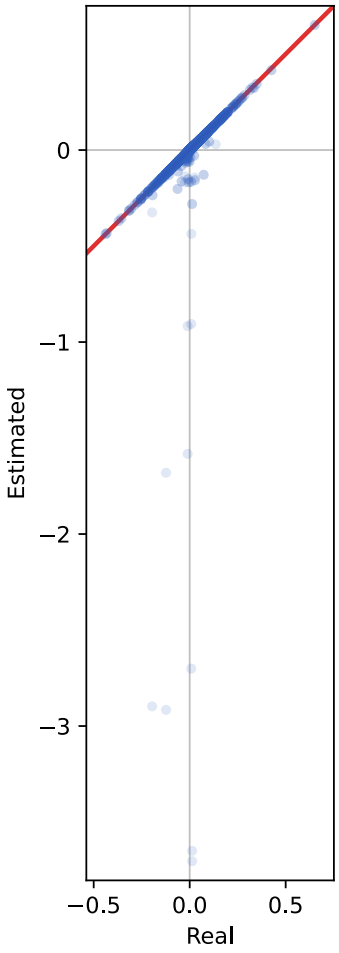


Figure 4.8: The estimated curvature of each of the 2930 surface points in `spot` plotted over the actual discrete Gaussian curvature. 2741 of the data points lie on the red line, indicating a perfect estimation.

4.2.3. Techniques

In this section, four different techniques to better preserve the shape of the mesh throughout the coarsening process are introduced and explained. Each of these is implemented in the Gravo MG algorithm and makes use of the computed estimation of the boundary.

4.2.3.1. Sparse Sampling

In Section 3.6.1.1, a distinction was made between MδIS-1 and MδIS-2, the latter being implemented in the original Gravo MG algorithm. Both variations come with their own advantages and disadvantages. For instance, it can be seen in Table 4.1 that, in comparison to MδIS-1, MδIS-2 does make the domains and the systems smaller, which means that the transfer of vectors and execution of Gauss-Seidel steps is generally faster, but it also hinders convergence, likely due to the strong discrepancy in vertex density between any two adjacent layers in the hierarchy. The choice between MδIS-1 and MδIS-2 is therefore a bit of a tradeoff, where in practice both of these techniques end up taking approximately the same time to get to the same residual error when applied in a tetrahedral context.

Algorithm	$\lvert V_k \rvert$ per layer	$\text{nonzero}(P^k_{k-1})$ per layer	$\text{nonzero}(A_k)$ per layer	Cycles
MδIS-1	13376 2280 408	44276 7195	191926 98514 23196	**7**
MδIS-2	13376 1070 165	47760 3344	191926 43006 7229	10
MδIS-1½	13376 1291 222	47224 3956	191926 50443 9996	**7**

Table 4.1: A comparison between the sampling techniques MδIS-1, MδIS-1½ and MδIS-2, used to create a multigrid hierarchy with a ratio of $\phi = 8$ that solves a Poisson problem on a uniform `spot`-tetrahedralization up to a residual error of 10^{-10}. The number of vertices and the nonzero matrix entries influence the time needed to execute one cycle.

It is important that the boundary is represented well throughout the hierarchy, something that the aggressive MδIS-2-sampling tends to hinder. The points that lie on the boundary are, however, only a small fraction of the points throughout the mesh, particularly for isotropically tetrahedralized meshes. If the point density here is higher than in the interior, it would only have a small effect on the execution time of a cycle. For this reason, a variation of the MδIS-sampling techniques has been experimented with that finds itself in between the two previously-discussed versions. We name this technique *MδIS-1½*.

MδIS-1½ has a simple definition: it applies MδIS-1 on the boundary, and it applies MδIS-2 on the interior. This means that, when selecting a point inside the mesh, the algorithm will eliminate all nearby points in the two-ring neighborhood of the point, but when selecting a point on the boundary, only the one-ring is checked, allowing other points to still be sampled even if their geometric proximity to the previously-sampled points is small. The expected result is that coarsened meshes will have an interior that is coarser than their surface, meaning that detail is maintained mainly where it would help improve convergence.

A brief look at the comparison of MδIS-1½ demonstrated in Table 4.1 shows that the sampling method does lead to a smaller system, without strongly affecting the convergence. A more thorough evaluation is demonstrated in Section 5.2.1.2.

4.2.3.2. Sampling Order

In Section 3.6.1.1, Gravo MG's subsampling technique, dubbed MδIS-2 in this work, was described as sampling points and then removing close points in the neighborhood. What was not elaborated upon was what point should be sampled out of the point set that is still available. While this matter is not really discussed in [Wie+23], it is a decision that can have significant effects on the generated hierarchy and thus the convergence of the solver. If a random order is chosen, it could happen that points around defining features of the mesh, such as corners or ridges, are sampled before the corner or ridge point itself can be sampled. These important points are then eliminated, leading to a degradation in the shape.

Instead, a sampling order can be chosen to intentionally prioritize points of a higher boundary rank over other points. By sorting the list of vertices on decreasing rank, breaking ties arbitrarily, it is guaranteed that for every point, it is either sampled or it is in the vicinity of a sampled point with a rank at least as high. In practice, this means that most corner points are guaranteed to be sampled, and many ridge points and surface points will be too. Since these are the points are crucial in giving shape to the domain, the shape is more likely to be preserved after the coarsening is done.

4.2.3.3. Pit Prevention

Even if the boundary-prioritizing sampling order from Section 4.2.3.2 is used, it is still possible for "pits" to occur in the sur-

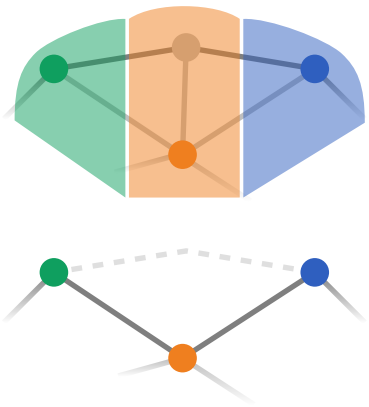


Figure 4.9: A schematic representation of how a pit naturally spawns. The fine points and clusters are shown above, the coarse points and the resulting boundary below.

Figure 4.10: The original `spot` mesh (left) and a quickly coarsened version of it (right). Upon coarsening the mesh, the outer surface (yellow) gets ripped open, leaving pits everywhere (purple).

Figure 4.11: A quickly coarsened version of the `spot` mesh, but with pit prevention enabled.

face of the mesh, as seen in Figure 4.10. These pits are not characterized by a lack of vertices, but by a lack of edges, triangles, and tetrahedra covering up the inside of the domain. This can affect the prolongation operators around the boundary negatively, precisely where it matters the most. It is therefore important that pits are first understood and then prevented.

Pits mainly occur in a very specific situation, visualized in Figure 4.9. Suppose there are multiple boundary points connected to each other, of which one does not get sampled. This surface point would then be assigned to the cluster of the closest point in its neighborhood. The closest point may actually be an *interior* point, instead of another surface point. If the not-sampled surface point is assigned to the cluster of the sampled interior point, and the surface points around it do not already neighbor each other, then the surface point gets "absorbed" into the interior point, causing a cavity, and the surface neighbors' clusters will not touch each other to patch that hole up. The result is a pit, and thus a drastic change in the shape of the mesh, which hinders accurate transfer between the layers in the multigrid hierarchy.

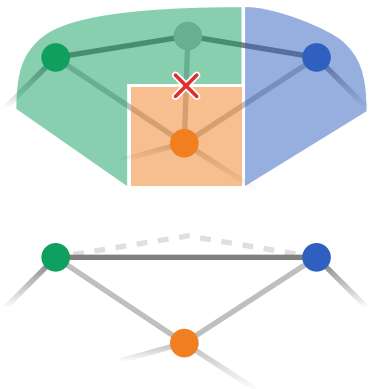


Figure 4.12: By explicitly forbidding a boundary point to join an interior cluster, the boundary clusters remain adjacent.

One way to prevent pits from appearing is to not allow fine points to be assigned to clusters represented by points of a lower rank. A surface point could then, for example, never be absorbed by an interior point. Enforcing this is a matter of adding a simple `if`-statement in the clustering stage. If an edge exists that connects a surface point to an interior point, this edge cannot be traversed in this direction, sort of as if the edge had a weight of ∞. This will ensure that points that surface points are guaranteed to join clusters of other surface points, as shown in Figure 4.12. The surface will consequently be completely connected, with no gaps revealing the interior of the mesh, as demonstrated in Figure 4.11.

4.2.3.4. Boundary-Aware Smoothing

In [Wie+23], the authors noted that slightly displacing the subsampled coarse points from their original position would lead to a small increase in convergence. The justification behind this is that this perturbation reduces the number of single-entry rows in the prolongation matrix, since the sampled fine points that would usually restrict only to their coarse counterpart will now restrict to other vertices as well.

The perturbation in [Wie+23] was implemented by moving the sampled coarse points towards the av-

erage position of the fine points in its cluster. This would cause the coarse points to still be spread out uniformly, but prevent the coarse points to lie exactly on top of their fine equivalent. While in principle this approach can directly be applied to the tetrahedral case as well, it causes the boundary to shrink and deform. This happens because clusters located on the boundary tend to have averages that lie a bit buried underneath the surface of the mesh, due to the inclusion of interior points within the cluster. The change in shape caused by the surface shrinking inwards can be detrimental for the performance of a multigrid solver, so it might not seem appealing to perturb the points at all. However, there exists a way to perturb the points without causing the boundary to shrink, and thus to still have the benefit of smoothing the vertex positions.

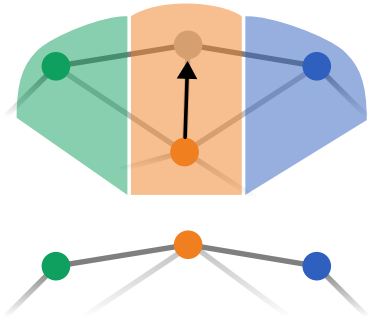


Figure 4.13: Boundary-aware smoothing helps to fill pits appearing in the surface, by dragging interior points outwards.

Figure 4.14: An example of pits on the `spot` mesh being pulled towards the surface, turning a mesh with pits (left) into one with a smooth surface (right), as if inflating it

For this reason, we apply a form of **boundary-aware smoothing**. The idea is to not necessarily take into account *all* points within the cluster when computing the average position, but rather only the fine points whose boundary rank matches that of the new coarse point. Coarse surface points then get moved to the average position of the fine surface points of the cluster, coarse ridge points to that of the fine ridge points, et cetera. The effect is that boundary points would no longer get shrunk inwards, as they will only be shifted into directions roughly tangential to the surface. The same thing can be said for ridge points, which will no longer be pulled inwards due to the local curvature of the surface around the point, and for corner points, which will now likely not be perturbed at all, unless there happen to be many corner points located close to each other. With boundary-aware smoothing, perturbation can be applied safely, preserving the general shape and defining features of the mesh.

A lucky consequence of boundary-aware smoothing is that it also immediately addresses the problem of pits appearing in the surface, as discussed in Section 4.2.3.3. The cause of these pits was clusters containing fine boundary points being represented by coarse interior points. By applying a round of boundary-aware smoothing, these coarse points will be moved towards an average position that is located right on the surface (see Figure 4.13). As a result, any pit that might have appeared is fixed, restoring the shape of the surface, as seen in Figure 4.14.

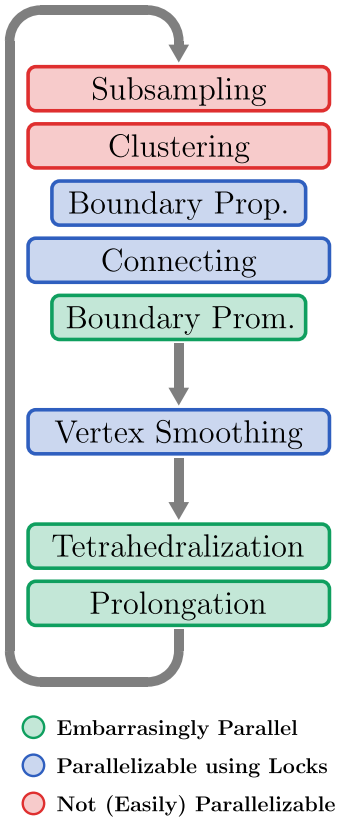


Figure 4.15: An overview of the iterated stages of Gravo MG and their parallelizability.

4.3. Parallelization

When taking a look at the entire algorithm, it can be noted that quite a number of individual stages can be parallelized. While we do not

apply this parallelization ourselves in the experiments that follow, it could help in accelerating the construction process on hardware that allows it, and thus merits a discussion. An overview of the algorithm and its parallelizability is given in Figure 4.15.

The estimation of the curvature on coarser levels to optionally promote points is embarrassingly parallel, as the only writing operation that is executed is the promotion of the point itself. The tetrahedralization and prolongation can also be parallelized by creating a list for each individual point for the threads to store their results in. That way, no two threads ever have to touch the same list, and the results, be that a list of triangles and tetrahedra or a list of prolongation weights, can be quickly collected into one list afterwards.

When it comes to propagating the boundary, connecting the neighboring clusters, or applying vertex smoothing, parallelization can still be employed, but will require some sort of locking system. The reason for this is that these stages still iterate over all of the fine points on one layer, but each also performs write operations on the corresponding coarse points on the next layer. By initiating a lock for each of the coarse points, these operations can be prevented from interfering with each other. The use of critical sections had been implemented first, but was replaced with the locking system; many of the operations do not interfere with each other at all, assuming that the number of coarse points is decently large, so it makes more sense to prevent the few collisions that might happen with individual locks, instead of halting every other thread simultaneously when one thread wants to add an element to a list no other thread was touching anyway.

Lastly, there are a few stages that cannot easily be parallelized. The clustering of the vertices cannot be done in parallel without some sophisticated approaches, as it relies on a run of Dijkstra's algorithm, which requires nodes to be visited in a certain order. Subsampling, if done with MδIS-1½, is also difficult to do concurrently, as it would allow for two neighboring points to be sampled simultaneously, which could affect the quality of the hierarchy and/or hinder convergence. Attempts to use locks to prevent this could result in deadlocks, which should be prevented at all costs.

5

Experiments & Discussion

To evaluate the performance of the extension of the Gravo MG algorithm, a series of experiments was conducted, comparing a handful of benchmarks under a multitude of circumstances and contexts. Section 5.1 discussed the general setup of the conducted experiments, mainly focusing on the definition of the problems. Then, in Section 5.2, several experimental studies are conducted to compare Gravo to variations of itself and to some state-of-the-art algorithms. In the end, Section 5.3 summarizes the results by presenting a brief discussion of Gravo's performance.

5.1. Setup

In this section, the setup of the experiments is described in detail. Section 5.1.1 describes how the tetrahedral meshes used in the experiments were obtained, and Section 5.1.2 discusses what linear problems were solved on these meshes to assess Gravo.

5.1.1. Meshes

In order to obtain tetrahedral meshes suitable for the experiments, surface meshes were taken and then tetrahedralized with the surface acting as a boundary. The surface meshes in question were obtained from the *Thingi10K* dataset [ZJ16]. A variety of meshes has been selected in order to ensure a balanced representation over various quantitative and qualitative attributes, e.g. the number of vertices or the complexity of the shape. All chosen meshes are watertight and both vertex- and edge-manifold.

We distinguish between two different types of tetrahedral meshes in our experiments: *adaptive meshes* and *isotropic meshes* (see Figure 5.2).

Adaptive meshes are meshes for which the density of points varies over different parts of the domain. Adaptivity can be useful in practice to, for example, provide more detail in interesting regions of the domain than in other regions. For volumetric meshes, this difference in detail is mostly visible in the density of the surface in comparison to the interior. Simulations on bodies tend to show the most interesting phenomena on the boundary of the domain, so it would make sense to add many vertices in proximity to the boundary, and leave the interior a bit sparser.

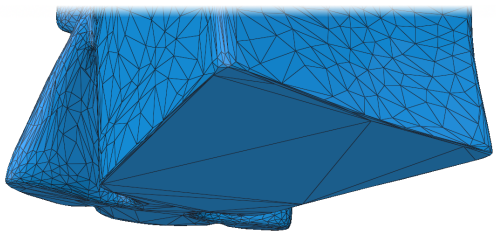


Figure 5.1: A bad triangulation of a mesh. Some triangles are too large, causing huge portions of the surface to contain no degrees of freedom. Other triangles are too thin, causing low-quality tetrahedra to be generated, which leads to unrealistic dependencies between points.

In our experiments, we generate adaptive meshes using *TetGen*, using the following simple command:

```
tetgen -pq1.6 file.ply
```

The `p`-flag indicates that we are coarsening a *piecewise linear complex* obtained from a `.ply`-file, and the `q1.6`-flag is included to allow TetGen to refine the

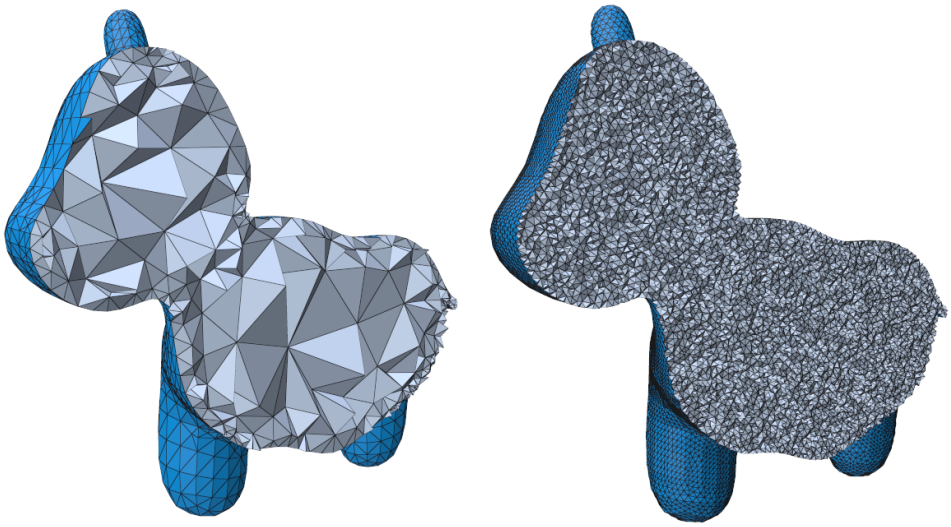


Figure 5.2: An adaptive tetrahedralization (left) and an isotropic tetrahedralization (right) of `spot` [CPS13].

mesh, and to enforce a certain quality measure for each of the generated tetrahedra, in this case by constraining their *radius-edge ratio*. This is defined as the ratio of the radius of the tetrahedron's circumsphere, i.e. the sphere that crosses through all its vertices, and the minimum edge length in the tetrahedron. For well-shaped tetrahedra, this ratio should be small. The above flag ensures that the generated tetrahedra have a radius-edge ratio of *at most* 1.6. To ensure that this is possible, *TetGen* might occasionally need to add *Steiner points* to the surface, which are points specifically added to allow certain tetrahedra to exist that otherwise could not. This could be useful for difficult corners, but also for large surface triangles, for example, those seen in Figure 5.1.

Occasionally, even adding Steiner points still leads to a mesh with a bad tetrahedralization, on which any problem is doomed to suffer from slow convergence. For this reason, some of the meshes from *Thingi10K* are remeshed before passing them to *TetGen*, to obtain a better-triangulated boundary and thus a better mesh. This remeshing is done using *CGAL*'s `isotropic_remeshing` feature [Coe+24], which aims to capture the same surface as well as possible while also keeping the edge lengths close to a predefined value. For adaptive meshes, an adaptive sizing field was used based on the curvature of the local region around the points, to still simulate varying vertex densities on the surface. The `Y`-flag is added to the *TetGen*-call that follows, which ensures the surface remains will not have any vertices added or removed.

Isotropic meshes are characterized by having approximately the same edge length throughout the entire domain. This holds not only for the surface but also for the interior. Unlike the adaptive meshes, whose vertex density tends to vary over the domain, the distance between any neighboring pair of points in an isotropic mesh is as close to a certain constant as possible. The distribution of the vertices is thus almost uniform all the way through. Since the surface to pass to *TetGen* is likely not already isotropic, we again make use of *CGAL*'s `isotropic_remeshing` feature, this time with a constant sizing field. The new edge length is arbitrarily chosen and tweaked to control the number of points. Then, the new isotropic surface is passed into *TetGen* with the following command:

```
tetgen -pmq1.6 file.ply
```

The `m`-flag tells *TetGen* to look for a `.mtr`-file, which has been generated to provide the algorithm with a target edge length, equal to the same arbitrarily chosen value. The output is an isotropic tetrahedral mesh, ready for use in experiments.

In Appendix A, an overview is given of all the meshes that have been used in the experiments, along with some general statistics about the meshes.

5.1.2. Problems

Different types of linear systems $A\mathbf{x} = \mathbf{b}$ show different behavior when being optimized by the multigrid solver. A selection of problems to solve has been made to demonstrate convergence in multiple contexts. The problem definitions rely on two types of matrices that encode two important properties of the mesh, those being its *stiffness* and its *mass*. The definitions of these matrices are discussed in Section 5.1.2.1. Following that, our method for fixing variables to values in a linear system is briefly explained in Section 5.1.2.2. Finally, the linear systems for the selected problems are introduced in Section 5.1.2.3.

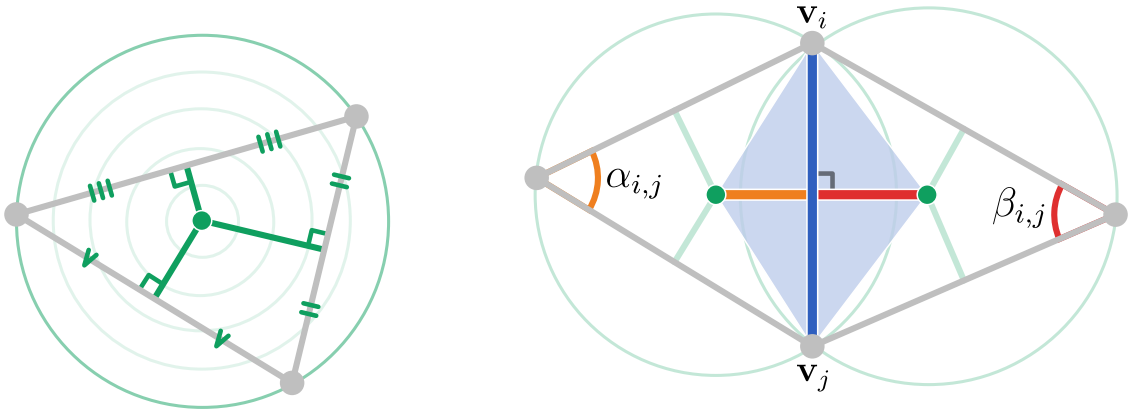
5.1.2.1. Matrix Definitions

The first matrix to define is the **stiffness matrix** S, which is a matrix describing the connections between different vertices of the mesh and the strength thereof. It is constructed by assigning each edge $(i, j) \in E$ in the mesh a scalar weight $w_{i,j}$, and then adding this weight to the diagonal entries $a_{i,i}$ and $a_{j,j}$, while subtracting it from the off-diagonal entries $a_{i,j}$ and $a_{j,i}$. Generally, the higher the weight $w_{i,j}$, the more important the edge (i, j) is compared to its neighbors.

Since the weight of an edge can be defined in multiple ways, different definitions exist for the stiffness matrix. In our experiments, we opted for the **cotangent Laplacian**, which is defined by the equation

$$S_{i,j} = \begin{cases} -\frac{1}{2}(\cot(\alpha_{i,j}) + \cot(\beta_{i,j})) & \text{if } (i,j) \in E, \\ \sum_{k \neq i} -S_{i,k} & \text{if } i = j, \\ 0 & \text{otherwise,} \end{cases} \tag{5.1}$$

where $\alpha_{i,j}$ and $\beta_{i,j}$ are the two angles opposing the edge (i, j). A geometric understanding of this formula is displayed in Figure 5.3. The main idea is to subdivide the mesh surface into quadrilateral shapes, such that each quadrilateral corresponds to one edge in the mesh. In the definition for the cotangent Laplacian, these quadrilaterals span the two vertices connected by the edge, and the two circumcenters of the adjacent faces. The actual matrix entry $S_{i,j}$ then corresponds to the *width* of the quadrilateral divided by its *height*, i.e. the distance between the circumcenters divided by the length of the edge. By applying some trigonometry, one can determine that the widths of the two halves of the quadrilateral are given by $\frac{1}{2}\cot(\alpha_{i,j})\,\|v_i - v_j\|$ and $\frac{1}{2}\cot(\beta_{i,j})\,\|v_i - v_j\|$, which, when added together and divided by the length of edge (i, j), lead to the expression listed in Equation 5.1.



(a) The *circumcenter* of a triangle is defined as the center of the circle spanning all its vertices.

(b) An edge between two triangles (blue), and its quadrilateral spanning four points (light blue). The matrix entry $S_{i,j}$ is the quadrilateral's width divided by its height, and can be computed using the cotangent of the opposing angles.

Figure 5.3: A geometric interpretation of the cotangent Laplacian.

This definition of S has the benefit of S being symmetric as well as positive semi-definite, regardless of the topology or the geometry of the mesh. There is, however, no guarantee that the edge weights will all be positive, as the cotangent evaluates to negative values when provided obtuse angles, i.e. angles larger than $90°$. Visually, this can be interpreted as the circumcenter lying *outside* of its triangle. While not ideal, the matrix S remains usable in the majority of contexts.

While the formula and geometric interpretation base themselves solely on *triangles*, instead of the tetrahedra which hold our interest, the same geometric principle can be applied to define a cotangent Laplacian for *tetrahedral* meshes as well [Cra19]. In our experiments, we rely on the tetrahedral `cotmatrix`-implementation from `libigl` [JP+18].

The second matrix to define is the **mass matrix** M, which describes how the mass of the mesh is distributed over the domain. A *lumped mass matrix* was chosen for these experiments. This is a *diagonal* matrix where each entry on the diagonal corresponds to the mass assigned to one of the vertices. For triangular meshes, this assignment can be done by calculating the areas of each of the triangles, splitting it into three parts, and dividing those areas among the three vertices spanning the triangle. How this split is done is a matter of choice; in these experiments, to maintain consistency with the definition of the stiffness matrix, the three-way split shown in Figure 5.3a is used, which involves three perpendicular bisectors meeting at the circumcenter of the triangle, cutting the triangle into parts. Every vertex v_i receives a part of the area from each of its adjacent faces (see Figure 5.4), the sum of which totals to the matrix entry $M_{i,i}$. Since this construction is actually equivalent to finding the *Voronoi cell* corresponding to the vertex, this matrix is often referred to as the **Voronoi mass matrix**. Due to the nature of Voronoi cells, all vertex masses in a manifold mesh are guaranteed to be positive.

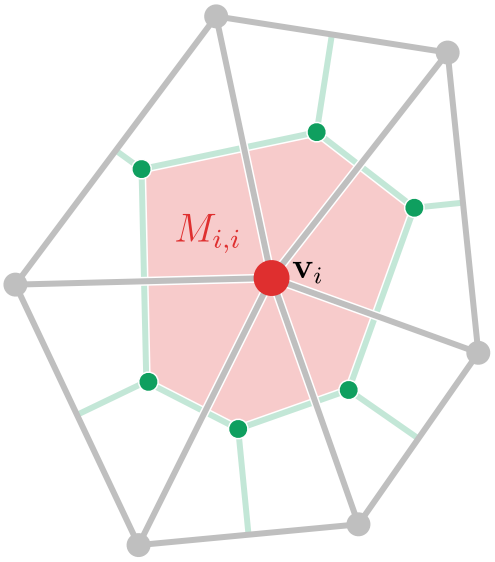


Figure 5.4: The mass of a vertex is determined by the area of the Voronoi cell surrounding it.

Again, this concept can be translated from a triangular context to a tetrahedral one without much trouble, now just considering the volumes of the Voronoi cells surrounding the vertices. An implementation from `libigl` can be employed to compute the Voronoi mass matrix for any tetrahedral mesh. This matrix is used in the experiments that follow.

5.1.2.2. Fixing Values

Some of the problems listed below rely on a notion of *fixed values*, i.e. vertices whose value is not a variable to solve for, but a constant shaping the problem itself. This is needed to support, among other things, Dirichlet boundaries. Given a system $A\mathbf{x} = \mathbf{b}$, we fix a variable x_i to the value v_i by replacing the i^{th} row with the equation $1 \cdot x_i = v_i$, and subtracting multiples of this equation from the others, to eliminate the x_i-term everywhere else. This second action is an elementary row operation, seen before in Section 3.3.1, which means it does not affect the solution space of the system.

$$\begin{bmatrix} \ddots & \vdots & \vdots & \vdots & \iddots \\ \cdots & a_{i-1,i-1} & a_{i-1,i} & a_{i-1,i+1} & \cdots \\ \cdots & a_{i,i-1} & a_{i,i} & a_{i,i+1} & \cdots \\ \cdots & a_{i+1,i-1} & a_{i+1,i} & a_{i+1,i+1} & \cdots \\ \iddots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} \vdots \\ x_{i-1} \\ x_i \\ x_{i+1} \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ b_{i-1} \\ b_i \\ b_{i+1} \\ \vdots \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} \ddots & \vdots & \vdots & \vdots & \iddots \\ \cdots & a_{i-1,i-1} & 0 & a_{i-1,i+1} & \cdots \\ \cdots & 0 & 1 & 0 & \cdots \\ \cdots & a_{i+1,i-1} & 0 & a_{i+1,i+1} & \cdots \\ \iddots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} \vdots \\ x_{i-1} \\ x_i \\ x_{i+1} \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ b_{i-1} - a_{i-1,i} \cdot v_i \\ v_i \\ b_{i+1} - a_{i+1,i} \cdot v_i \\ \vdots \end{bmatrix}$$

Figure 5.5: Fixing $x_i = v_i$ in a linear system.

It should be noted that at this point, the i^{th} row and column could in theory be removed from the system altogether, because they do not in-

teract with the rest of the entries anymore. Even though this would make the system smaller, the decision was made not to implement this for three reasons. First, the removal of the entries would be a bit more involved than just deleting the entries from A and $\mathbf{b}$, since the prolongation matrices are built without any knowledge of the problem, including which of the values are fixed; changing the system would require changing the prolongation matrices as well, which is a computationally tedious task. Second, the difference in performance would be quite small, since the matrices used in the experiments are already sparse, and these redundant rows only add one entry each to the quite large pile of numbers that are *not* redundant or removable. Third, while this feature could definitely be considered an optimization, it would be an optimization of the *problem*, and not of the *hierarchy construction*, which is the algorithm being benchmarked. There does not appear to be any reason to believe a more optimized problem will result in a better analysis and comparison between different solvers, so none of the experiment results are expected to be any different.

5.1.2.3. Linear Systems

In the conducted experiments, four types of linear problems were solved on a variety of domains under a variety of conditions. These four are *Poisson's equation*, the *biharmonic equation*, *smoothing*, and *gradient deformation*, each of which is to be discussed next.

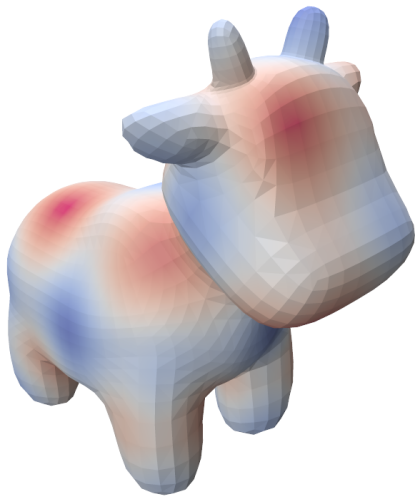


Figure 5.6: The applied Dirichlet boundary conditions for Poisson's and the biharmonic equation.

First, **Poisson's equation** is an elliptic PDE given by the equation

$$\Delta\varphi = f,$$

where φ is a function of the type $\Omega \to \mathbb{R}$. On the boundary $\partial\Omega$, the values of φ are fixed to predefined values. It can be discretized by substituting the function φ for the vector $\mathbf{x}$, the function f for a vector $\mathbf{f}$, and Δ for its discrete version, the **mesh Laplacian matrix** L, defined as $M^{-1}S$. This would yield the matrix equation $M^{-1}S\mathbf{x} = \mathbf{f}$, but since the mesh Laplacian matrix is not symmetric, one of the conditions for the Gauss-Seidel method to work, it is more convenient to instead define the equation as

$$S\mathbf{x} = M\mathbf{f},$$

which is functionally the same, but suitable for iterative relaxation, due to the symmetric nature of S. We fix the boundary points to particular values, which are determined by entering their positional coordinates into the arbitrarily-chosen formula

$$g(x, y, z) = \sin(10x) + \sin(10y) + \sin(10z),$$

giving a smooth fluctuation in boundary values as seen in Figure 5.6. Furthermore, we enforce $\mathbf{f} = 0$ for all other points, thus turning the equation into a Dirichlet problem, also referred to as a *harmonic equation*, in our experiments. Since the equation in question focuses on the interior of the mesh and not on the boundary, this problem is only run on isotropically tetrahedralized meshes, not on adaptive meshes, which have very few points on the interior to begin with and would thus not be as interesting for this problem.

Second, the **biharmonic equation** is a fourth-order PDE given by

$$\Delta^2\varphi = 0.$$

Similarly to Poisson's equation, using the mesh Laplacian, the equation can be discretized to

$$(M^{-1}S)(M^{-1}S) = \mathbf{0},$$

and can be converted into the suitable and symmetric equation

$$SM^{-1}S\mathbf{x} = \mathbf{0}.$$

The biharmonic equation is particularly interesting to experiment with, due to its complexity in comparison with the other problems, better demonstrating the difference between the convergence rates of

different solvers. In the experiments, the same Dirichlet boundary conditions as before are applied to the boundary points. Similar to the Poisson problem, we only run this problem on isotropic meshes, not on adaptive ones.

Third, **smoothing** is the process of reducing outliers by averaging out vertex locations in a local neighborhood. The linear system that will be used to apply this smoothing is given as

$$(M + \alpha S)X'_V = MX_V,$$

where X_V is a $|V| \times 3$-sized matrix in which each row represents the position of one of the $|V|$ vertices, X'_V represents the new respective positions to be solved for, and α is an arbitrary scalar that we fix to 10^{-3}. Due to the inclusion of S into the equation, every vertex is pulled to the average position of its neighbors, simulating the smoothing effect. No boundary conditions are applied to this system, causing different convergence behavior than Poisson's equation and the biharmonic equation. The problem also distinguishes itself by having lower residual errors than other problems and, therefore, generally requiring fewer cycles to converge; if the original shape of the mesh is passed as an initial guess to the solution, most of the points will already be close to their position in the finally obtained solution.

Lastly, **gradient deformation** is a type of deformation that can be applied to a mesh, in which every tetrahedron t_i is transformed by a certain matrix 3×3-matrix D_i, up to translation. The linear system is given by

$$(G^\intercal M_T G)X'_V = G^\intercal M_T (DG)X_V,$$

where G is the *gradient matrix*, M_T is a diagonal mass matrix that describes the mass of each tetrahedron, instead of each vertex as in Section 5.1.2.1, and D is a collection of matrices D_i to transform each tetrahedron with, constructed by interleaving these matrices row by row. G is obtained using `libigl`'s `gradmatrix` function, which has support for tetrahedral meshes. The matrix $G^\intercal$ is included to obtain a least-squares approximation of the desired deformation, since it is not guaranteed that every transformation can be perfectly abided by simultaneously, and M_T is included to penalize errors in larger tetrahedra more severely than in smaller tetrahedra. For our experiments, we define the transformation matrix for each tetrahedron by computing its barycentric center (x_i, y_i, z_i), and then computing the matrix as

$$D_i = \begin{bmatrix} \cos(\frac{1}{2}\pi y) & \sin(\frac{1}{2}\pi y)(1 + \frac{1}{4}x) & 0 \\ -\sin(\frac{1}{2}\pi y) & \cos(\frac{1}{2}\pi y)(1 + \frac{1}{4}x) & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

which resembles a transformation as if the mesh is being bent or curved backwards at the top and bottom (see Figure 5.7). Since this system only reasons about vertex positions relative to each other, the system is indifferent to translations of the solution. As a result, the matrix is non-invertible, and more than one solution exists. To combat this, the system is slightly altered by adding a regularization term to the left-hand side of the equation, leading to the new well-defined equation

$$(G^\intercal M_T G + \alpha M)X'_V = G^\intercal M_T (DG)X_V.$$

Again in this equation, α is an arbitrary scalar that we fix to 10^{-3} in our experiments.

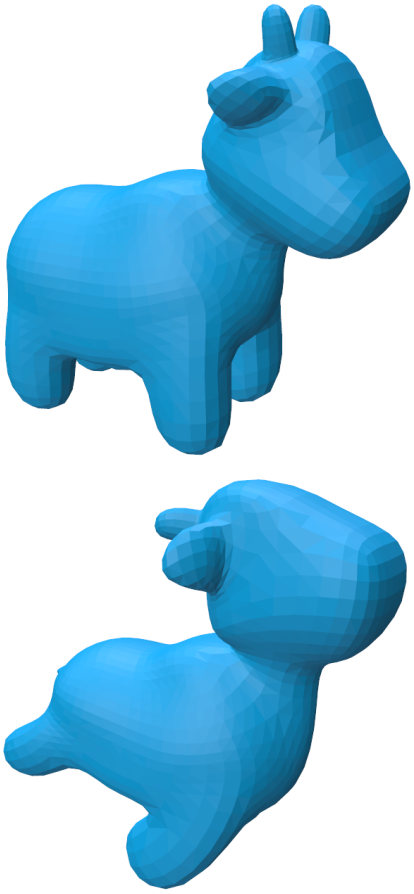


Figure 5.7: Whoosh! (The transformation applied to our gradient deformation problem.)

5.2. Experiments & Results

In this section, we outline the experiments that were executed to evaluate Gravo MG. The meshes, matrices, and linear systems that were discussed before are used here to judge how well Gravo MG can actually solve practical optimization problems. The evaluation is split into three parts. In Section 5.2.1, an ablation study is performed to try to assess the influence of each of the individual design decisions on the optimization process. Then, Section 5.2.2 compares Gravo MG to some state-of-the-art alternatives for solving systems under similar constraints. Finally, 5.2.3 evaluates Gravo MG in a more regular setting, comparing it to prolongation done through refinement instead of coarsening.

5.2.1. Ablation Study

To determine what parts of the algorithm help in increasing performance, we perform an ablation study on various features of the algorithm, including the different boundary preservation techniques introduced and discussed in Section 4.2.3.

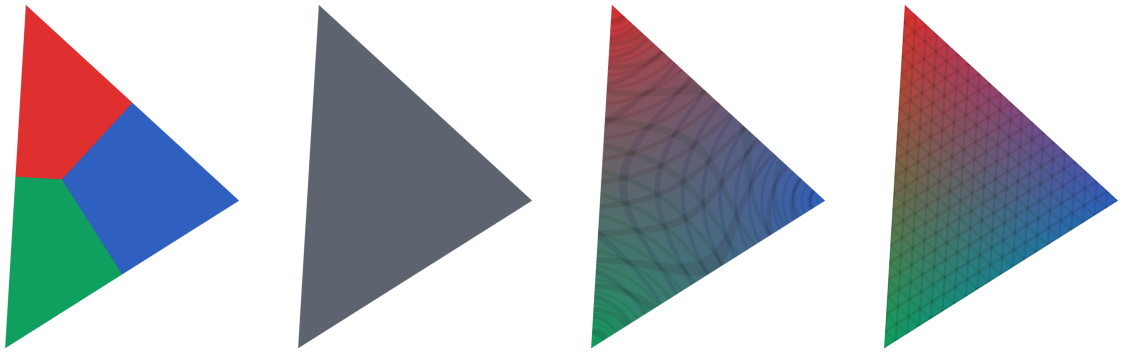
For all of the discussed experiments, the following holds:

- V-cycles are used, with two iterations of Gauss-Seidel smoothing both before and after the recursive call;
- The multigrid method coarsens the domain until a domain is added with at most 1000 points.
- Instead of painting the boundary manually, the initial ranks of the boundary points are computed automatically using dihedral angles and discrete Gaussian curvature (see Section 4.2.2).

5.2.1.1. Prolongation Weights

The reason Gravo MG constructs a triangulation and a tetrahedralization is to use barycentric coordinates. Without describing volumes and surfaces on top of the point cloud, there would be no sensible definition of how to compute barycentric coordinates. To justify the effort done by Gravo MG, we compare the use of barycentric weights to other common definitions that could be used instead. The list of weighting schemes we compared is as follows (see also Figure 5.8):

- `BINARY`: Every fine point prolongates to only one coarse point with a weight of 1, as is done in [XTL19];
- `UNIFORM`: Every fine point prolongates to some coarse points chosen as is done in Section 4.1.2, and gives each of the n chosen points a weight of $\frac{1}{n}$;
- `INV-DIST`: Every fine point prolongates to some coarse points chosen as is done in Section 4.1.2, and gives all of the chosen points weights such that they are proportional to the inverse of their Euclidean distances to the fine point;
- `BARYCENTRIC`: Our approach, every fine point prolongates to some coarse points chosen as is done in Section 4.1.2, and gives each point a weight determined by the fine point's barycentric coordinates within the primitive that the coarse points span.



(a) Binary weights. **(b)** Uniform weights. **(c)** Inverse distance weights. **(d)** Barycentric weights.

Figure 5.8: The different weighting schemes, visualized on a single triangle. The intensities of the red, green, and blue colors indicate the intensity of the weights towards the respective corners.

The algorithm was run on `devil` and `trex`, which are an isotropic and an adaptive mesh, respectively. None of the optimization features discussed in Section 4.2.3 was used or enabled. The results are displayed in Figure 5.9. In these graphs, the residual error is laid out over the estimated number of multiplications that one cycle takes. This estimation is based on the density of the prolongation matrices and the matrices on the left-hand side of the equations on each multigrid level. Each dot in the graph represents the start or end of a cycle.

What is clear from the results is that the use of barycentric coordinates generally speeds up the solving process by a wide margin. In most of the experiments that were run, `BARYCENTRIC` appears to take fewer cycles to get to a lower residual error than any of the other techniques. This means that, in comparison to `UNIFORM` and `INV-DIST`, this approach is expected to be superior when plotted over units of time as well, as these other two weighting schemes do not lead to sparser matrices, and thus take approximately as much time to execute one cycle. The binary weights (`BINARY`) as seen in [XTL19] do reduce the number of nonzero entries, and because of that, this approach could beat the barycentric weights by taking smaller steps in rapid succession; in Figure 5.9c, this almost happens. However, in each of the executed experiments, the barycentric weighting scheme shows a clear superiority over the alternative schemes, be it with a small or a wide margin. This demonstrates the benefit of a prolongation method not based on just a graph representation of a mesh, as is used in [Shi+06].

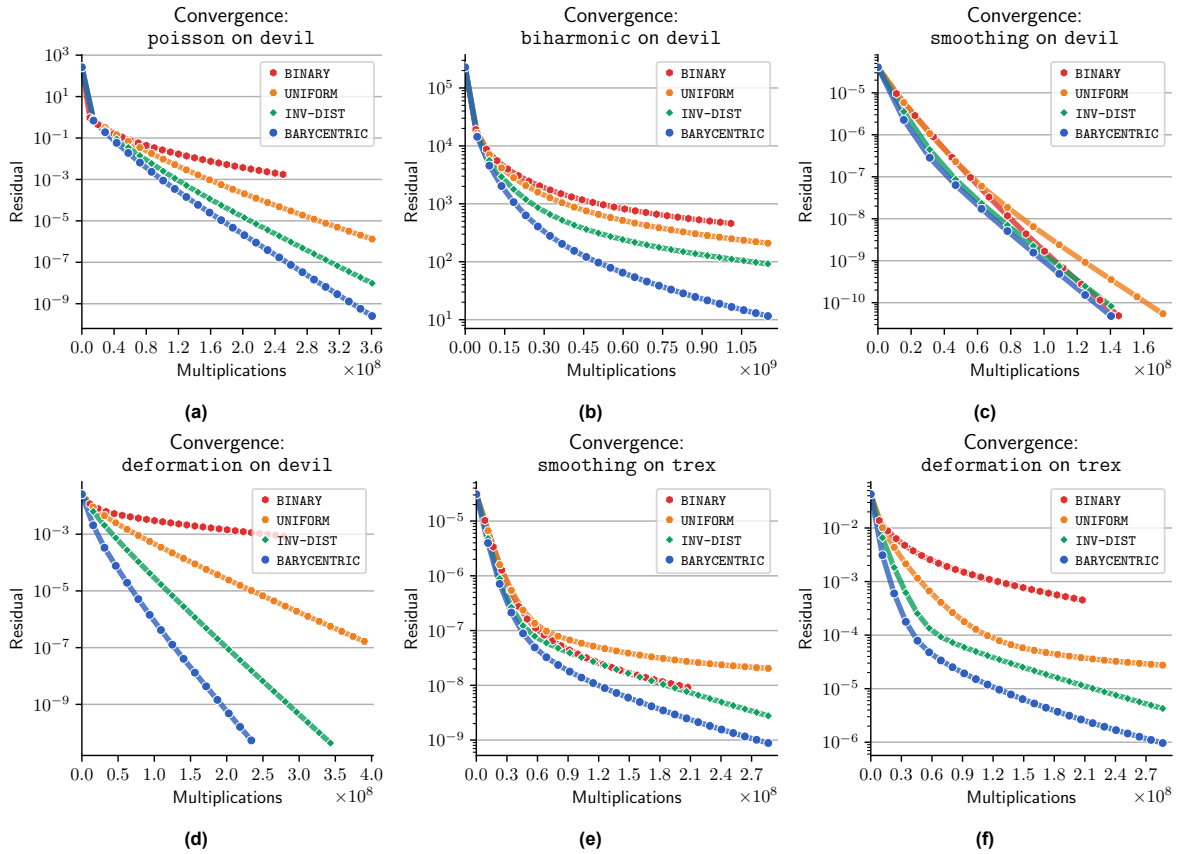


Figure 5.9: The effects of weighting schemes, compared and visualized.

5.2.1.2. Sparse Sampling

To evaluate the effectiveness of the used MδIS-1½ approach from Section 4.2.3.1, we benchmark it in comparison to MδIS-1 and MδIS-2 against a variety of meshes. The list of sampling strategies that have been compared is:

- `MDIS-1`: Eliminate vertices in the one-ring everywhere;
- `MDIS-2`: Eliminate vertices in the two-ring everywhere;
- `MDIS-1½`: Our approach, eliminate vertices in the one-ring on the surface, in the two-ring in the interior.

Again, the algorithm was run on the adaptive meshes `ball` and `trex`, and on the isotropic meshes `devil` and `die`. None of the other optimization features discussed in Section 4.2.3 was used or enabled. The results are displayed in Figure 5.10.

The hypothesis of `MDIS-1½` being more efficient softly shines through in the plots. In almost all of the executed experiments, the new approach comes out as the most or one of the most efficient options among the ones listed, albeit by a small margin. A consistent tradeoff can be seen when comparing `MDIS-1` to `MDIS-2`; `MDIS-1` reduces the residual error more with each cycle, but the cycles take longer to compute, whereas `MDIS-2` can execute more cycles within the same amount of time, while they do not reduce the error as much. The consequence is that both techniques often perform quite similarly, as their advantages and disadvantages tend to cancel each other out. Occasionally, the two even seem to almost completely overlap, e.g. in Figure 5.10h. `MDIS-1½`, on the other hand, appears to strike a bit of a balance between the two. While converging less per cycle than `MDIS-1` and taking slightly more time per cycle than `MDIS-2`, the differences are small enough such that, often, `MDIS-1½` performs at least as well or even a bit better than both of these two approaches. Clear examples of this behavior are seen in Figures 5.10b, 5.10h, and 5.10i, for example. It appears that enforcing more careful coarsening on the boundary leads to a small increase in convergence, whereas coarsening the interior rapidly does not affect the performance of the solver to a significant extent.

The results are not too remarkable, of course. For example, in the plots that show experiments on adaptive *ball* mesh (Figures 5.10c and 5.10f), there seems to be no difference between `MDIS-1` and `MDIS-1½`. This can be justified by noting that the interior of these meshes is already sparse, and thus the difference between the hierarchies created by the two sampling strategies is small or even non-existent. Furthermore, on particular problems, such as `biharmonic`, the difference is barely there; Figures 5.10j and 5.10k show little difference between any of the strategies. And lastly, even in the cases where there does appear to be a measurable distinction between the plots, the speedup obtained through `MDIS-1½` is minor at best. Nevertheless, since using this sampling approach does not add to the construction time, even this small gain makes implementing it worth it.

The weakly positive results make a small case for the benefit of MδIS-1½, but there is a more general and more interesting conclusion to be made from this. Namely, that maintaining detail on the interior of the mesh really is less important than on the boundary of the mesh. The MδIS-1½ sampling technique that was introduced, discussed, and evaluated is simply one of the many different sampling methods that are capable of making a distinction between these two parts of the domain, and more sophisticated approaches could probably be designed to optimize even further on both sparsity and solving time.

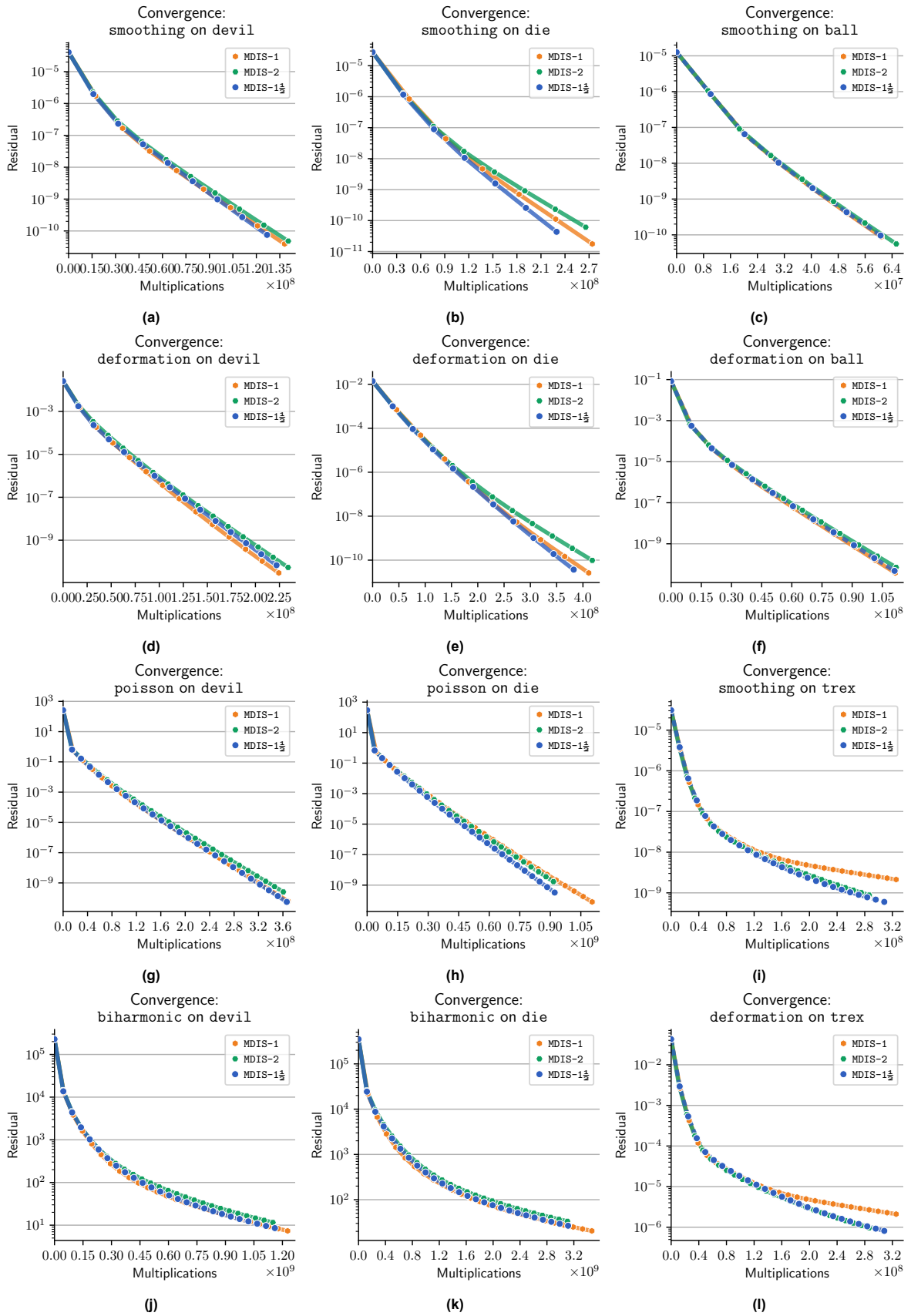


Figure 5.10: Different sampling strategies compared.

5.2.1.3. Sampling Order

The sampling order proposed in the algorithm relies on the notion of boundary ranks. This is just one of the many, many ways in which vertices can be ordered. To assess the performance of this order in particular, we compare it to a number of other possible sorting techniques. The list of chosen techniques to compare is as follows:

- `RANDOM`: Traverse the points in random order;
- `SWEEP`: Sort the points based on their position, specifically increasing based on the linear sum $x_i + y_i + z_i$;
- `DEGREE`: Sort the points increasingly based on the number of connections to neighboring points;
- `RANK`: Our approach, sort the points descendingly on their boundary rank. Break ties arbitrarily.

The algorithm was run on the adaptive meshes `arc_triomphe` and `trex`, and on the isotropic meshes `devil` and `die`. For the adaptive meshes MδIS-2 was used, and for the isotropic meshes MδIS-1½. No other feature discussed in Section 4.2.3 was used or enabled. The results are displayed in Figure 5.11.

In most of the executed experiments, `RANK` appears to have a positive effect on the convergence. While occasionally the difference is negligible (see Figures 5.11e and 5.11k) or even a bit on the negative side (Figures 5.11b and 5.11j), there are many instances where sampling based on the boundary rank leads to clearly better results. Convincing examples can be found with the mesh `trex`, of which the results are displayed in Figures 5.11i and 5.11l. A possible explanation for the effectiveness of `RANK` on this mesh is the many extruded surfaces that `trex` contains, which are not well preserved if `RANDOM` or `SWEEP` is used, and the irregularity of the vertex density of `trex`, which could hinder `DEGREE` in the long run. In general, due to the occasionally significantly positive results, we conclude that `RANK` is a nice addition to the algorithm, particularly in comparison to the baseline `RANDOM` sampling order that was originally used.

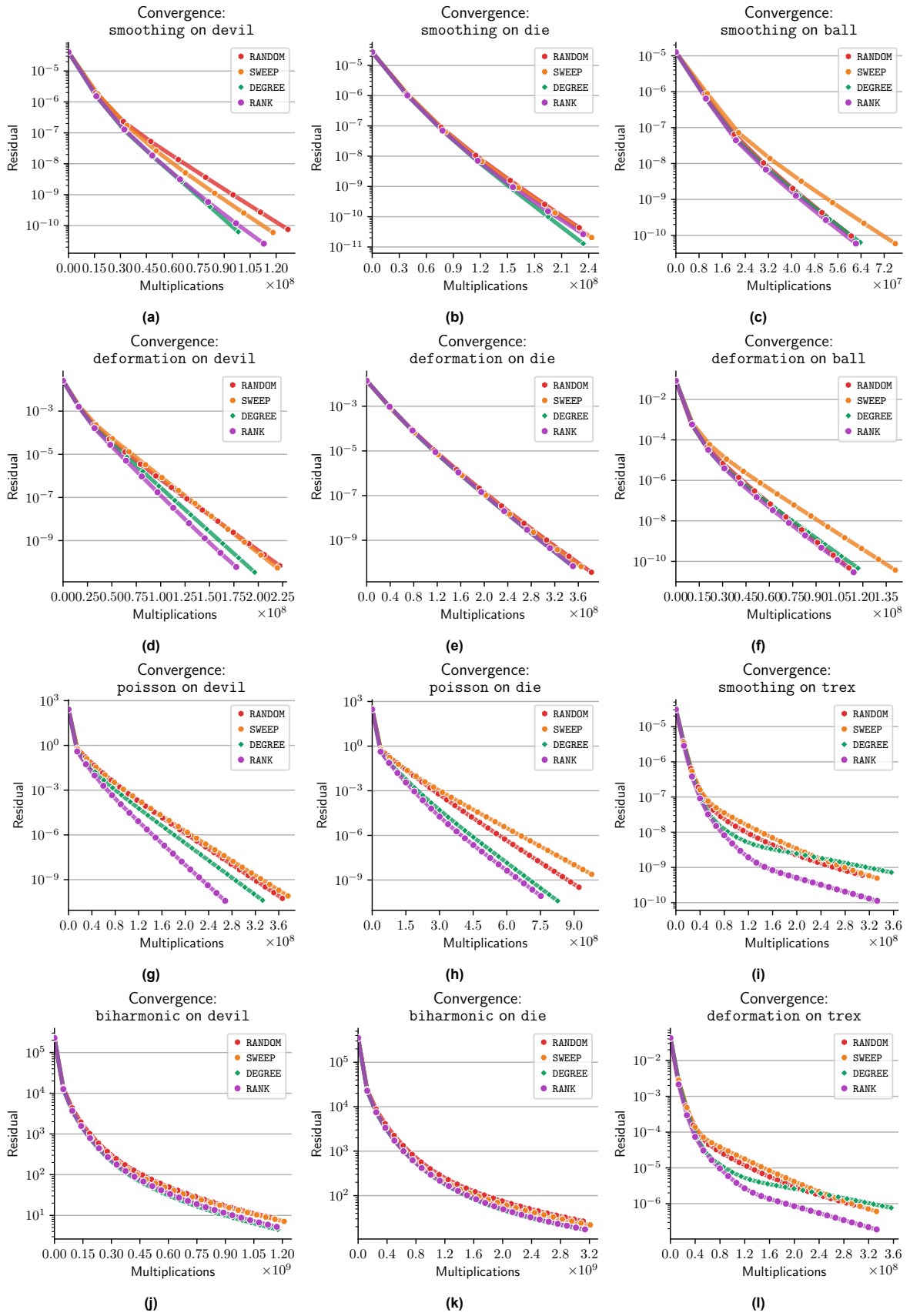


Figure 5.11: The effects of sampling order, compared and visualized.

5.2.1.4. Pit Prevention

Pit prevention ensures that no pits arise on the surface by controlling how clusters are made close to the boundary. To find out how much it contributes to the whole, we run tests with pit prevention first enabled and then disabled, and compare their results. These two settings are compared both when (unrestricted) perturbation is done and when it is not, because the extremity of the generated pits can vary depending on whether perturbation is applied or not, and it could be interesting to see how pit prevention affects the results in both situations. This gives us the following four options:

- `NONE/OFF`: Without perturbation or pit prevention;
- `ALL/OFF`: With perturbation, but without pit prevention;
- `NONE/ON`: Without perturbation, but with pit prevention;
- `ALL/ON`: With perturbation *and* pit prevention.

The algorithm was run on the adaptive meshes `arc_triomphe` and `trex`, and on the isotropic meshes `devil` and `die`. For the adaptive meshes MδIS-2 was used, and for the isotropic meshes MδIS-1½. Additionally, the boundary rank-based sampling order discussed was applied. The results are displayed in Figure 5.12.

The effect of pit prevention is, in most cases, not that extreme. For example, in Figures 5.12b, 5.12e, 5.12j, and 5.12k, there appears to be practically no difference in whether it is enabled or not. The pits apparently did not cause that many issues for those problems specifically. In some cases, however, there does seem to be a slight benefit. For example, in Figures 5.12a, 5.12d and 5.12h there is a measurable difference between `ALL/ON` and `ALL/OFF`. While these variations can in principle be explained by variance, these few results are interesting enough for us to keep pit prevention in the algorithm.

In the next section, pit prevention will be seen again together with boundary-aware smoothing, a combination that shows some small and interesting potential.

5.2.1.5. Boundary-Aware Smoothing

Finally, we evaluate the effect of boundary-aware smoothing on the hierarchy construction and the solving process. The hypothesis is that the convergence rate slightly improves due to the dislocation of the points from their original location and the consequent reduction of one-row-entries in the prolongation matrix, as briefly hypothesized in [Wie+23], but does not simultaneously degrade the convergence due to any strong misrepresentation of the boundary on the coarser levels. The following options were selected to do the comparison with:

- `NONE/ON`: Without any perturbation, but with pit prevention;
- `ALL/OFF`: With full unrestricted perturbation and pit prevention;
- `BND-AW/OFF`: With boundary-aware perturbation, but without pit prevention;
- `BND-AW/ON`: With boundary-aware perturbation and pit prevention;

For the cases without boundary-aware smoothing, pit prevention is enabled, since the experiments in Section 5.2.1.4 indicated it provided a slight advantage. The results are displayed in Figure 5.13.

In a majority of cases, boundary-aware smoothing does not seem to give any advantage. Sometimes it performs roughly the same as any other setting (see Figures 5.13b and 5.13g) or boundary-aware smoothing even makes the performance a bit worse (see Figure 5.13d). It appears that, like pit prevention, most of the benefits of enabling boundary-aware smoothing are slight and merely coincidental.

However, there is an interesting result to be observed on the mesh `arc_triomphe`. It appears that enabling boundary-aware smoothing on this mesh drastically improves convergence for both problems that were run on it. This is in comparison to both having full perturbation and having no perturbation at all, meaning that the inclusion of the boundary really does make a difference. Furthermore, on `trex` boundary-aware smoothing initially seems to hinder convergence, causing `BND-AW/OFF` to plateau early on for the deformation problem. Enabling pit prevention appears to mitigate the issue, though, even causing a slight advantage over performing no perturbation at all.

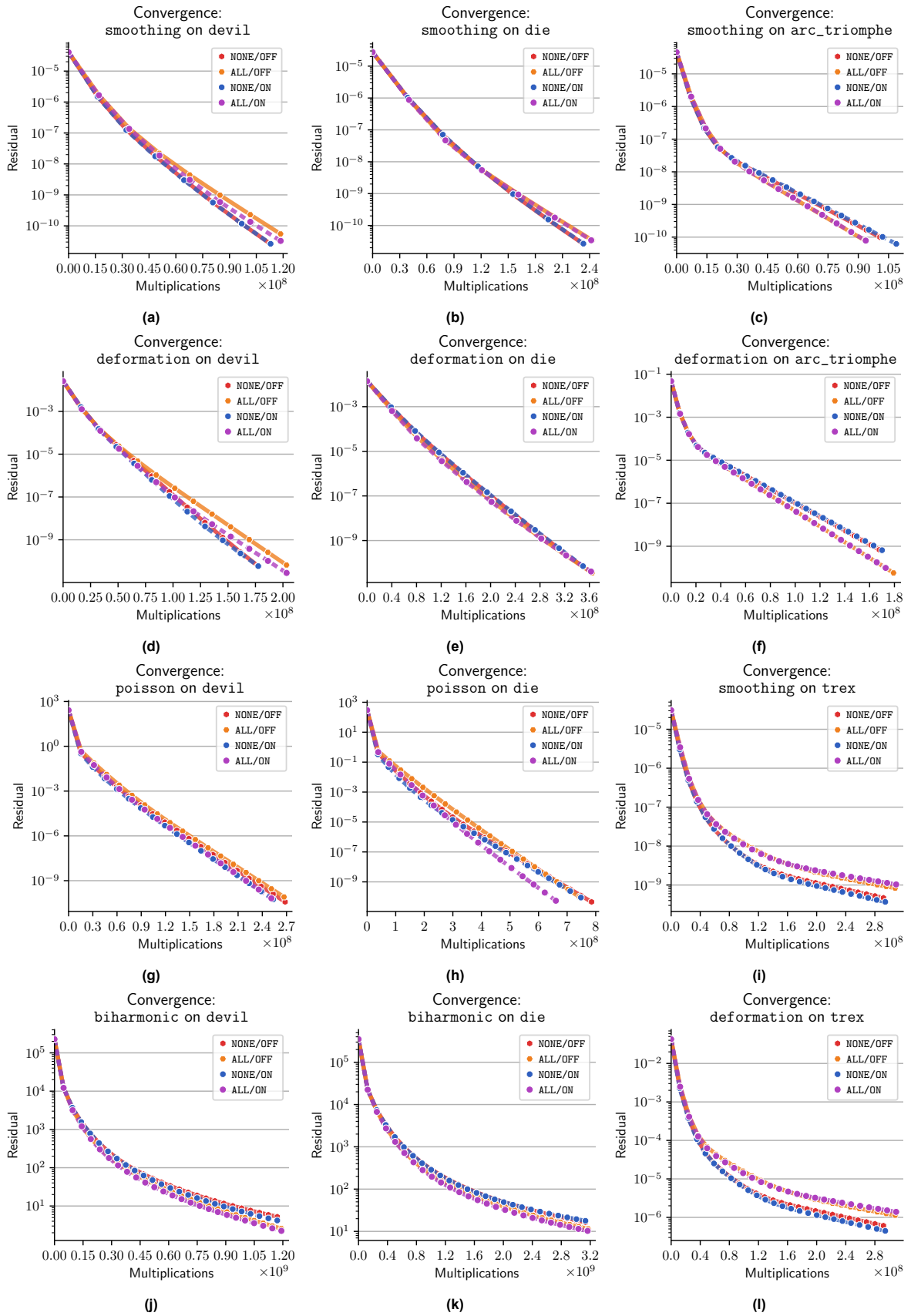


Figure 5.12: The effects of pit prevention, compared and visualized.

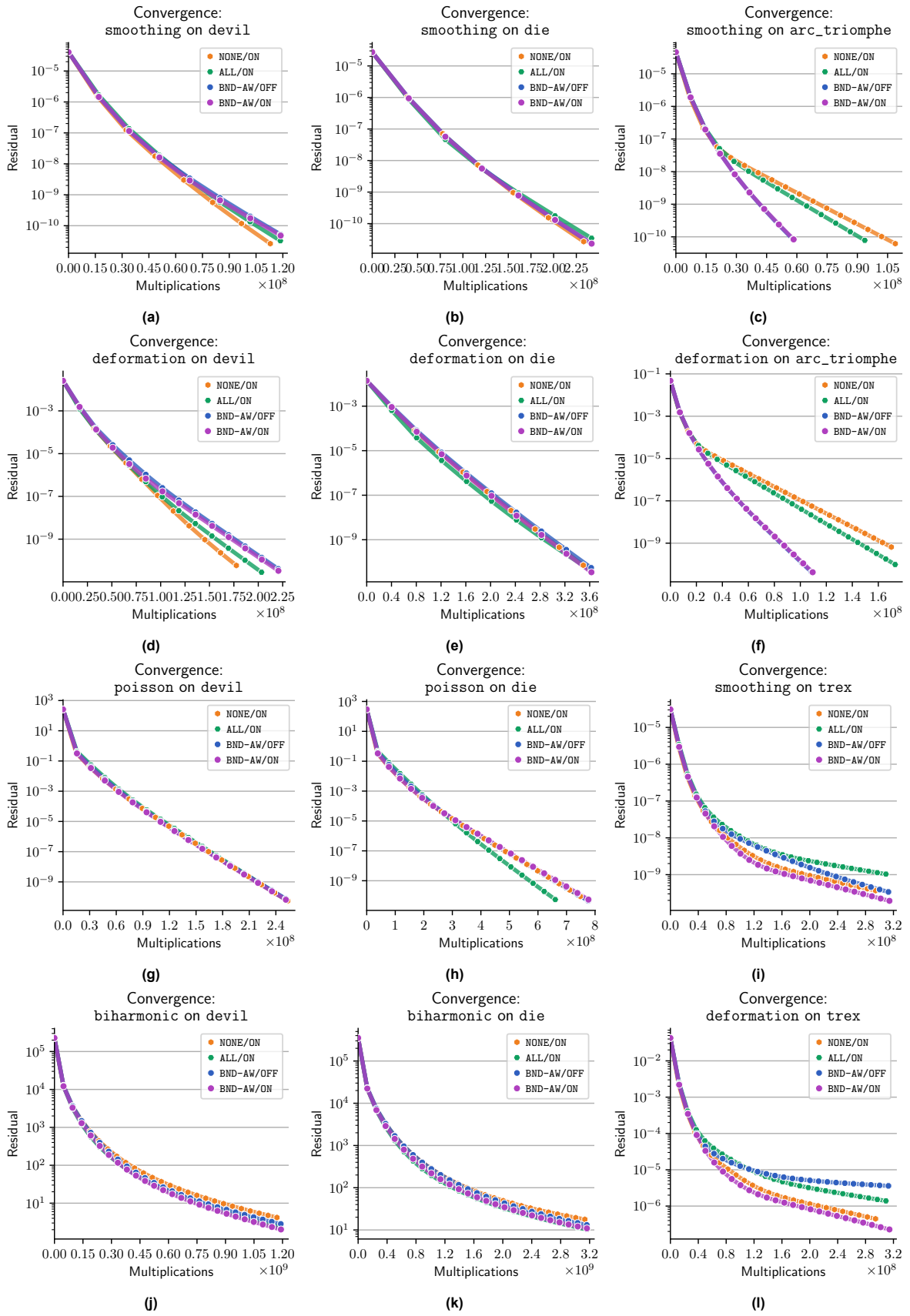


Figure 5.13: The effect of boundary-aware smoothing on the convergence of solutions, visualized.

5.2.2. Comparison to State-Of-The-Art

Up until now, Gravo MG has been extensively tested against different variations of itself. To assess whether it can compete well against other completely different techniques, we perform a short study comparing Gravo MG to some state-of-the-art algorithms that can also be applied to these kinds of problems. In this section, we first introduce the algorithms to which we compare (Section 5.2.2.1), and then compare both their solving times (Section 5.2.2.2) and hierarchy construction times (Section 5.2.2.3) to one another.

5.2.2.1. Selected Algorithms

PyAMG [Bel+23] is an algebraic multigrid library developed to be used in Python. *Algebraic multigrid* differs from geometric multigrid by not explicitly incorporating the geometry or topology of the mesh in any way or form while constructing the hierarchy or matrices. The domain remains completely unknown. The prolongation matrices are instead based purely on the system $A\mathbf{x} = \mathbf{b}$ to be solved, and any connection between the implied vertices on the same or on adjacent levels must be inferred from the values in A. By not relying on information from the mesh, algebraic multigrid methods can be applied more often and more flexibly than geometric multigrid methods; whenever there is a linear system, a hierarchy can be constructed. However, since the hierarchy is bound to the matrix that is being solved, a hierarchy cannot be reused if the linear system is replaced with another. A geometric multigrid hierarchy is not based on the system, but on the mesh, and can thus be used repeatedly to solve a variety of problems on the same domain.
PyAMG supports both **Ruge-Stüben** AMG and **Smoothed Aggregation** AMG, which operate in fundamentally different ways and lead to different kinds of hierarchies. Both are compared separately in our experiments.

The method from [Shi+06], dubbed **Shi06** from this point onward, is a geometric multigrid method relying on graph coarsening. It represents a mesh as a graph of vertices and edges, disregarding any faces and tetrahedra. It applies a similar graph coarsening technique as Gravo MG, but does not retriangulate or retetrahedralize the mesh, and instead relies on inverse distance weights for prolongation.

We will include two versions of Gravo MG in these experiments too, dubbed `GRAVO-DEF` and `GRAVO-OPT`. `GRAVO-DEF` ("default") incorporates the "natural" parameters of Gravo, i.e. those that correspond closely to the original implementation from [Wie+23]. In practice, this means sampling using MδIS-2 (see Section 4.2.3.1) using a random sampling order, and not applying any kind of pit prevention or perturbation. `GRAVO-OPT` ("optimized") is the canonical implementation discussed before; it is considered the real algorithmic output of this thesis.

The list of iterative multigrid methods that will be put alongside each other is thus as follows:

- `AMG-RS`: An implementation of Ruge-Stüben algebraic multigrid, given by `pyamg`;
- `AMG-SA`: An implementation of Smoothed Aggregation algebraic multigrid, given by `pyamg`;
- `SHI06`: The graph-based Shi06-algorithm from [Shi+06];
- `GRAVO-DEF`: The Gravo MG algorithm, configured with "default" parameters;
- `GRAVO-OPT`: The Gravo MG algorithm, configured with the optimization features discussed in Section 4.2.3.

Aside from multigrid methods, there also exist various direct solvers, which can solve arbitrary linear systems $A\mathbf{x} = \mathbf{b}$ without relying on any kind of multigrid technique. One example of such a solver is the **PARDISO** solver [Sch+01], a direct solver that employs LU-factorization and concurrency to accelerate the solving process. When dealing with sparse linear systems, it excels in solving speed to the point where it can compete with multigrid methods even on large problems. It is therefore interesting to determine at what point iterative methods, such as Gravo MG, become more appealing than a state-of-the-art direct solver.
To directly interface with PARDISO from the experiment setup, we make use of the Python package `pypardiso`[1].

[1] `https://github.com/haasad/PyPardiso`

5.2.2.2. Comparing Convergence

To compare the convergence in a fair way, the experiments were set up such that each algorithm is given the same mesh and the same corresponding problem. They would then proceed to generate a sequence of prolongation matrices that define the corresponding multigrid hierarchy and return them. This sequence was then inserted into a standard multigrid solver, along with the linear system it was created for, and the system was iteratively solved until a certain residual error was achieved or a certain number of cycles had been executed. PARDISO was not included in these experiments, as it is a direct solver, not an iterative one. PARDISO is individually compared in a different format in Section 5.2.2.4.

The meshes that were chosen for this experiment were the adaptive meshes `ball`, `trex`, `arc_triomphe`, and `asteroid`, and the isotropic meshes `devil`, `die`, `venus`, and `lovecraft`. The results are displayed in Figure 5.14.

Most of the plots indicate a clear advantage of Gravo over the other algorithms. Convincing examples of this are the plots given in Figures 5.14e, 5.14v, and 5.14x, though many more corroborate the same fact to varying degrees of intensity. In the vast majority of the executed experiments, either `GRAVO-DEF` or `GRAVO-OPT` dominates with quite a margin, demonstrating the benefit of using barycentric weights in combination with geometric multigrid. A notable observation is how the patterns in the plots can be roughly categorized by their corresponding linear system.

On Poisson problems (e.g. Figure 5.14g), it appears that `AMG-RS` and `SHI06` fall off quite fast, while `GRAVO-DEF` and `GRAVO-OPT` keep marching on downwards. For a brief moment in time, `AMG-SA` appears to consistently reside below `GRAVO-DEF`, though after a certain number of iterations, `GRAVO-DEF` catches up, as `AMG-SA` struggles to reduce the residual error, likely having trouble with a particular region of the mesh at which the prolongation is not representative of an actual geometry. The difference between `GRAVO-DEF` and `GRAVO-OPT` can be observed as well, with the optimizations clearly giving Gravo an upper hand in reducing the error.

On biharmonic problems (e.g. Figure 5.14j), a similar advantage can be observed. Where the other algorithms tend to stagnate, Gravo not only reduces the error to the same extent in a fraction of the time, but it also carries on to reduce it to a significantly higher degree. Introducing the optimizations just improves the situation, occasionally allowing the solver to maintain a higher pace even after already having conquered a head start, as seen in Figure 5.14v, for example.

On smoothing problems (e.g. Figure 5.14a), the effects of Gravo are less accentuated. In several situations, other approaches, such as `AMG-SA` and `SHI06`, are similarly efficient or even slightly better (e.g. Figures 5.14a and Figures 5.14c). On other instances, however, an improvement by Gravo can be observed, such as in Figures 5.14b. Of note is that when `GRAVO-DEF` appears to give disappointing results, `GRAVO-OPT` sometimes *does* in lowering the residual error rapidly, as can be seen in Figures 5.14i and 5.14o.

On gradient deformation problems (e.g. Figure 5.14d), Gravo performs remarkably well. In most cases, it achieves convergence up to a certain error relatively fast, whereas e.g. `AMG-RS` appears to immediately flatline. `AMG-SA` and `SHI06` appear to perform generally better, with `SHI06` even outperforming `GRAVO-DEF` in Figure 5.14f. Overall, though, `GRAVO-OPT` comes out as superior in almost all of these experiments, occasionally by a magnificent margin (see Figures 5.14e, 5.14f and 5.14d), likely due to the careful preservation of the boundary of the domain. Curiously enough, on the `asteroid` mesh, of which the experiment results can be seen in Figure 5.14l, the `GRAVO-OPT` appears to perform slightly *worse* than `GRAVO-DEF`. Upon a closer look, it appears that the number of executed cycles is approximately the same; it is the *duration* of these cycles that form the problem. The optimizations generally lead to a slightly denser subsampling of vertices, which causes cycles to take longer to run. On `asteroid`, this density apparently does not improve convergence, and in fact just slows the solver down. In most cases, however, it is safe to say that `GRAVO-OPT`, and also Gravo in general, perform well on these problems.

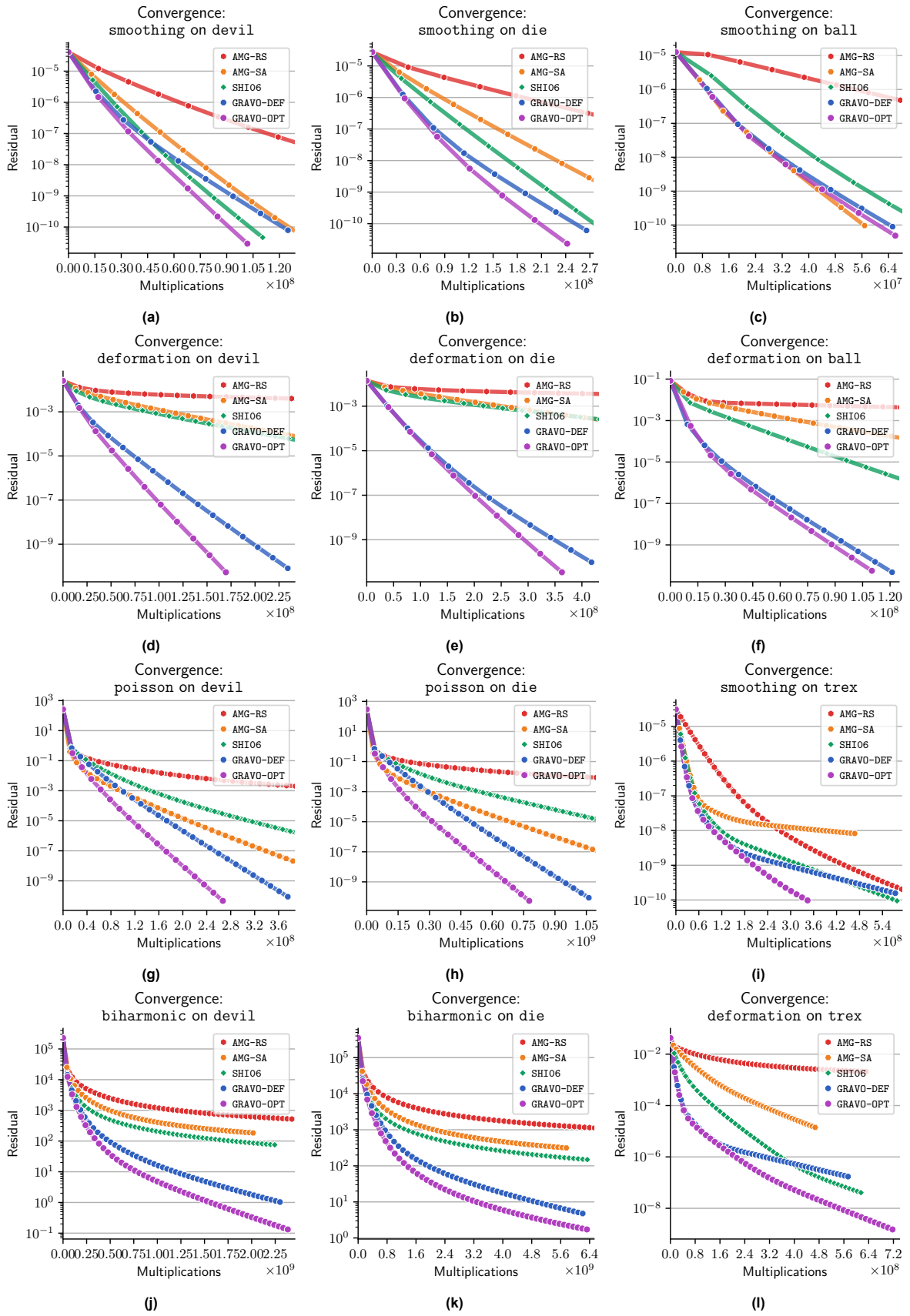


Figure 5.14: Different iterative algorithms compared.

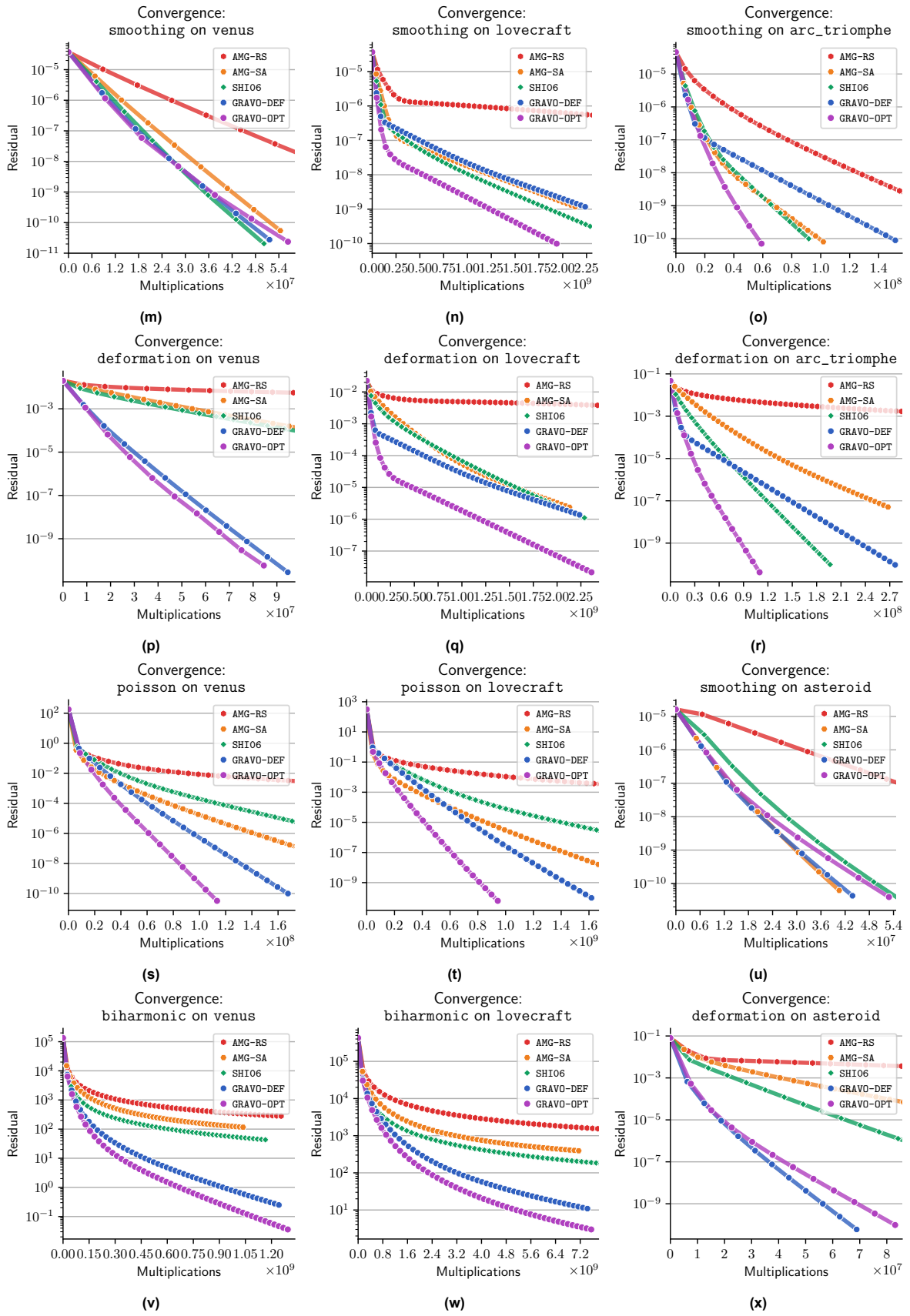


Figure 5.14: Different iterative algorithms compared.

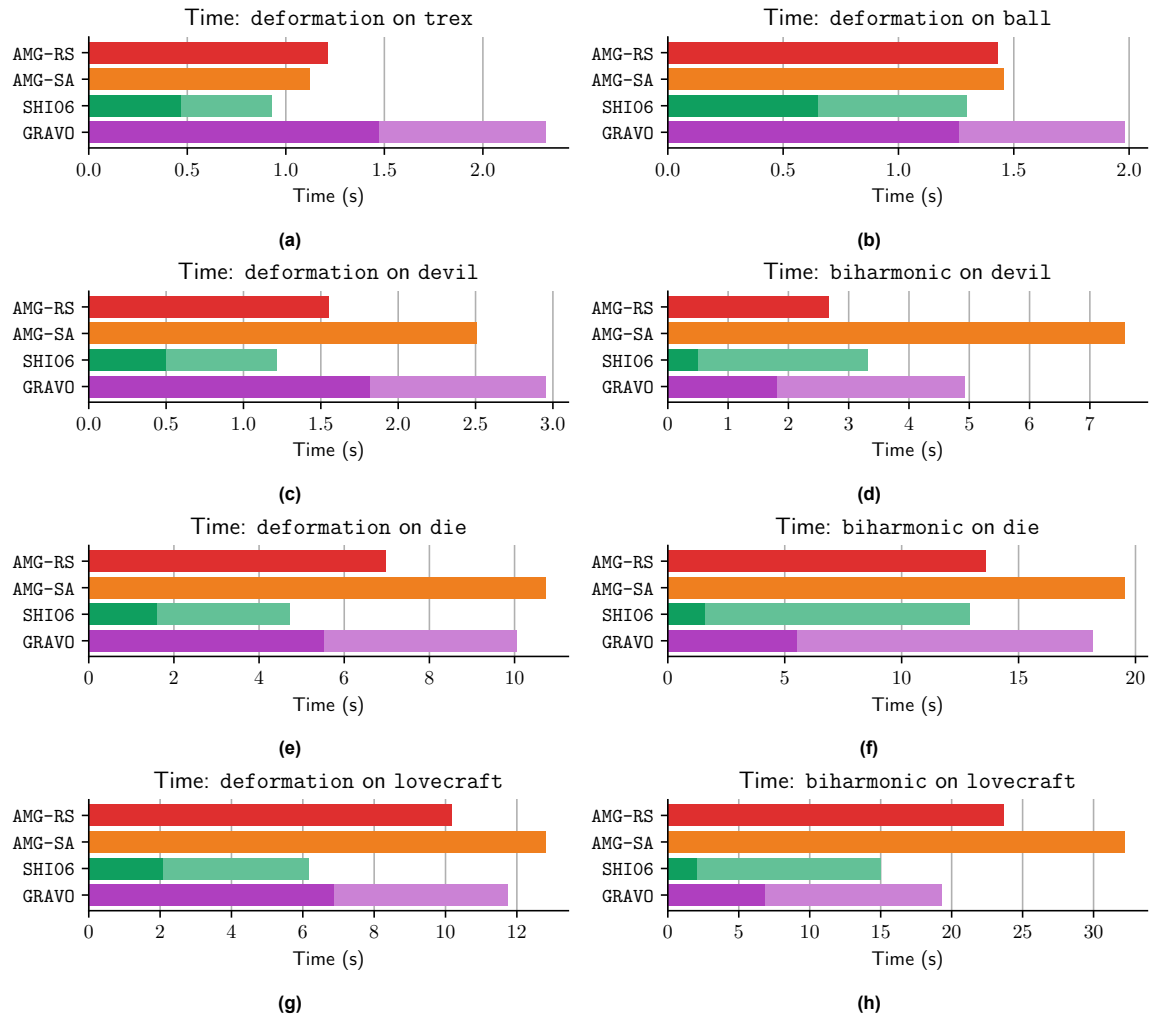


Figure 5.15: The construction times of the different algorithms compared to each other. Light regions indicate reduction times.

5.2.2.3. Comparing Construction Time

Gravo has a clear advantage when it comes to convergence, but this comes at a cost. The more stages are added to the algorithm, the longer it takes to construct the multigrid hierarchy in the first place. This work aims to provide a healthy balance between the time needed to set up the hierarchy and to use it, and thus, it is relevant how long it takes for Gravo to build it. To get a proper estimate on this, we benchmark Gravo, including its optimizations, against the same algorithms as before, but now focusing on its runtime.

The duration that is relevant is the duration between the moment both sides of the linear systems (A and $\mathbf{b}$) have been computed and the moment the solver can start executing cycles. This encapsulates both the time needed to compute the prolongation operators (the construction time) and the Galerkin projection of the matrix (the reduction time). For the AMG methods in the list, the reduction time taken by our *own* standard multigrid solver is ignored, as `pyamg` immediately reduces the matrix as it constructs the prolongation matrix, and thus the reduction time contributed by taking out the prolongation matrices and repeating the reduction procedure in another solver is ultimately redundant.

The resulting times from the executed experiments are displayed in Figure 5.15.

On the smaller domains (e.g. Figures 5.15a and 5.15b), `GRAVO` appears to need more time than the other approaches to build its hierarchy. It takes roughly twice as long for it to finalize its computation for relatively sparse systems, as `GRAVO` lingers particularly long on the tetrahedralization stage and the prolongation stage. As the domain gets larger, though, things start to shift. The algebraic multigrid

methods start taking more and more time, eventually exceeding the time needed for GRAVO to build its own hierarchy and reduce the system. The ratio between SHI06's runtime and GRAVO's runtime stays roughly constant, though. This is to be expected, as SHI06 follows a similar paradigm as GRAVO, but performs a lot fewer computations on each layer.

As the linear system gets denser, the behavior of the algorithms starts changing as well. The algebraic methods require more time to build their hierarchy, even though the geometric multigrid methods SHI06 and GRAVO merely obtain a longer reduction time, not a longer construction time. The algebraic methods, AMG-SA in particular, require much more time, eventually exceeding GRAVO's time, despite not converging as well as GRAVO at all. Furthermore, the increased reduction time also makes the gap between SHI06 and GRAVO quite a bit smaller, to the point where the construction time is no longer such a decisive factor for performance. This reduction time could vary if a better optimized multigrid solver is used, but as the system's sparsity decreases, the difference between the two performances decreases too.

5.2.2.4. Comparison to Directly Solving

Next, we compare Gravo to the direct solver PARDISO (`pypardiso`). We do this by running PARDISO on a problem and timing how long it takes to terminate and return a solution. Then, we assess how much Gravo (including optimizations) can optimize the problem within the same amount of time, taking into account the time needed to construct the hierarchy, to reduce the system, and to iteratively approach a solution to it. Since geometric multigrid hierarchies can be reused for new linear systems on the same domain, we also consider how much time Gravo would need, excluding the hierarchy construction time. This leaves us with three different settings to compare:

- GRAVO(CON+RED+SOL): Gravo MG, including its construction time and reduction time, in addition to its solving time;
- GRAVO(RED+SOL): Gravo MG, including its reduction time in addition to its solving time;
- PARDISO: The direct solver PARDISO.

The results are displayed in Figure 5.16.

The difference between the performance of the iterative solver and the direct solver depends, as can be expected, on the size of the problem. On smaller systems, such as those defined on `ball` and `trex`, the direct solver is clearly superior, already terminating before Gravo is even done creating its hierarchy (see e.g Figures 5.16a and 5.16c). If a hierarchy is already available, Gravo can perform a couple of cycles and reduce the error accordingly, but it can hardly be considered worthwhile to set aside the direct solver for the iterative solver on this scale.

Things drastically change as the domain gets bigger or the system gets denser. The effects can already be observed on a "medium-sized" mesh such as `devil`, with the dense `biharmonic` problem. Even though it takes roughly three-and-a-half minutes for the direct solver to terminate, Gravo has reduced the residual error to a relatively low amount after thirty seconds already. With the iterative solver, the user can optimize it to whatever residual error is deemed appropriate for their use case, which might not need to be that low at all.

On the large mesh `die`, it becomes much more reasonable to use the iterative solver over the direct one. PARDISO requires minutes to complete, whereas Gravo can construct a hierarchy, load in a simple linear system, and solve it to a residual error of at most 10^{-10} in less than *one* minute. When it comes to the biharmonic problem, PARDISO was not even able to find a solution, instead throwing an error code a couple of minutes into its computation. Gravo, on the other hand, was able to optimize a solution to the problem without issues. For any system that is denser or larger than these systems, Gravo is clearly a sensible choice.

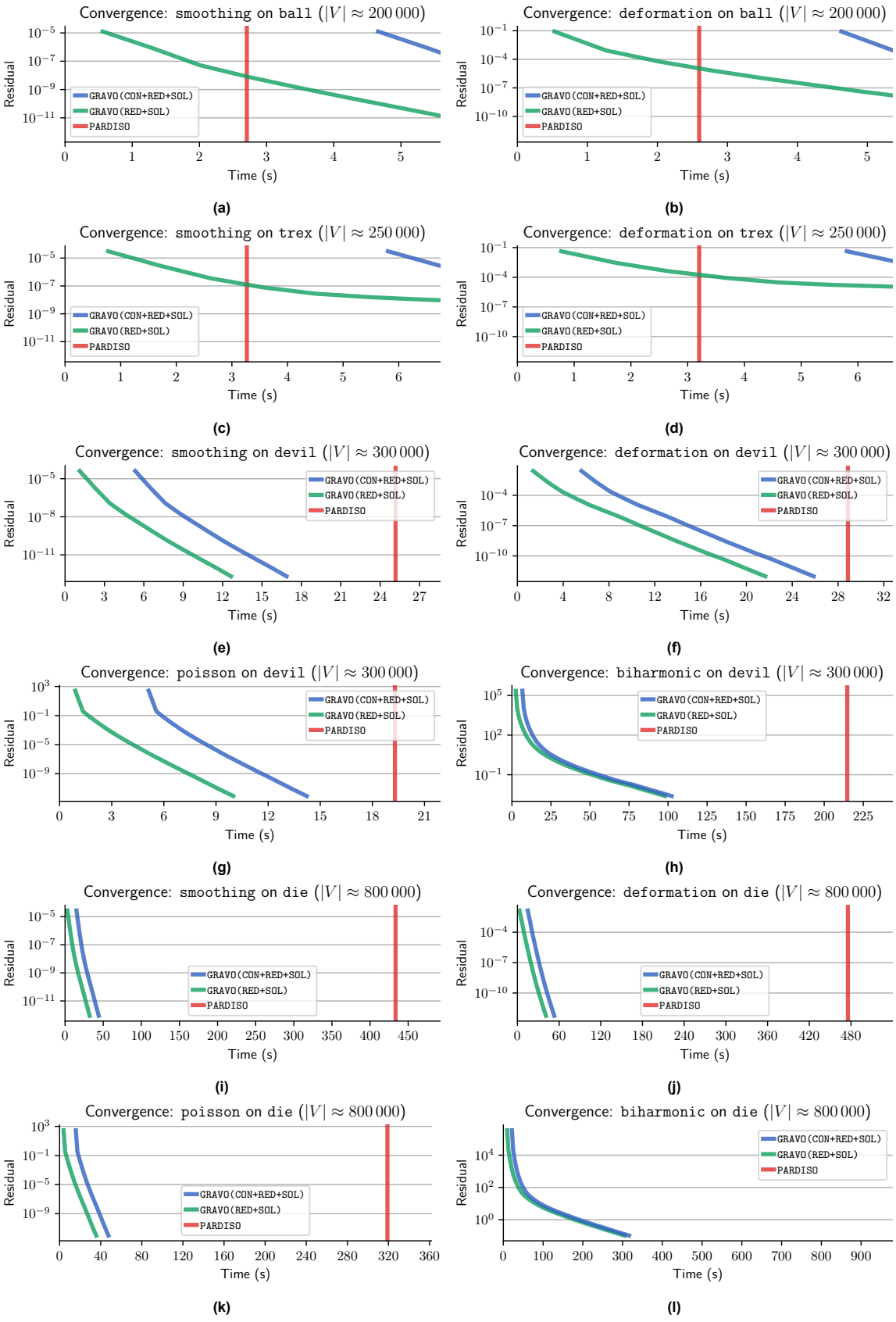


Figure 5.16: Gravo and PARDISO compared. The red vertical line indicates the amount of time needed for PARDISO to terminate. If no vertical line is drawn, that means PARDISO suffered a running failure.

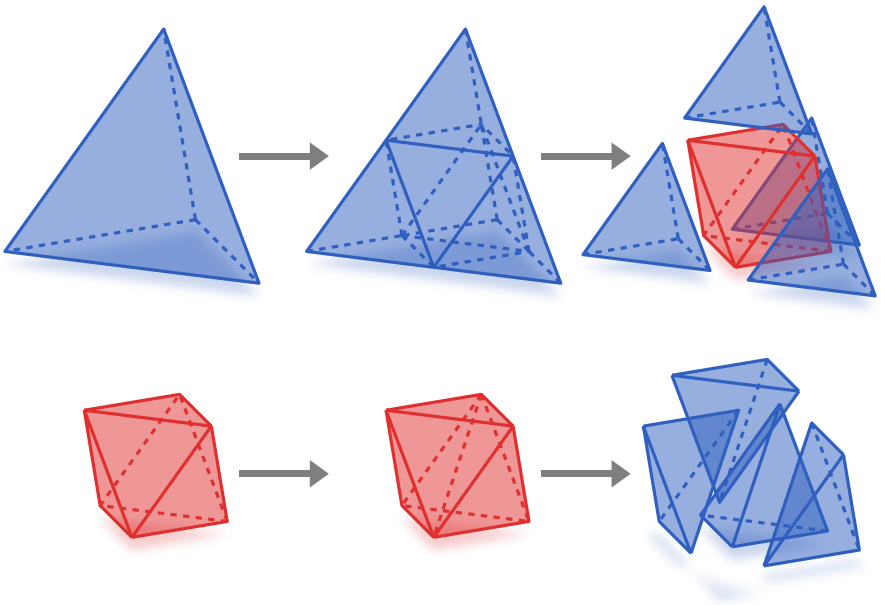


Figure 5.17: A tetrahedron refined into eight tetrahedra, by adding midpoints on each edge. The octahedron inside (red) can be split up in three different ways; one way is shown above.

5.2.3. Comparison to Prolongation through Refinement

In practice, multigrid domains are usually not constructed through coarsening, but rather through refinement. By taking a standard geometric shape representing some regular domain, such as a cuboid or a tetrahedron, and iteratively refining it in order to add more detail and degrees of freedom, one can create a multigrid hierarchy on which multigrid computations can be done. This construction method is more restrictive, since it can only be created from coarse domains that lack small shapes, rather than from complicated fine meshes that already contain a lot of detail. This might make multigrid methods that depend on refinement inapplicable for solving equations on domains with particular shapes, which is the focus of this work. However, the regularity imposed by the refinement does allow for clean and sparse prolongation operators, since each new point is defined using a set of particular coarse points; a set that can be directly used to define the prolongation operators as well.

Refining a three-dimensional Cartesian domain made out of cuboids would simply be a matter of putting an extra point at the center of every edge, face, and cell, and then splitting up each cuboid into eight smaller cuboids, each with half the size in each dimension. Refining tetrahedra can be done similarly, but requires a slightly alternate approach, since a tetrahedron is not self-similar like a cuboid is. To refine a tetrahedron, we make use of **red refinement**, as visualized in Figure 5.17. We split every edge into two equal parts, with a new point defined in the middle. These points are then connected to the midpoints of other adjacent edges with a new edge of half the length.

A tetrahedron, unlike a cuboid, is not self-similar, in the sense that eight smaller tetrahedra with the same shape cannot create a bigger tetrahedron. We can still put a new point in the middle of each edge, and connect them to other nearby points. After subtracting the four emerged tetrahedra from the corners, an octahedron remains, which can be split into four other tetrahedra in three different ways, depending on which of the three different diagonals contained within the octahedron is chosen as a common edge between all of them. A sensible choice could be to choose the diagonal with the shortest length, as that will lead to tetrahedra with the highest quality.

A refined domain can be trivially coarsened by just “undoing” the refinement. The result is a sparse prolongation operator, where every fine point that is also present in the adjacent layer will prolongate to its corresponding point with a weight of 1, and every fine point put on the midpoint of a coarse edge will prolongate to its two endpoints with a weight of $\frac{1}{2}$. Since every row in the prolongation matrix will contain at most two entries, the transfer of matrices and vectors can be done more efficiently, and the linear systems obtained through Galerkin multigrid will also be sparser and therefore easier to apply

Gauss-Seidel to. Because of this, a multigrid hierarchy defined this way could be considered an “ideal” hierarchy, which might not be guaranteed to strictly perform better than any other hierarchy for all problems, but at least provides a reliable and sparse baseline that should not show any odd behavior.

To evaluate the performance of Gravo MG, we compare it to a multigrid hierarchy generated through refinement. This experiment is executed by taking a simple tetrahedral mesh, refining it a certain number of times (say N), and then applying Gravo MG on the finest generated mesh, limiting the number of coarsening iterations to N as well. We can then compare the prolongation operators generated through the refinement to the prolongation operators generated by Gravo MG afterwards, and compare their convergence and runtime when applying them to solve a couple of problems.

The mesh that has been chosen is a simple cube mesh with a total of 729 vertices. The cube has been refined a total of three times, leading to a cube with 274625 vertices in total. The problem that is solved is the Poisson problem as described in Section 5.1.2.3, up to a residual error of 10^{-10}. Both Gravo MG and Shi06 have been compared to the refinement operators to properly contextualize the magnitude of the differences between the three algorithms.

The results are displayed graphically in Figure 5.18 and quantitatively in Table 5.1.

What stands out is the similarity between the convergence of the refinement operators and Gravo MG when set out over the number of cycles. Apparently, Gravo MG can, in the right setting, compete with idealistic operators, suggesting its practicality for domains with simpler and more predictable geometry. A large part of this performance can be explained by the fact that the sampling order (see Section 4.2.3.2) and boundary-aware smoothing (Section 4.2.3.4) ensure a near-perfect preservation of the corners, the ridges, and the surface of the cube, something that algorithms such as Shi06 do not guarantee. The fact that the interior points are spread out differently and the point density decays at a different rate appears to make little difference for the convergence of the system.

It should be noted that, in practice, despite the similarity in the number of cycles, Gravo MG’s operators would still take more time to converge than the refinement operators. The cause of this is the difference in sparsity of the prolongation matrices between the two algorithms. Gravo MG guarantees that every row in the prolongation matrix contains at most four entries, whereas the refined prolongation operator limits it to at most two. As a result, both Gravo MG’s prolongation matrices and consequently their reduced left-hand-side matrices are significantly denser, and thus cycles involve more multiplications and take up more time. This is deemed a disadvantage that cannot be overcome, and also does not need to be overcome, since this is caused by not taking advantage of the regular structure of the cube. Gravo MG is designed to operate on meshes that do not have any such structure, in which a regular structure is very unlikely to be present in the first place.

Algorithm	$\lvert V_k \rvert$ per layer	$\mathrm{nonzero}(P^k_{k-1})$ per layer	$\mathrm{nonzero}(A_k)$ per layer	Cycles
Refinement	274625 35937 4913 729	513313 66961 9097	3680781 513313 66961 9097	14
Gravo MG	274625 21982 3212 500	1066832 81750 11226	3680781 1197224 274284 43812	14
Shi06	274625 38195 7964 1924	544323 70801 14086	3680781 940834 282700 80586	73

Table 5.1: A numerical overview of the size of the systems in the respective hierarchies, including the number of cycles required for solving them.

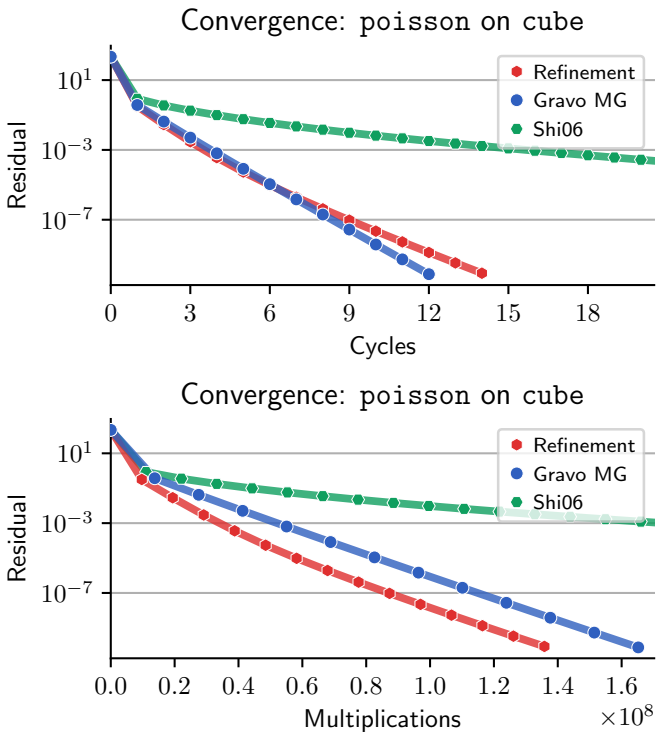


Figure 5.18: A qualitative comparison between refinement operators, Gravo MG and Shi06. The same experiment is plotted over the number of cycles and the number of multiplications.

5.3. Discussion

The above experiments have demonstrated that Gravo creates hierarchies that allow multigrid solvers to minimize the residual error very rapidly. The usage of a barycentric weighting scheme is a large contributor to this phenomenon. This can be rationalized by the consideration that barycentric weights act as piecewise linear weights elevated to higher dimensions. By being able to apply this kind of close-to-linear interpolation in the transfer of vectors, solutions to problems in which the solution values tend to vary smoothly and predictably, such as the Dirichlet problem, can be preserved better.

The added dimension also brings with it new points of failure and new opportunities to optimize. Most of the introduced optimizations were focused on the boundary of the mesh, since the boundary appeared to have the most influence on the usability of the hierarchy. Among other possible reasons, this is likely because the boundary is the most important feature to define the domain, or because points on the boundary have fewer neighbors, usually all on one side.

A few things that have been more influential and important than others. First off, the inclusion of the boundary rank made a significant change to the preservation of the sharp details of the domain. This has shown to have a very positive effect on convergence, likely as the residual error around the sharp points was not addressed well, causing outliers that hinder convergence in the region around it. Some other attempted optimizations were less fruitful, or at least not in many contexts, such as pit prevention, which only showed an improvement in a small number of cases.

It is important to acknowledge that, despite strong attempts to get an accurate estimation of the performance of the optimizations, it is possible that some of the given results are a bit skewed, particularly in the ablation study. This is, in part, because none of the parameters and stages operate in isolation, and enabling any two features might result in interaction between them, leading to different behavior than if they operated in isolation. It would be excessive to exhaustively simulate every possible combination of parameters, and thus, the choice was made, for the sake of this thesis, to accept the experiment setup and its results for what they are.

In terms of construction time, Gravo holds up quite well. Of particular relevance is its scalability in

terms of the number of vertices in the input mesh. Whereas algebraic multigrid methods start to struggle, Gravo maintains an approximately linear growth in its runtime, which eventually grants it superiority. The algorithm from [Shi+06] remains the fastest option, and is generally around three times faster than Gravo. As the input mesh gets more complex, this will result in an increasingly larger difference between the construction times of the two algorithms, which becomes even more important if the construction needs to be repeatedly executed for many different geometries. Nevertheless, Gravo's performance during the solving stage justifies this overhead, and if the geometry hierarchy generated by these methods is reused, the few seconds lost in the construction time ought not to matter too much.

The similarity in construction between Gravo and these relatively simplistic and fast approaches indicates that Gravo itself can be considered a "fast" method itself. It does so generally without compromising on the quality of the resulting hierarchy, as demonstrated by the comparison to the prolongation operators defined through iterated refinement. We thus estimate, even though no direct comparison was done due to practical reasons, that Gravo can compete with methods from [BKS11] and [SK11], which have mostly been praised for their fast convergence, and not for their fast construction.

6

Conclusion

6.1. Summary

In this work, we have presented an adaptation of Gravo MG for tetrahedral meshes. This adaptation is able to construct hierarchies for arbitrary tetrahedral meshes. It is performant, and can build a hierarchy and reduce a system for larger meshes in a time comparable to that of other light-weight choices, such as algebraic multigrid methods. Just like the original Gravo MG, it does not require a perfect mesh and is able to operate on any domain, as long as the vertices and edges are defined.

Gravo MG can be an interesting choice for researchers and engineers who need to solve discretized PDEs regularly. The balance between construction time and convergence time makes it appealing in situations where high precision is necessary, but long computation times cannot be afforded, and due to the flexibility of the algorithm, little preprocessing of the domain is needed, making it a near out-of-the-box option for any problem in a geometric setting. Since linear systems themselves appear in many applications, the contributions provided in this work could potentially be of relevance in any field that requires any computation to be done on irregular meshes.

6.2. Future Work

Research can always be improved, and as such, we highlight a few possible branches that could be expanded upon if someone were to continue ours.

6.2.1. Better Control on Vertex Density

An observation made in our work is that different regions in the domain do not need to have the same level of detail in the multigrid hierarchy. A good preservation of the boundary points has shown to be much more important than that of the interior points of the mesh. In our redesigned sampling approach, we create a distinction between points by adapting the exclusion condition in our implementation of MδIS. This is, however, a quite crude way of encouraging the desired behavior, which is only effective up to a certain degree and can provide odd-looking meshes after having executed a coarsening pass.

While some experimentation was done with adaptive sizing fields, no useful measure was found to further take advantage of the distinctive properties of the points and their position in the mesh. Such a measure, or another way to control the vertex density to the solver's advantage, could be of great help for practical use cases.

6.2.2. Different Definition of Boundary Rank

Boundary ranks are currently given as input to the algorithm, and thus, the responsibility of defining them is put on the user. These boundary ranks can, in principle, be painted onto the surface, but this requires effort and manual preparation of the input, which is often not desirable and not always feasible. In the experiments, we made use of a rigorously defined metric, promoting points to a rank if an adjacent dihedral angle or its curvature exceeded a certain hard-coded value, which did not always

lead to continuous ridges or marked corners.

A built-in algorithm to determine the ridges and corners on a mesh, at least on the input mesh, could benefit the usability of Gravo. It would remove a required parameter from the input and thus put less responsibility on the user. The used criteria from the experiments would be a suboptimal choice, due to their unreliability. It would be of interest to design a more sophisticated algorithm that can quickly and adaptively decide which points deserve priority during the sampling stage based on the geometry of the input.

6.2.3. Robustness against Bad Spots

While running experiments, it has stood out how sensitive the performance of multigrid solvers was to small irregularities in the input mesh. This could be an inconsistency in the vertex density on the input mesh, a particularly badly tetrahedralized region, or something else entirely. Spots in which such phenomena occur have shown to need much more time to converge than others. This consequently prevents close parts of the domain from converging as well, and overall causes a large outlier in the solution and a large residual error that is difficult to get rid of.

Figuring out how to make Gravo more robust against these kinds of issues would make it much more reliable and trustworthy. First, it would be necessary to figure out what problems are actually caused by these spots and how to identify them beforehand. Then, the question would be how to address them without increasing the runtime so much that Gravo loses its appeal.

6.2.4. Experiments in Real-Life Settings

Gravo has been tested on a variety of problems that show different types of behavior. While these problems have their own use cases and applications in the real world, they can be described as rather “elementary”. Their complexity is quite limited, and they do not fully reflect the context in which multigrid solvers are applied in industrial settings. An example would be linear elasticity, which is used to simulate elastic bodies, a task well-suited for a multigrid method on an irregular grid. Seeing how Gravo’s hierarchy would perform when given these types of problems would be insightful, and could motivate its introduction into real applications or motivate research into further optimizing it and addressing its shortcomings.

References

[AKS03] Burak Aksoylu, Andrei Khodakovsky, and Peter Schröder. “Multilevel Solvers for Unstructured Surface Meshes”. In: *SIAM Journal on Scientific Computing* 26 (July 2003). doi: 10.1137/S1064827503430138.

[And+21] Robert Anderson et al. “MFEM: A modular finite element methods library”. In: *Computers & Mathematics with Applications*. Development and Application of Open-source Software for Problems with Numerical PDEs 81 (Jan. 2021), pp. 42–74. issn: 0898-1221. doi: 10.1016/j.camwa.2020.06.009. url: https://www.sciencedirect.com/science/article/pii/S0898122120302583.

[Arn+21] Daniel Arndt et al. “The deal.II finite element library: Design, features, and insights”. In: *Computers & Mathematics with Applications*. Development and Application of Open-source Software for Problems with Numerical PDEs 81 (Jan. 2021), pp. 407–422. issn: 0898-1221. doi: 10.1016/j.camwa.2020.02.022. url: https://www.sciencedirect.com/science/article/pii/S0898122120300894.

[Bas+21] Peter Bastian et al. “The Dune framework: Basic concepts and recent developments”. In: *Computers & Mathematics with Applications*. Development and Application of Open-source Software for Problems with Numerical PDEs 81 (Jan. 2021), pp. 75–112. issn: 0898-1221. doi: 10.1016/j.camwa.2020.06.007. url: https://www.sciencedirect.com/science/article/pii/S089812212030256X.

[Bel+23] Nathan Bell et al. “PyAMG: Algebraic Multigrid Solvers in Python”. In: *Journal of Open Source Software* 8.87 (July 2023), p. 5495. issn: 2475-9066. doi: 10.21105/joss.05495. url: https://joss.theoj.org/papers/10.21105/joss.05495.

[BKS11] Peter R. Brune, Matthew G. Knepley, and L. Ridgway Scott. *Unstructured Geometric Multigrid in Two and Three Dimensions on Complex and Graded Meshes*. arXiv:1104.0261 [cs] version: 2. Apr. 2011. doi: 10.48550/arXiv.1104.0261. url: http://arxiv.org/abs/1104.0261.

[BL11] Achi Brandt and Oren E. Livne. *Multigrid Techniques*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, Jan. 2011. isbn: 978-1-61197-074-6. doi: 10.1137/1.9781611970753. url: https://epubs.siam.org/doi/book/10.1137/1.9781611970753.

[Bra77] Achi Brandt. “Multi-Level Adaptive Solutions to Boundary-Value Problems”. In: *Mathematics of Computation* 31.138 (1977). Publisher: American Mathematical Society, pp. 333–390. issn: 0025-5718. doi: 10.2307/2006422. url: https://www.jstor.org/stable/2006422.

[Bra86] Achi Brandt. “Algebraic multigrid theory: The symmetric case”. In: *Applied Mathematics and Computation* 19.1 (July 1986), pp. 23–56. issn: 0096-3003. doi: 10.1016/0096-3003(86)90095-0. url: https://www.sciencedirect.com/science/article/pii/0096300386900950.

[BS99] Randolph Bank and Andrew Sherman. “Some Refinement Algorithms And Data Structures For Regular Local Mesh Refinement”. In: *Applications of Mathematics and Computing to the Physical Sciences* (June 1999).

[CC02] T Y Chao and W K Chow. “A REVIEW ON THE APPLICATIONS OF FINITE ELEMENT METHOD TO HEAT TRANSFER AND FLUID FLOW”. en. In: (2002).

[Coe+24] David Coeurjolly et al. “Polygon Mesh Processing”. In: *CGAL User and Reference Manual*. 6.0.1. CGAL Editorial Board, 2024. url: https://doc.cgal.org/6.0.1/Manual/packages.html#PkgPolygonMeshProcessing.

[CPS13] Keenan Crane, Ulrich Pinkall, and Peter Schröder. “Robust fairing via conformal curvature flow”. en. In: *ACM Transactions on Graphics* 32.4 (July 2013), pp. 1–10. issn: 0730-0301, 1557-7368. doi: 10.1145/2461912.2461986. url: https://dl.acm.org/doi/10.1145/2461912.2461986.

[Cra19] Keenan Crane. “The n-dimensional cotangent formula”. en. In: (2019).

[DGW11] Christian Dick, Joachim Georgii, and Ruediger Westermann. “A Hexahedral Multigrid Approach for Simulating Cuts in Deformable Objects”. In: *IEEE Transactions on Visualization and Computer Graphics* 17.11 (Nov. 2011), pp. 1663–1675. issn: 1941-0506. doi: 10.1109/TVCG.2010.268. url: https://ieeexplore.ieee.org/abstract/document/5674032.

[GSH07] Nicholas I. M. Gould, Jennifer A. Scott, and Yifan Hu. “A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations”. en. In: *ACM Transactions on Mathematical Software* 33.2 (June 2007), p. 10. issn: 0098-3500, 1557-7295. doi: 10.1145/1236463.1236465. url: https://dl.acm.org/doi/10.1145/1236463.1236465.

[GV96] Gene H. Golub and Charles F. Van Loan. *Matrix computations (3rd ed.)* USA: Johns Hopkins University Press, 1996. isbn: 0-8018-5414-8.

[Hem95] P W Hemker. “Finite Volume Multigrid for 3D-Problems”. en. In: (1995).

[Hop96] Hugues Hoppe. “Progressive meshes”. en. In: (1996).

[Jac15] Alec Jacobson. *How does Galerkin multigrid scale for irregular grids?* 2015. url: https://www.alecjacobson.com/weblog/4383.html.

[JP+18] Alec Jacobson, Daniele Panozzo, et al. *libigl: A simple C++ geometry processing library*. 2018.

[JW00] H. Jasak and H. G. Weller. “Application of the finite volume method and unstructured meshes to linear elasticity”. en. In: *International Journal for Numerical Methods in Engineering* 48.2 (2000), pp. 267–287. issn: 1097-0207. doi: 10.1002/(SICI)1097-0207(20000520)48:2<267::AID-NME884>3.0.CO;2-Q. url: https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291097-0207%2820000520%2948%3A2%3C267%3A%3AAID-NME884%3E3.0.CO%3B2-Q.

[Kob+98] Leif Kobbelt et al. “Interactive multi-resolution modeling on arbitrary meshes”. en. In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques - SIGGRAPH ’98*. Not Known: ACM Press, 1998, pp. 105–114. isbn: 978-0-89791-999-9. doi: 10.1145/280814.280831. url: http://portal.acm.org/citation.cfm?doid=280814.280831.

[Liu+21] Hsueh-Ti Derek Liu et al. *Surface Multigrid via Intrinsic Prolongation*. arXiv:2104.13755. May 2021. doi: 10.48550/arXiv.2104.13755. url: http://arxiv.org/abs/2104.13755.

[MST10] A McAdams, E Sifakis, and J Teran. “A parallel multigrid Poisson solver for fluids simulation on large grids”. en. In: (2010).

[NGH04] Xinlai Ni, Michael Garland, and John C. Hart. “Fair morse functions for extracting the topological structure of a surface mesh”. en. In: *ACM Transactions on Graphics* 23.3 (Aug. 2004), pp. 613–622. issn: 0730-0301, 1557-7368. doi: 10.1145/1015706.1015769. url: https://dl.acm.org/doi/10.1145/1015706.1015769.

[PPM92] J. Peraire, J. Peiro, and K. Morgan. “A 3D finite element multigrid solver for the Euler equations”. In: NTRS Author Affiliations: NASA Headquarters, Imperial College of Science, Technology, and Medicine, Swansea, University NTRS Report/Patent Number: AIAA PAPER 92-0449 NTRS Document ID: 19920049041 NTRS Research Center: Legacy CDMS (CDMS). Jan. 1992. url: https://ntrs.nasa.gov/citations/19920049041.

[RL03] N. Ray and B. Levy. “Hierarchical least squares conformal map”. In: *11th Pacific Conference onComputer Graphics and Applications, 2003. Proceedings.* Canmore, Alta., Canada: IEEE Comput. Soc, 2003, pp. 263–270. isbn: 978-0-7695-2028-5. doi: 10.1109/PCCGA.2003.1238268. url: http://ieeexplore.ieee.org/document/1238268/.

[RS87] J. W. Ruge and K. Stüben. “4. Algebraic Multigrid”. In: *Multigrid Methods*. Frontiers in Applied Mathematics. Society for Industrial and Applied Mathematics, Jan. 1987, pp. 73–130. isbn: 978-1-61197-188-0. doi: 10.1137/1.9781611971057.ch4. url: https://epubs.siam.org/doi/10.1137/1.9781611971057.ch4.

[Sch+01] Olaf Schenk et al. “PARDISO: a high-performance serial and parallel sparse linear solver in semiconductor device simulation”. In: *Future Generation Computer Systems*. I. High Performance Numerical Methods and Applications. II. Performance Data Mining: Automated Diagnosis, Adaption, and Optimization 18.1 (Sept. 2001), pp. 69–78. issn: 0167-739X. doi: 10.1016/S0167-739X(00)00076-5. url: https://www.sciencedirect.com/science/article/pii/S0167739X00000765.

[Sha+19] Nicholas Sharp et al. *Polyscope*. 2019.

[Shi+06] Lin Shi et al. “A fast multigrid algorithm for mesh deformation”. In: *ACM Trans. Graph.* 25.3 (2006), pp. 1108–1117. issn: 0730-0301. doi: 10.1145/1141911.1142001. url: https://dl.acm.org/doi/10.1145/1141911.1142001.

[SK11] Xinwei Shi and Patrice Koehl. “Adaptive skin meshes coarsening for biomolecular simulation”. In: *Computer Aided Geometric Design* 28.5 (June 2011), pp. 307–320. issn: 0167-8396. doi: 10.1016/j.cagd.2011.04.001. url: https://www.sciencedirect.com/science/article/pii/S0167839611000331.

[SNI91] M. Storti, N. Nigro, and S. Idelsohn. “Multigrid methods and adaptive refinement techniques in elliptic problems by finite element methods”. In: *Computer Methods in Applied Mechanics and Engineering* 93.1 (Dec. 1991), pp. 13–30. issn: 0045-7825. doi: 10.1016/0045-7825(91)90113-K. url: https://www.sciencedirect.com/science/article/pii/004578259190113K.

[VMB96] P. Vaněk, J. Mandel, and M. Brezina. “Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems”. en. In: *Computing* 56.3 (Sept. 1996), pp. 179–196. issn: 1436-5057. doi: 10.1007/BF02238511. url: https://doi.org/10.1007/BF02238511.

[Wes04] Pieter Wesseling. *An introduction to multigrid methods*. eng. Corr. reprint. Philadelphia: R.T. Edwards, 2004. isbn: 978-1-930217-08-9.

[Wie+23] Ruben Wiersma et al. “A Fast Geometric Multigrid Method for Curved Surfaces”. en. In: *Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Proceedings*. Los Angeles CA USA: ACM, July 2023, pp. 1–11. isbn: 979-8-4007-0159-7. doi: 10.1145/3588432.3591502. url: https://dl.acm.org/doi/10.1145/3588432.3591502.

[XTL19] Zangyueyang Xian, Xin Tong, and Tiantian Liu. “A scalable galerkin multigrid method for real-time simulation of deformable objects”. en. In: *ACM Transactions on Graphics* 38.6 (Dec. 2019), pp. 1–13. issn: 0730-0301, 1557-7368. doi: 10.1145/3355089.3356486. url: https://dl.acm.org/doi/10.1145/3355089.3356486.

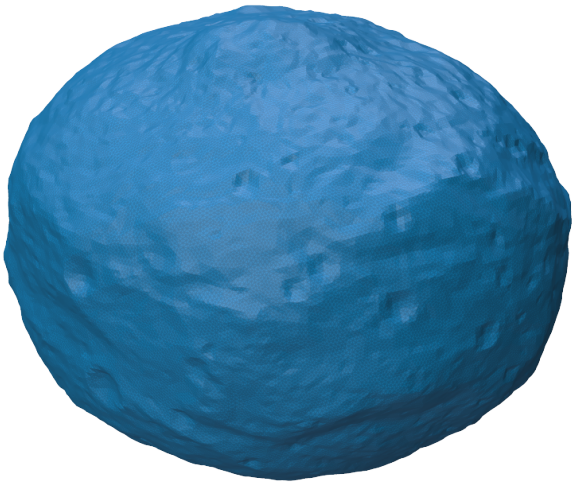
[ZJ16] Qingnan Zhou and Alec Jacobson. *Thingi10K: A Dataset of 10,000 3D-Printing Models*. arXiv:1605.04797 [cs]. July 2016. doi: 10.48550/arXiv.1605.04797. url: http://arxiv.org/abs/1605.04797.

A

Meshes

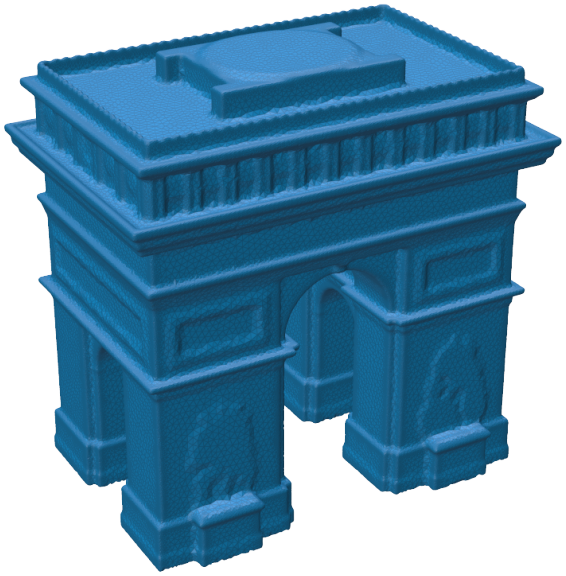
Below is a list of all meshes that have been referenced in any experiments throughout the thesis. Statistics and images of these meshes are included as well.

Name	$\lvert V\rvert$	$\lvert T\rvert$	$\lvert \mathrm{Bnd}_1\rvert$	$\lvert \mathrm{Bnd}_2\rvert$	$\lvert \mathrm{Bnd}_3\rvert$	Remeshed	Isotropic
`cube`	729	3 072	386	92	8	✗	✓
`asteroid`	135 305	511 291	97 398	0	0	✓	✗
`arc_triomphe`	147 877	530 909	112 570	547	18	✓	✗
`venus`	164 086	937 991	22 886	68	2	✓	✓
`ball`	202 244	752 178	148 966	0	0	✓	✗
`trex`	253 348	1 071 450	154 656	1 130	66	✗	✗
`devil`	316 559	1 819 700	41 318	828	44	✓	✓
`die`	809 385	4 835 763	46 069	542	0	✓	✓
`lovecraft`	1 030 046	6 126 864	67 206	2 260	81	✓	✓



`asteroid`

`venus`

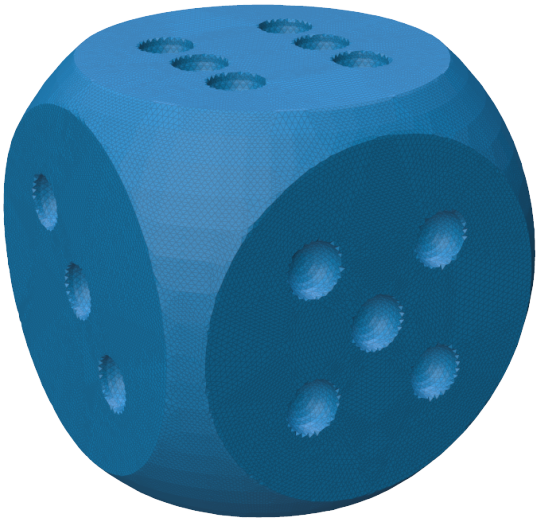


arc_triomphe

ball

trex

devil



die

lovecraft