Bachelor's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Andreas Bjelland Berg

Combining Gmsh and MRST

Developing software for more efficient grid creation in two dimensions

Bachelor's thesis in Mathematical Sciences
Supervisor: Knut-Andreas Lie
Co-supervisor: Øystein Klemetsdal, August Johansson
June 2022

NTNU
Norwegian University of
Science and Technology

Andreas Bjelland Berg

Combining Gmsh and MRST

Developing software for more efficient grid creation in two dimensions

Bachelor's thesis in Mathematical Sciences
Supervisor: Knut-Andreas Lie
Co-supervisor: Øystein Klemetsdal, August Johansson
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

NTNU

Kunnskap for en bedre verden

Abstract

Creating an accurate, discrete grid representation of a real-life, continuous domain can be challenging. This is especially true when trying to model geological formations, as they tend to be highly heterogeneous, often with crossing faults or wells. Accurate grids are, however, crucial for creating realistic simulations. This drives a need for software that can quickly and efficiently generate grids conforming to constraints of the real-life domain.

In this thesis, I discuss theory behind existing methods for creating grids, including a discussion of triangulations and their relationship to grids. I then present two existing tools for creating grids – Gmsh and MRST, and how they can be used. Finally, I introduce a new software module for combining the two – `gmsh4mrst`, and discuss how it can be used, as well as how it can be developed further in the future.

This thesis was written in collaboration with SINTEF Digital.

Oppsummering

Det kan være svært utfordrende å skape presise, nøyaktige grid-representasjoner av virkelige, kontinuerlige domener. Dette er spesielt sant når det kommer til modellering av geologiske formasjoner, ettersom de har en tendens til å være svært heterogene, ofte med kryssende forkastninger og brønner. Nøyaktige representasjoner er likevel kritiske for å skape realistiske simuleringer. Dette skaper et behov for programvare som kjapt og effektivt kan skape grids som representerer det virkelige domenet.

I denne oppgaven diskuterer jeg teorien bak eksisterende metoder for å skape grids, inkludert en diskusjon om trianguleringer og deres relasjon til grids. Jeg presenterer videre to eksisterende verktøy for å skape grids – Gmsh og MRST, og hvordan de kan brukes per dags dato. Avslutningsvis introduserer jeg et nytt program for å kombinere disse to – `gmsh4mrst`, og diskuterer hvordan dette kan brukes, samt hvordan det kan videreutvikles i fremtiden.

Denne oppgaven ble skrevet i samarbeid med SINTEF Digital.

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Code Segments

1 Introduction

The climate is changing at an ever-increasing pace. With 19 of the 20 hottest years on record happening since the turn of the millennium [15], it is adamantly clear that something must be done. One proposed part of the solution for climate change is carbon capture and storage, where CO2 is captured – whether from the air or before it is released – and stored for long periods of time [7]. One way of storing CO2 safely for longer periods of time is by utilizing subsurface, geological formations, such as depleted oil- and gas reservoirs. In order for this to actually help climate change mitigation over time, the leakage rate must less than 1% per 1000 years [17]. To ensure this can be achieved, accurate and precise simulation tools are required.

The first step of any simulation workflow starts with an accurate model of the domain. Subsurface geological formations are typically highly heterogeneous, often consisting of many layers with different physical properties [3]. This is further complicated by large-scale geological features such as faults. To store CO2 in the reservoirs, there must also be a way to create wells leading into the reservoirs, adding an additional feature our models must handle. We therefore need a modelling system that can handle heterogeneity and represent several different, often crossing, physical features, while still being computationally efficient enough to be usable.

Several methods for computer modelling have been attempted. The earliest methods consisted of Cartesian grids, extended from slices of the domain. Later methods included corner-point grids [12] – Cartesian grids with irregular polygons, which further evolved into unstructured grids, with PEBI grids being a popular choice [3]. In unstructured grids, wells are typically represented as cell centroids, while faults are traced by faces in the grid.

There exists several tools for simulating geological systems. One of these tools is the MATLAB Reservoir Simulation Toolbox (MRST). MRST includes a module for constructing PEBI grids – UPR – which supports faults through face alignment of the cells, and wells through cell centroid alignment [4]. Due to how UPR creates its background grid, the software is only capable of constructing small- to medium-sized grids, and may struggle in certain cases. The primary goal of this thesis has been to develop an extension to the UPR module, by replacing the grid refinement algorithm with mesh creation in Gmsh, thus enabling users of MRST to easily create more complex and detailed grids. We create three methods for automatically creating meshes in Gmsh, then create a MATLAB module interfacing all three methods. Special care is taken to ensure the module follows these principles, and create conforming grids.

The thesis is structured as follows. Section 2 goes through some necessary background theory, particularily focused on Voronoi diagrams and PEBI grids. Section 3 discuss existing software. The section introduce both Gmsh and MRST, and includes both limitations and examples of the two software packages. It also discusses UPR further. In Section 4, I first discuss the existing method for combining Gmsh and UPR. I then introduce a new package for combining Gmsh and MRST in Section 4.2, including a guide of installation and use, as well as an in-depth discussion of the implementation details of the new package. It naturally includes examples of how

the package can be used, as well as a number of plots and illustrations. The thesis then concludes by discussing how the package can be improved going forward in Section 5.

2 Theory

This project is a continuation on previous work by Berge [3], and most of the theory in this section is, unless otherwise stated, based on Berge [3] or Berge et al. [4].

To produce and use discrete representations of the real world, there must be a way to map the real, continuous domain to a discrete representation. This is typically done by looking at a set of points in the real domain.

Definition 2.1 (Point set)**.** A point set $P \in \mathbb{R}^n$ is a finite set of points in $\mathbb{R}^n$.

While there exist infinite point sets, we limit ourselves to finite sets in order to have a discrete representation. As point sets only include the points themselves, they cannot be used to describe the room between the points. In order to handle this, another type of representation is needed. Two points can be connected by a line segment, which describes the room between the two points in one dimension. To describe the room between three or more points in two dimensions, we introduce the concept of *tesselation*.

Definition 2.2 (Tesselation)**.** A tesselation is a way to cover a two-dimensional plane with a repeating pattern of geometrical figures, with no gap or overlaps.

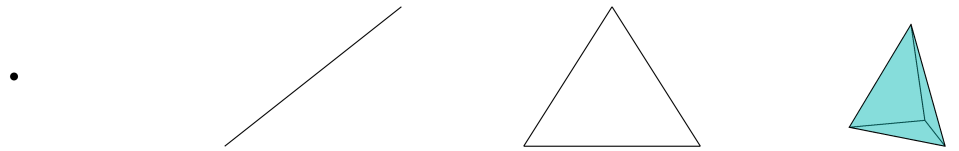
The simplest geometrical figure that can be used in a two-dimensional tesselation is the triangle, i.e., a polygon with three edges and three vertices. We can extend the idea of a triangle into *simplices*.

Definition 2.3 (Simplex)**.** A simplex is a generalization of a triangle to arbitrary dimensions. An n-dimensional simplex is the simplest geometrical figure in an n dimensional space.

Using simplices enables the inclusion of data about our domain that is not directly represented in the points. In geological modelling, this primarily includes physical properties such as subsurface faults, wells and other geological features, which are necessary for accurate simulations.

As discrete representations of objects in physical space are limited to at most three spatial dimensions, we only need to consider the first four simplices (Figure 1):

- A 0-simplex is a point.
- A 1-simplex is a line.
- A 2-simplex is a triangle.



(a) 0-simplex (b) 1-simplex (c) 2-simplex (d) 3-simplex

Figure 1: Illustration of the first 4 simplices.

- A 3-simplex is a tetrahedron.

In general, a k-simplex is a k-dimensional geometric object consisting of the *convex hull* of its $k+1$ vertices.

Definition 2.4 (Convex hull)**.** The convex hull of a point set P is the minimal convex set containing P.

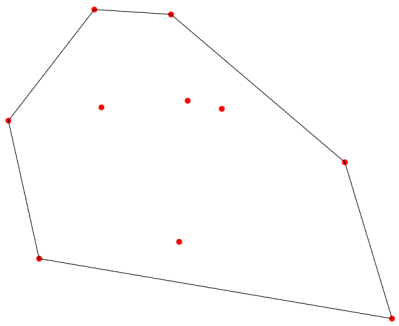


Figure 2: Example of a convex hull for a set of 10 randomly generated points in $\mathbb{R}^2$.

The result of converting a point set into a set of simplices is called a *triangulation*.

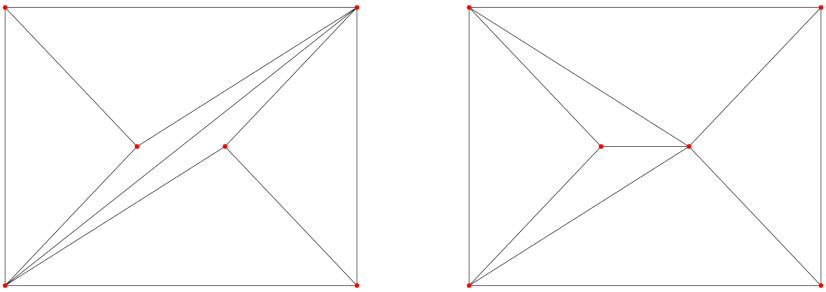
Definition 2.5 (Triangulation)**.** A triangulation T of a point set P is the set of simplices T such that:

- the set of vertices in T equals P, and
- the union of all simplices in T equals the convex hull of P, i.e., the union of all simplices in T makes up the minimal convex set containing P.

For a given point set, one can generally create multiple valid triangulations. Figure 3 shows two triangulations for the same point set P.

2.1 Delaunay triangulation

Delaunay triangulations were first introduced by Delaunay [5] in 1943. Delaunay triangulations maximize the smallest angle of the simplices in the triangulation and



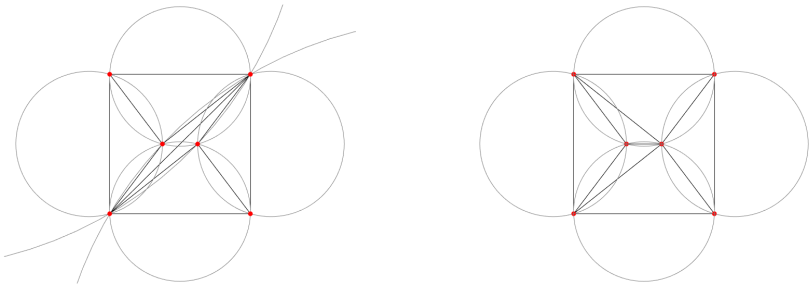
(a) Random triangulation (b) Delaunay triangulation

Figure 3: Example of two triangulations for a set of points $P \in \mathbb{R}^2$.

therefore produce representations that work nicely in grid-based applications. The Delaunay triangulation of a point set P is defined as follows:

Definition 2.6. The T triangulation of the point set P is a Delaunay triangulation if no vertices in P are inside the *circumcircle* of any simplex in T.

Definition 2.7 (Circumcircle)**.** A circumcircle of a polygon is a circle passing through all vertices of the polygon. The circumcircles of a triangulation is the set of circumcircles of all simplices in the triangulation.



(a) Circumcircles of Figure 3a (b) Circumcircles of Figure 3b

Figure 4: Example of circumcircles for the two triangulations in Figure 3. Note that Figure 4a has a been cropped so that two of the circumcircles are only shown as arcs.

In Figure 4b, no vertices of P are inside any of the plotted circumcircles. This shows that the triangulation in Figure 3b is a Delaunay triangulation. In Figure 4a, two vertices are inside each of the two large circumcircles created from the centermost, obtuse triangles. This shows that the triangulation in Figure 3a is not Delaunay.

Delaunay triangulations are unique as long as at most three vertices are on the same circumcircle [3, Theorem 2.1]. In cases where this is not true, such as where four vertices make up a cyclic quadrilateral, there exist several valid Delaunay triangulations for the same point set P. This is illustrated in Figure 5, where the two center vertices make up cyclic quadrilaterals with both the top and bottom pair of vertices. This enables four valid Delaunay triangulations of the point set.

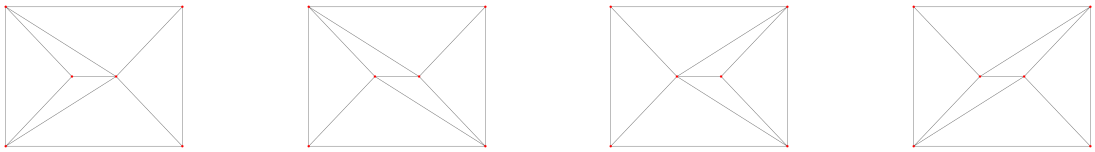
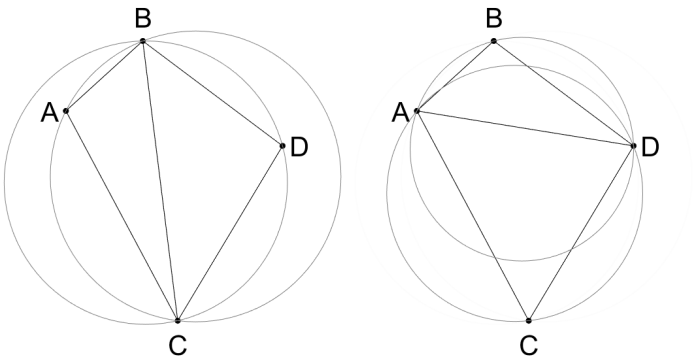


Figure 5: Non-uniqueness of Delaunay triangulations. Two cyclic quadrilaterals enable four valid Delaunay triangulations.

Several algorithms exist for generating Delaunay triangulations. The simplest of these is a flip algorithm. This algorithm is based on the flipping technique: Observe any two triangles ABC and BCD that share a common side BC, where the triangles do not meet the Delaunay condition. These triangles can be converted to two triangles that do meet the Delaunay condition by flipping them – swapping the common side BC for a new common side AD, thus creating new triangles ABD and ACD. An example of this technique is shown in Figure 6. The algorithm is simple once this technique is known: To create a Delanunay triangulation, start with any triangulation T, then flip edges of any non-conforming triangles until every triangle meet the Delaunay condition.



(a) Initial triangulation. (b) After flipping.

Figure 6: Illustration of flipping a common side to create Delaunay triangulations. We start with an initial triangulation ABC and BCD in (a). These do meet the Delaunay condition, as shown from the gray circles. We therefore flip the common side and get the new triangulation ABD and ACD shown in (b).

2.2 Voronoi diagrams

A *Voronoi diagram* is a form of gridding formally introduced by Voronoi [19], which is closely related to Delaunay triangulations.

Definition 2.8 (Voronoi diagram)**.** Let P be a point set in $\mathbb{R}^n$ consisting of points $p_i \in P$ for $i = 1, \ldots, m$. Each point p_i is called a *site*. A point $x \in \mathbb{R}^n$ belongs to the *Voronoi cell* v_i associated with site p_i if the distance between p_i and x is equal to the minimum distance from x to any site $p \in P$. Formally, we can define this as

$$v_i = \{x : x \in \mathbb{R}^n \text{ s.t. } |x - p_i| \leq |x - p| \quad \forall p \in P\}\,. \tag{1}$$

The set of all Voronoi cells v_i is called a *Voronoi diagram* or *Voronoi grid*.

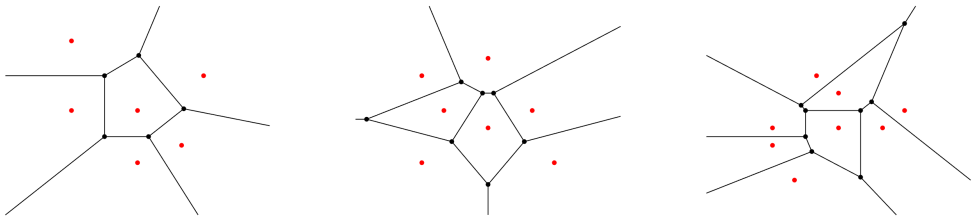


Figure 7: Example of Voronoi diagrams for different point sets, marked in red.

The intersection between two Voronoi cells v_i and v_j is called a face, and is denoted by $v_{i,j}$. The face is made up of all points $x \in \mathbb{R}^n$ that are the same distance from p_i and p_j, and where this distance is the same as the minimum distance from x to any $p \in P$. We can define this formally as

$$v_{i,j} = \left\{ x : x \in \mathbb{R}^d \text{ s.t. } |x - p_i| = |x - p_j| \leq |x - p| \quad \forall p \in P \right\}. \tag{2}$$

We can, in general, say that a k-face of a Voronoi diagram is the k-dimensional intersection between Voronoi cells. We can define this formally as

$$v_{i,\dots,j} = \left\{ x : x \in \mathbb{R}^d \text{ s.t. } |x - p_i| = \cdots = |x - p_j| \leq |x - p| \quad \forall p \in P \right\}. \tag{3}$$

When dealing with a two-dimensional domain, i.e., $P \in \mathbb{R}^2$, there are 1-dimensional faces as edges where two cells meet and 0-dimensional faces as vertices where two or more edges intersect. In Figure 7, the 1-dimensional faces are marked by lines, and the 0-dimensional faces are marked by black dots. When $P \in \mathbb{R}^3$, we have 2-dimensional faces as planes where two 3-dimensional cells meet, 1-dimensional faces as edges where two planes intersect, and 0-dimensional faces as vertices where two or more edges intersect.

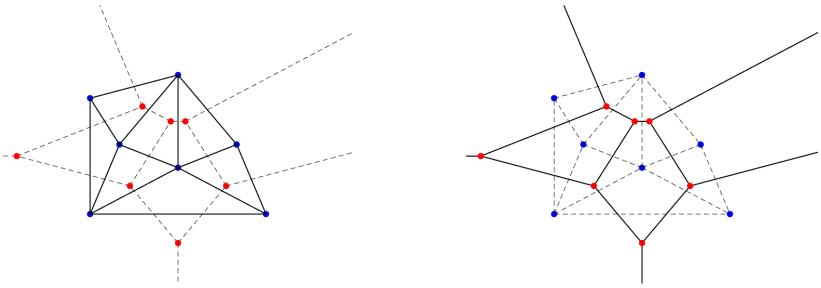
2.2.1 Relationship between Delaunay and Voronoi

For two-dimensional domains, it follows from Equation 3, that 0-dimensional faces, i.e., Voronoi vertices, are points where the distance to three Voronoi sites are equal and smaller than the distance to any other sites. In other words, sites p_i, p_j and $p_k \in P$ create a vertex $v_{i,j,k}$ when the open circle intersecting all sites is empty of any sites in P. The vertex is placed in the center of the open circle. The same holds true in $\mathbb{R}^3$, where four sites have a 0-dimensional face (vertex) if and only if the open ball intersecting all four sites is empty of any sites in P.

This appears to be very similar to the definition of Delaunay triangulation in Definition 2.6. In fact, Delaunay triangulations and Voronoi grids are considered dualities.

Let T be a Delaunay triangulation and V be a Voronoi diagram, both on the same point set. Then each node in T is associated with a cell in V, and vice versa. More specifically, the centers of the circumcircles used to prove T is Delaunay are vertices

in the Voronoi diagram, and if there is an edge in T between two sites, then the sites share a face in V. Likewise, the sites in the Voronoi diagram are vertices in the Delaunay triangulation. This is illustrated in Figure 8: sites are marked by blue dots and the circumcenters of the Delaunay triangulation are marked by red dots.



(a) Delaunay triangulation (b) Voronoi diagram

Figure 8: Delaunay triangulation and Voronoi diagram for the same point set.

2.2.2 Clipped Voronoi diagrams

One important, but in our case unwanted, property of Voronoi diagrams is that they extend to infinity. This is visible in both Figure 7 and Figure 8. As we are interested in mapping real-world objects to discrete domains that can be used for simulations, we need a way to fit the domain of our model to the boundary of the physical domain we are modelling. To handle this, we can modify the definition of Voronoi diagrams slightly, introducing *clipped Voronoi diagrams*.

Definition 2.9 (Clipped Voronoi diagram)**.** Let P be a point set in a bounded domain $\mathcal{D} \subset \mathbb{R}^n$. We define a clipped Voronoi diagram as the set of all cells v_i, where

$$v_i = \{x : x \in \mathcal{D} \subset \mathbb{R}^n \text{ s.t. } |x - p_i| \leq |x - p| \quad \forall p \in P\}\,. \tag{4}$$

While there is little difference between clipped and unclipped Voronoi diagrams, limiting the domain is beneficial for simulations and computer representation. Note that clipping a Voronoi diagram means the border cells are no longer strictly Voronoi, but this tends to have little impact on the overall quality of the cells.

2.2.3 Centroidal Voronoi diagrams

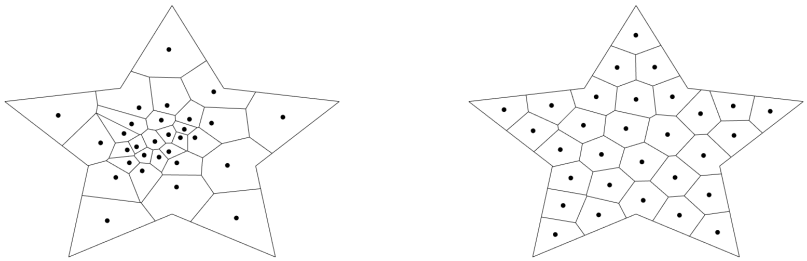
The properties of a Voronoi diagram depend on the distribution of its sites. One especially interesting case happens when the sites and centers of each cell align. We use the mass center of the Voronoi cells to calculate the center of each cell. The mass center of a Voronoi cell is given as

$$c_i = \frac{\int_{V_i} y\rho(y)dy}{\int_{V_i} \rho(y)dy}, \tag{5}$$

where V_i is the volume of the cell and $\rho(y)$ is a given mass density function of the cell. If the mass centers and sites in a Voronoi diagram line up, we have a special subset of Voronoi diagrams called a *centroidal Voronoi diagram (CVD)*.

Definition 2.10 (Centroidal Voronoi diagram)**.** Let P be a set of sites in a Voronoi diagram $\mathcal{V}$. The diagram is a *centroidal Voronoi diagram* if $c_i = p_i$ for all $p_i \in P$.

As a result of this alignment, centroidal Voronoi diagrams typically have highly regular and evenly sized cells, which is why they are often called *optimal Voronoi diagrams* [3]. This property is shown in Figure 9.



(a) Random Voronoi diagram (b) Centroidal Voronoi diagram

Figure 9: Comparison of centroidal and non-centroidal Voronoi diagrams

Several methods for generating CVDs exist, and one of the simplest is by using fixed-point iteration. The method, called Lloyd's method, is shown in Algorithm 1.

Algorithm 1: Lloyd's method

```
Make an initial set of sites
Until convergence:
    Construct Voronoi diagram of sites
    Compute centroids of the Voronoi diagram
    Use the centroids as new sites
```

While this method is simple and easy to implement, its linear convergence limits its usability. A quasi-Newton method was introduced by Liu et al. [14], using the CVD energy function given by

$$F(x) = \sum_{i=1}^{m} \int_{V_i} \rho(\mathbf{y}) |\mathbf{y} - c_i|^2 d\mathbf{y}, \tag{6}$$

which is minimized when $p_i = c_i$. This method is both faster and more robust than Lloyd's method [14].

2.2.4 Optimal Delaunay triangulation

Another method for generating nice Voronoi diagrams is by exploiting the duality between Delaunay triangulations and Voronoi diagrams and using a force based

algorithm. The optimal Delaunay triangulation, introduced by Persson and Strang [16], associates each Delaunay vertex with a joint and each edge with a spring. By defining an element size function $h(x, y)$, we attain the uncompressed length of the springs. We then let the forces follow Hooke's law, but only for repulsive forces, i.e., the force f from a spring with length l and uncompressed length l_0 is given by

$$f(l, l_0) = \begin{cases} l_0 - l, & l < l_0, \\ 0, & l \geq l_0. \end{cases} \tag{7}$$

The total force on a point, $F(p_i)$, is simply the sum of the force from all springs connected to p_i. We have two special types of points to consider. Points along the border must avoid getting pushed out of the domain. To do so, we add an external force perpendicular to the border pointing inwards, balancing the internal forces. We can also have fixed joints. These are not allowed to move, no matter the applied force. The optimization loop, then, is simple:

Algorithm 2: Optimal Delaunay triangulation

```
Make an initial set of sites
Until convergence:
    Calculate Delaunay triangulation of sites
    For each edge in the triangulation:
        Calculate force of the associated spring
    For each non-fixed vertex in the triangulation:
        Calculate the sum of forces on the vertex, including
        external forces
        Move vertex a set step length along the sum of forces
```

This method is the primary method for generating background sites in UPR, and is further discussed in Section 3.1.1. An example of applying the optimal Delaunay triangulation algorithm on a point set is shown in Figure 10.

2.3 PEBI grids

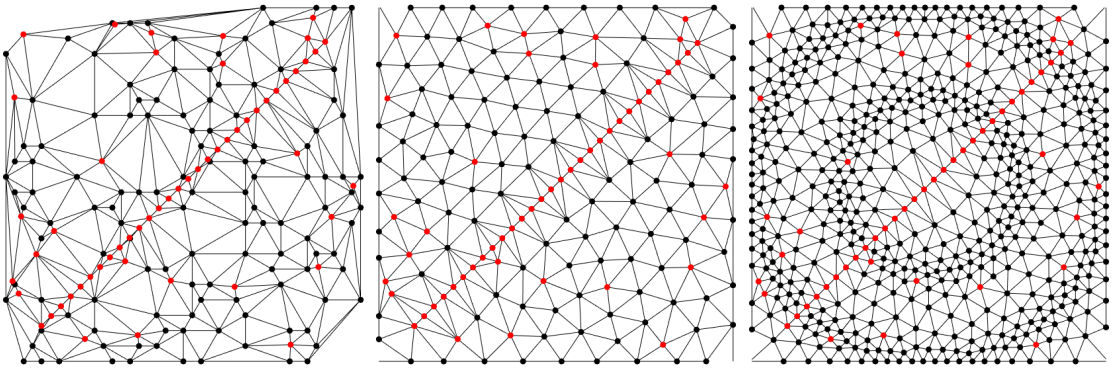
Within reservoir modelling, the term PEBI grids – PErpendicular BIsector grids – is often used instead of Voronoi grids. Berge [3] operates with the following definition:

Definition 2.11 (PEBI grid)**.** Let $P \in \mathbb{R}^n$ be a finite point set and let $\mathcal{B}$ be the PEBI grid of P. For each site $p_i \in P$, a cell b_i is generated by the following algorithm:

1. For each other site in $p_j \in P$, create the "perpendicular bisector plane" of p_i and p_j, i.e., the plane that splits P in two subspaces perpendicular to the line from p_i to p_j, in the middle of the two points.
2. Let $H(p_i, p_j)$ be the subspace that contains p_i.

The cell b_i is then defined as the intersection of all these subspaces, i.e.,

$$b_i = \bigcap_{p_j \in P \setminus p_i} H(p_i, p_j). \tag{8}$$



(a) Before ODT (b) Uniform h (c) Non-uniform h

Figure 10: Example of applying the optimal Delaunay algorithm on a point set. The initial point set is shown in (a). The result of applying the optimal Delaunay algorithm with a uniform element size function is shown in (b). The result of applying the optimal Delaunay algorithm with a non-uniform element size function is shown in (c). The red points are fixed.

This process is visualized in Figure 11, in which p_i is the site marked by a red dot, with blue dots representing the other $p_j \in P$. The dashed lines are the lines from p_i to each p_j, with the solid black lines representing the two dimensional perpendicular bisector planes between p_i and each p_j. The light-red area is the cell b_i.

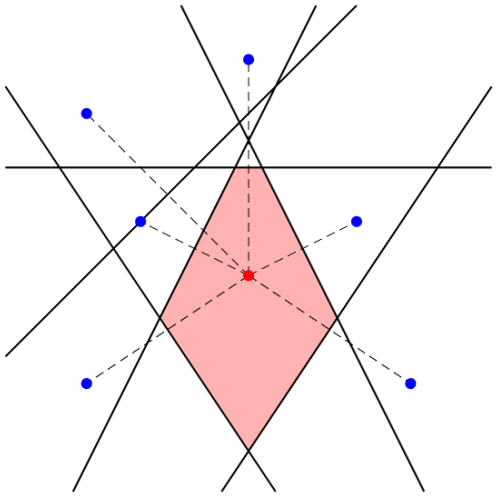


Figure 11: Visualization of PEBI generation.

It is relatively easy to show that PEBI grids and Voronoi diagrams are the same. Let P be a finite point set in a domain $\mathcal{D}$, with $\mathcal{B}$ as a PEBI grid of P, and consider a randomly selected point $x \in \mathcal{D}$. Assume that x belongs to PEBI cell b_i. This means it is part of all subspaces $H(p_i, p_j)$, and thus must be at least as close to p_i as any other $p_j \in P$. As this holds for all j, it follows from Equation 4 that $x \in v_i$.

Now consider the Voronoi diagram of P. As every point in $\mathcal{D}$ belongs to at least one Voronoi cell, assume a randomly selected point $x \in \mathcal{D}$ belongs to cell v_i in the

Voronoi diagram of P. We know from Equation 4 that x is at least as close to v_i as any other v_j. This means that x will be in the subspace $H(p_i, p_j)$ for all j, meaning it must be in the intersection of all $H(p_i, p_j)$. It thus follows from Definition 2.11 that $x \in b_i$.

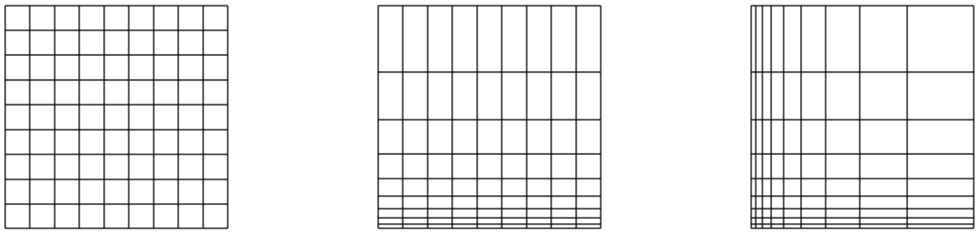
As PEBI is widely used in the literature, as well as in the software used for this thesis, I will use the term PEBI grids, rather than Voronoi diagrams, going forward.

2.4 Transfinite grids

One way of creating structured grids is by defining a continuous function f around the constraint of the grid, then interpolating the constraint function to create a function on the whole domain [6]. This creates a *transfinite grid*.

Definition 2.12 (Transfinite grid)**.** A transfinite grid is defined by interpolating a function f defined around the constraint of the grid into the domain of the grid.

The simplest form of transfinite grids are defined by a linear function f around the constraint, creating a rectangular, structured grid. Three examples of transfinite grids are shown in Figure 12.



(a) Linear f (b) Exponential y (c) Exponential x and y

Figure 12: Examples of transfinite grids. In (a), f is linear along both axes. In (b), f is exponential along the y-axis, but linear along the x-axis. In (c), f is exponential along both axes.

3 Software

As there exists a vast range of solutions for creating grids to represent real domains, learning, using and discussing all of them are way beyond the scope of this thesis. I will instead focus on two tools – MRST in Section 3.1 and Gmsh in Section 3.2 – as well as ways to combine the two tools in Section 4.

3.1 MRST

The MATLAB Reservoir Simulation Toolbox (MRST) is an open-source toolbox for reservoir simulation, developed by the Computational Geosciences group at SINTEF

Digital. Originally aimed at the study of discretization and flow solvers, the project has since expanded and currently offers much of the same functionality that can be found in commercial reservoir simulators [13]. Its main target is still research, with the primary focus being on rapidly developing and demonstrating contemporary methods and concepts [18].

To keep its extensive set of features maintainable, MRST is organized with a set of core functionality, as well as several optional add-on modules. The core includes methods for handling grids, data, and basic drive mechanisms such as gravity, sources, and wells, as well as an implementation of automatic differentiation. The add-on modules includes tools for discretization, solvers for incompressible flow, simulators based on automatic differentiation, specialized computational methods aimed at solving concrete problems, several utility modules, as well as tools that can be used to aid the modeling of the reservoirs. The latter group contains the UPR module.

3.1.1 The UPR Module

The "Unstructured PEBI-grids for Reservoirs" (UPR) module was developed by Berge [3] as part of his 2016 master's thesis. The module is designed to generate PEBI grids conforming to line and surface constraints (typically: well trajectories and fault surfaces) and can handle several challenging cases, such as multiple faults intersecting, intersections at sharp angles, and intersections between wells and faults. The module also contains methods to generate such grids in three dimensions, but I will focus on 2D applications.

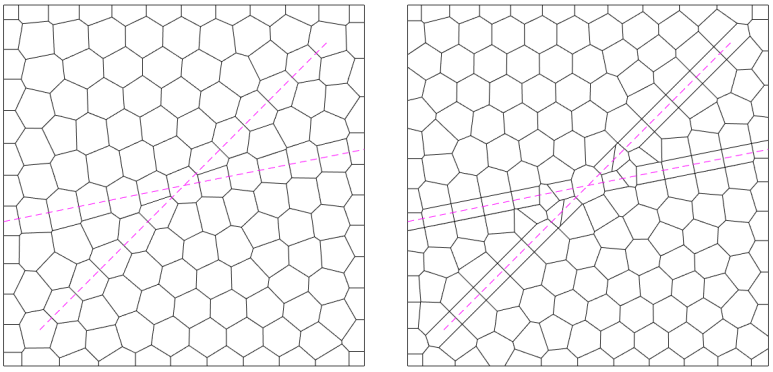
Well sites: UPR represents wells as cell centroids and handles the base case, where the well follows a single curve, by simply placing a set of well sites along that curve. The distance between two consecutive sites is not necessarily constant along the curve, but care is taken such that the distribution satisfies the *well condition* [3].

Definition 3.1 (Well condition)**.** Let p_1 and p_2 be two consecutive well sites. The *well condition* is satisfied if there exists a circle intersecting both points p_1 and p_2 that does not contain any other sites.

It follows from Definition 2.6 that the edge between two consecutive well sites makes up an edge in a Delaunay triangulation if the well condition is satisfied, ensuring the two wells are neighbors in the PEBI grid. To simplify calculations, UPR uses a circle centered on the midpoint between two consecutive well sites to check the well condition.

Extra care is also taken when two wells intersect, as handling each well independently may lead to consecutive sites not connecting, or low-quality cells. To handle this, the intersecting wells are split and a well site is placed at the intersection. This well site is then shared among all intersecting at the point, i.e., all sites that now start or end at the intersection.

UPR also allows protection sites around the well sites to ensure regular well shapes and control the radius for the well cells. This is done by creating a pair of protection cells for each well site. These protection cells are placed normal to the well path, one on each side of the well site, with a distance equal to the target well radius. No protection sites are used for well intersection sites. An example of this is shown in Figure 13.



(a) Well without protection. (b) Well with protection.

Figure 13: Example of protective sites in UPR. The magenta lines represent wells.

Fault sites: Faults are represented by cell edges and are created by tracing faults with two lines of fault sites – one on each side of the fault lines. UPR does this by creating a set of circles along each fault, then creating fault sites where these circles intersect. It starts by placing a set of points, $C = \{c_i\}$, along the fault, with a distance between points given by a density function $d_i = \rho(c_i, c_{i+1})$. To create two fault points, two consecutive circles must intersect in two places. This creates the following distance limitations:

$$d_i \leq R_i + R_{i+1}, \tag{9}$$

$$d_i \geq |R_i - R_{i+1}|, \tag{10}$$

where R_i is the radius of the circle centered at c_i. In UPR, the radius of each circle is set to be proportional to the average of the distance to the circles on either side, i.e.,

$$R_i = c_f \frac{d_i + d_{i-1}}{2}, \tag{11}$$

where c_f is a constant circle factor, determining the distance between the fault and the fault sites. Fault sites are then placed where two circles intersect, satisfying the *fault condition*.

Definition 3.2 (Fault condition)**.** Let p_1 and p_2 be two fault sites placed at the intersection of the same pair of fault circles. The *fault condition* is satisfied if the interior of the two fault circles does not contain any sites.

The fault condition is enough to prove that the fault is traced by a face in the PEBI grid. Let c_1 and c_2 be the circles creating p_1 and p_2, and let c_p be a point on the open line segment between c_1 and c_2. Let c be a closed circle intersecting p_1 and p_2 with

centroid c_p. This circle is then an element in a subset of $\{c_1, c_2\}$, thus containing no other sites than p_1 and p_2. As c intersects both p_1 and p_2, we have that

$$|c_p - p_1| = |c_p - p_2| < |c_p - p|$$

for all other sites p. From Definition 2, c_p must therefore be on the PEBI face v_{p_1,p_2}.

As with wells, extra care must be taken when fault lines intersect, as fault sites from different faults may disrupt the fault conditions. To handle these cases, faults are first split up at intersections, such that they may only appear at the start or end of each fault line segment. A fault circle is then placed at the intersection, and is shared by all fault lines starting or ending at the intersection. All other circles are placed like normal. For each circle neighboring an intersection circle, there are three potential options:

Do nothing: If the interior of the circle does not contain any other sites, the circle is left as is.

Shrink the circle: If the interior of the circle contains any other fault sites, the radius of the circle is shrunk. Let p_i be the site inside the circle. The circle, as well as the circle generating p_i, is shrunk down such that they both intersect the intersection circle at the same point. If multiple p_is exist, the smallest radius is chosen.

Combine circles: If the radius is shrunk down enough to violate the condition in Equation 9, then it is simply combined with the conflicting circle. The result is a single fault site located in the middle of the two previous fault sites. The new circle is considered to be an intersection, and the process repeats for its neighbors.

Intersections between wells and faults: Special care is also taken when wells and faults intersect. UPR handles this, like with the previous types of intersections, by first splitting the well and fault at the intersection. The first fault circle of the fault starting at the intersection, and the last fault circle of the fault ending at the intersection, are both placed half a step from the intersecting point. The two fault sites created from these circles are then label well sites, making up the end sites of the well segments ending at the intersection.

Other sites: UPR employs three methods for generating the remaining PEBI sites. The first and simplest method is to just distribute the sites in a Cartesian grid. After all sites have been placed, all background sites violating well and fault conditions are removed. In addition to this, each well and fault site gets a grid size, defined for well sites as the distance between two consecutive well sites, and for fault sites as the distance between two sites generated by the same two fault circles. Any background sites within the grid size of a well or fault site are also removed, creating a structured PEBI grid conforming to both wells and faults.

To refine structured grids, especially for cells near constraints, UPR employs a standard multilevel quad-tree local grid refinement [3, pp.49]. The method refines a cell

by splitting it in four, connecting the opposing edges to each other. To refine a PEBI cell, the site creating the cell is replaced by four sites, one in each quadrant of the cell.

The primary way of creating unstructured grids in UPR is by using the force-based method discussed in Section 2.2.4. To refine cells near constraints, UPR uses the following element size function [3, Equation 4.2]:

$$h_r(p) = \min\left[h_{\max}, h_{\min}\exp\left(\frac{d(p,W)}{\epsilon}\right)\right], \tag{12}$$

where $h_{\max}$ is the desired grid size far from constraints, $h_{\min}$ is the desired grid size close to constraints, and the distance $d(p,W)$ is the closest distance from the point p to the set of constraint sites W. The parameter ϵ controls the transition – a small ϵ means the transition from $h_{\min}$ to $h_{\max}$ happens quickly, while a large ϵ leads to a wider transition zone. All constraint sites are kept fixed. UPR relies on the implementation from a package called `distmesh` [16]. An example of this algorithm can be seen in Figure 10.

The last method UPR implements is by using the CVD energy function to optimize the Voronoi diagram, as discussed in Section 2.2.3. Again, fault and well sites are considered fixed. The CVD energy function is then minimized, but without moving any of the fault or well sites.

A generalized overview of UPR's algorithm for unstructured gridding [3, pp.51] is shown in Algorithm 3.

Algorithm 3: UPR Unstructured Gridding

```
Create faults and wells
    Place a set of well sites along each well path, according to the
    well cell density function
    Place a set of circles centered along the faults according to
    the fault cell density function. Place fault sites at circle
    intersections
    If two or more wells intersect:
        Place a well site at the intersection
    If two or more faults intersect:
        Place a circle at the intersection
        Adjust the neighboring circles as needed
        Place fault sites at circle intersections as before
    If a well and a fault intersect:
        Place a circle on each side of the intersection, half a step
        away. The two sites created by these circles are considered
        well sites.
Create a set of reservoir sites in the domain
Create other sites, such as refinements
Remove all reservoir sites that violate fault or well condition
Remove all reservoir sites closer to a fault or well site than their
respective grid size.
```

Limitations: The key limitation of UPR is its site placement algorithm. While `distmesh` tends to produce uniform grids closely aligning with the given constraints, it is often slow, especially for grids with fine details. As these cases may be present in practical scenarios, such as when modelling small wells in relatively large domains, it would be beneficial to implement a method for handling these cases efficiently and accurately.

3.2 Gmsh

Gmsh is an open-source grid generator, CAD engine, and post-processor developed by Geuzaine and Remacle [9]. Supporting both a graphical user interface, a dedicated scripting language, and APIs in C, C++, Julia, and Python, Gmsh has become one of the most popular finite element mesh generators in the world [8].

First released in 1998, Gmsh is built on a simple philosophy – to be a fast, light, and user-friendly mesh generator [9]. Gmsh is designed to generate large meshes in little time, claiming they can generate "larger than average" meshes in less than a minute on a standard computer, while being able to visualize the mesh in real-time. The program is designed to have as little footprint as possible, with easy installation and a comprehensible code base. Finally, Gmsh is designed such that a novice user is able to quickly create simple meshes, with an extensible and well-documented API.

3.2.1 Gmsh modules

Gmsh is structured in four separate modules. *The Geometry module* is how Gmsh creates new physical geometry. Gmsh was originally designed with a limited CAD engine, primarily suited for simple structures [9]. However, as the user base of Gmsh grew, so did the need for better CAD support, allowing Gmsh to mesh industrial CAD models. To handle this, Gmsh's geometry module is built on a set of four abstract data structures. These are:

- vertices, G_i^0, of dimension 0,
- edges, G_i^1, of dimension 1,
- faces, G_i^2, of dimension 2, and
- regions, G_i^3, of dimension 3.

Combining these structures into a boundary representation, Gmsh is able to accurately define any 3D model. The model is built as a list of entities, with each entity possibly consisting of multiple data structures.

The Meshing module handles, naturally, the meshing done in Gmsh. The meshing is based on the target mesh size of all points (x, y, z) in the domain being given by the mesh size field function $\delta(x, y, z)$. This function can be defined by:

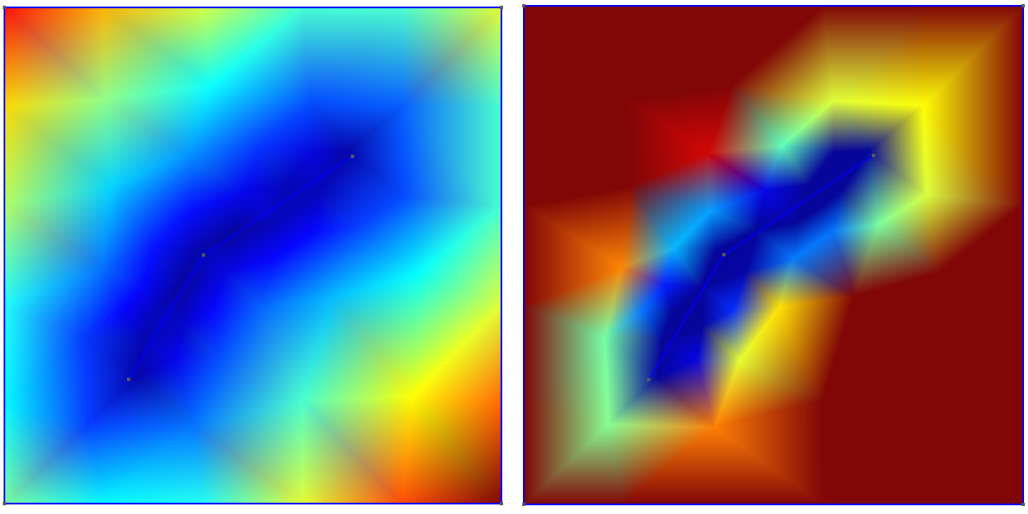
- giving vertices a target mesh size, then interpolating along edges,
- giving model edges a mesh gradient,
- defining mesh sizes in another mesh, or
- setting default mesh sizes that adapt to the curvature of model entities.

These fields are highly flexible, and can depend on several factors. Examples of these fields include:

Distance fields: Distance fields are used to calculate the distance from a set of lines and points. An illustration of the output from a distance field is shown in Figure 14a.

Threshold fields: Threshold fields are used to set values depending on another variable, often the distance from a distance field. The threshold field has four parameters: minimum and maximum for both distance and size. The field outputs a stepped curve – when the distance is lower than the minimum distance, the minimum size is returned. Between the minimum and maximum distance, the size scales linearly, and when the distance is above the maximum, the size is equal to the maximum size. An illustration of the output from a threshold field is shown in Figure 14b. An illustration of the threshold curve is shown in Figure 15.

Minimum fields: Minimum fields simply returns the minimum of a set of fields. It is often used to get the smallest cell size from a set of size fields.



(a) Distance field (b) Threshold field

Figure 14: Output from Gmsh distance and threshold fields. The distance field calculates the distance from a central line, and the threshold field uses the distance field as input. The blue color indicates a low output, i.e. the minimum distance and size, respectively, and the red indicates a high output.

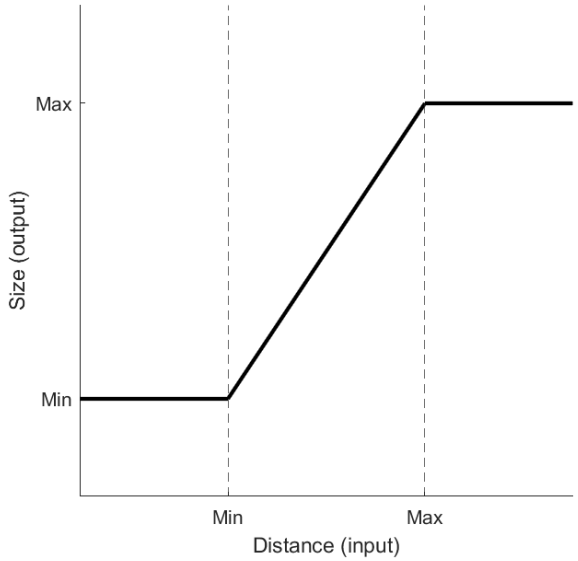


Figure 15: Illustration of the Gmsh threshold field.

The meshing module provides several algorithm for meshing 2D domains. An overview of these, as well as a short description, is given below. All algorithms start with a Delaunay mesh containing all points of an originally created 1D mesh.

MeshAdapt: Splits long edges, collapses short edges and swaps edges if that creates a better geometrical configuration. MeshAdapt is the most robust algorithm for very complex curved surfaces and is run automatically if Delaunay or Frontal-Delaunay fails [10].

Delaunay: New points are inserted at the center of the circumcircle with the largest radius. Delaunay is the fastest for very large meshes of plane surfaces and usually handles complex mesh size fields well [10].

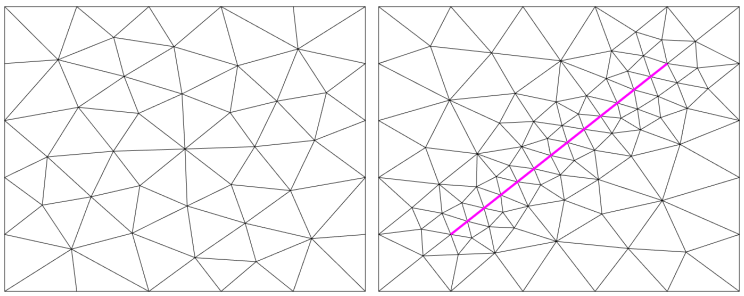
Frontal-Delaunay: A form of Delaunay computing both points and connections at the same time. Typically produces high element quality [10].

Frontal-Delaunay for Quads: Experimental variant of Frontal-Delaunay. Tries to create right-angle triangles that can be combined into rectangles [10].

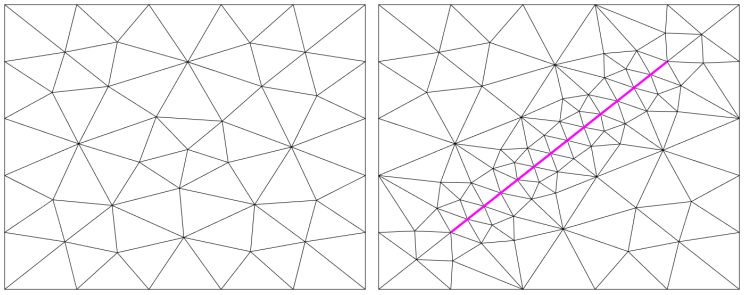
BAMG: Experimental algorithm creating anisotropic triangles, i.e., triangles with different properties in different directions, such as triangles that are long and narrow [10].

Examples of the meshes created by these algorithms can be seen in Figure 16. While Gmsh provides several different algorithms, they all produce some type of unstructured triangulation. To convert these into quadrilateral grids, Delaunay offers four recombination algorithms. Two of these – Blossom and Blossom full-quad – are based on a minimum-cost algorithm [9]. There is very little available information about the last two: Simple and Simple full-quad.

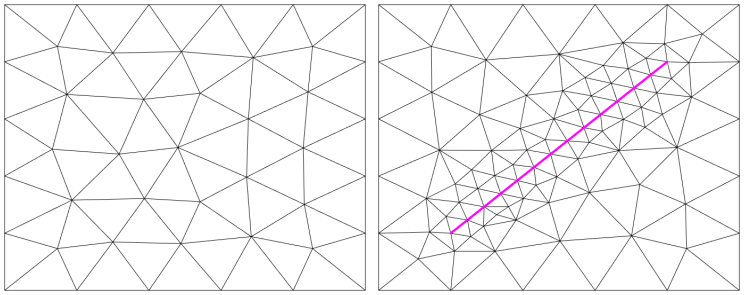
Gmsh is primarily aimed at creating unstructured meshes. It does, however, provide some methods for generating structured meshes, including support for transfinite



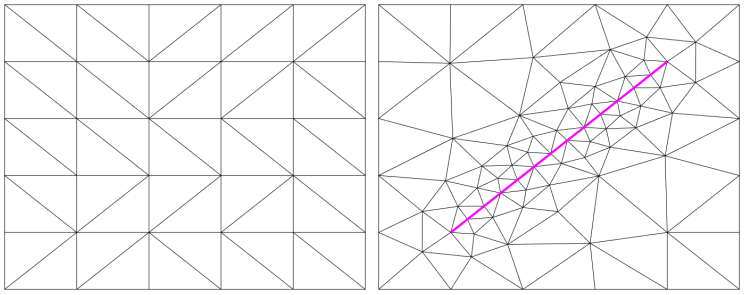
(a) MeshAdapt



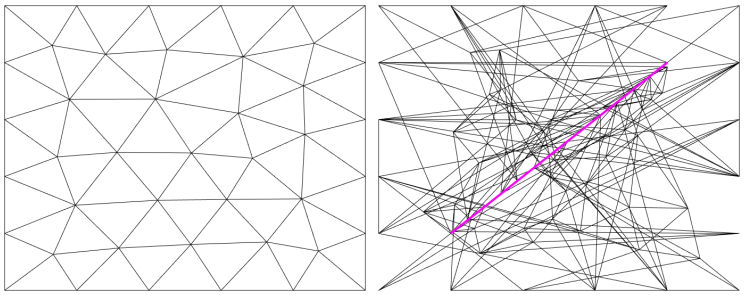
(b) Delaunay



(c) Frontal-Delaunay



(d) Frontal-Delaunay for Quads



(e) BAMG

Figure 16: Comparison of the Gmsh meshing algorithms. The left set of meshes are empty, while the right set have an embedded face constraint marked in magenta. Note that BAMG fails to create a quality mesh with an embedded line.

meshes [9]. While these require some manual work to work nicely, they provide a way to generate structured, controlled meshes. An example of transfinite meshes is shown in Figure 19, with code in Code Segment 3.

The Solver module provides two ways to connect Gmsh to external finite element solvers. The most straight-forward way is by directly using the Gmsh API to load geometry, meshes, and data [9]. This method requires a significant programming investment from the solver developers, as the connection between the solver and the API must be created.

To maintain the goal of being a user friendly generator, Gmsh provides a Gmsh interface through Unix or TCP/IP sockets. This lets a user launch solvers directly from Gmsh, and requires few adaptations of the solver software. Solvers are outside the scope of this thesis.

The Post-Processing module enables loading and display of data views, alongside visualizations of the geometry and mesh. Views are collections of values, both scalar and tensors, where scalar fields can be visualized as iso-surfaces or color maps and vector fields as arrows or displacement maps. Post-processing is outside the scope of this thesis.

3.2.2 Examples of Gmsh

While Gmsh is designed to be user friendly, the scope of features and interfaces make the provided APIs somewhat complex. This section will present some basic examples of Gmsh programs, as well as their outputs. Source code for the examples is shown in Appendix A.

Simple square mesh: While a square mesh is simple to envision, the creation of one requires a bit of code. Code Segment 1 shows how this can be implemented, using Gmsh's Python API. The output is shown in Figure 17.

Adjusting the mesh size field: Gmsh offers two ways to adjust the local resolution of the produced mesh. The first option, as shown in Figure 16, is to embed lines in the mesh, making Gmsh use the lines as faces in the mesh. The second method is to adjust the mesh size field function.

An example of the latter method is shown in Code Segment 2, with output in Figure 18. In the example, we use a line as basis for a threshold field that scales the mesh size based on distance from the line. We also use a point as basis for a math field that scales the mesh size based on the square of the distance from the point. We finally combine the fields by using the minimum mesh size. Note that neither the point nor the line are embedded in the mesh, meaning that there is no guarantee that the faces of the mesh align with either. The example also shows how Gmsh can connect to multiple CAD kernels; in this case using the OpenCASCADE kernel [10].

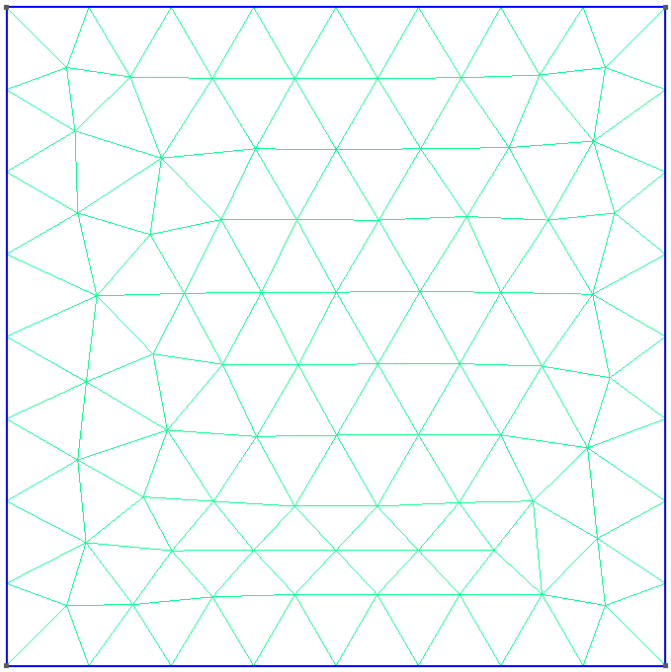


Figure 17: Simple square grid generated in Gmsh. Output from Code Segment 1.

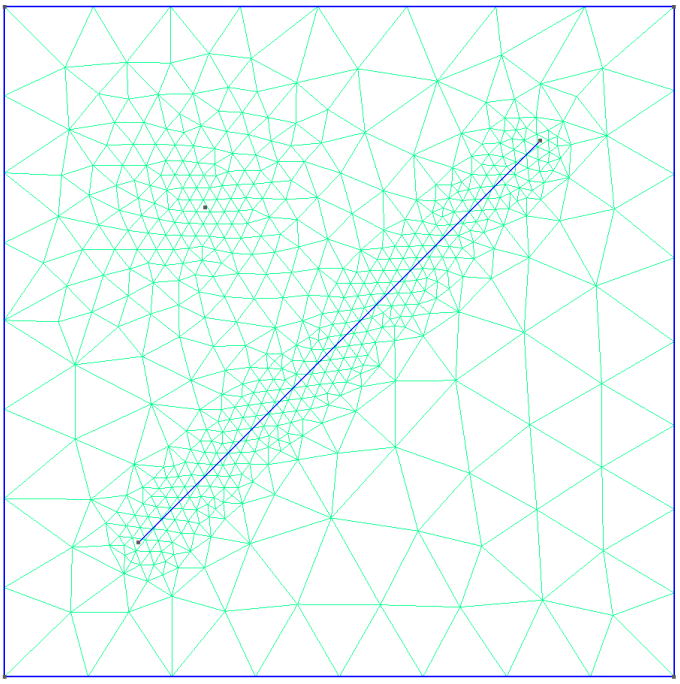


Figure 18: Locally adjusted mesh resolution generated in Gmsh. Output from Code Segment 2.

Structured meshes: Structured meshes, or local patches of structured meshes, can be created by using transfinite curves and surfaces. An example of this is shown in Code Segment 3, with output in Figure 19. In the example, two patches of transfinite meshes are created. The green mesh consists of linearly distributed transfinite nodes, while the orange consists of nodes following a geometric progression with exponent 1.5. The cyan mesh is a standard unstructured mesh. To get rectangular cells, we use the recombine feature, combining the triangular grid into quadrilaterals within the transfinite meshes.

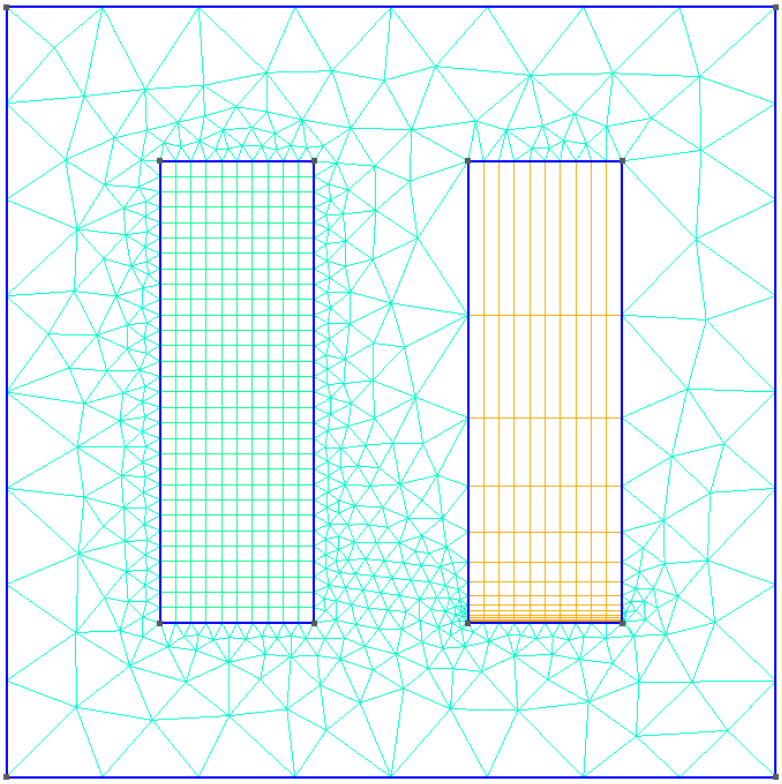


Figure 19: Local patches of structured meshes generated in Gmsh. Output from Code Segment 3.

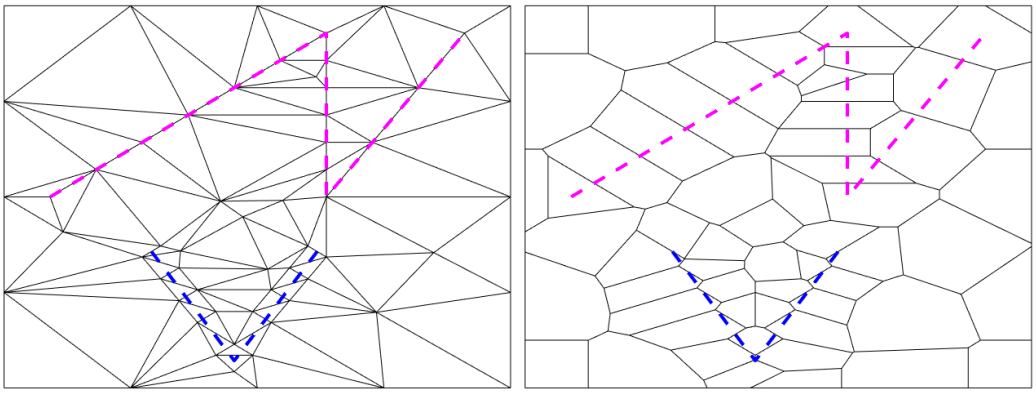
3.2.3 Limitations of Gmsh

Although Gmsh is a well-developed solution for both meshing and CAD work, it does have some limitations, especially when combining it with MRST and UPR. One key difference between the two systems arises from the fact that Gmsh operates with triangulations, while the UPR module of MRST operates with PEBI grids. As these are dualities, we can easily convert between triangulation nodes to PEBI sites. This does, however, have a potential of losing information about cell faces.

When converting from a triangulation to a PEBI grid, vertices in the triangulation are used as sites in the PEBI grid, and two sites connected in the triangulation share a PEBI face. As such, any face constraint in the triangulation is guaranteed to have a cell path in the PEBI grid, but there is no guarantee for the width of the path. This is especially true when the path is turning. An even bigger issue arises

from cell constraints in the triangulation. There is no guarantee of triangulation cell centers ending up on a PEBI face, and even less chance of the faces aligning with the cell constraints.

An illustration of this challenge is shown in Figure 20. The cells of the PEBI grid align somewhat nicely with the cell constraint on straight lines, but fails without detailed manual control in the corners. The vertices of the PEBI grid are somewhat close to the face constraint, but the edges do not align with the constraint line.



(a) Triangulation (b) PEBI grid

Figure 20: Illustration of misaligned constraints when converting a triangulation to a PEBI grid. The magenta line represents a face constraint in the triangulation, and should preferably be a cell constraint in the PEBI grid. The blue line represents a cell constraint in the triangulation, and should preferably be a face constraint in the PEBI grid.

The simplest way of handling this limitation would be to manually control the mesh cell placement to ensure smooth conversion for both types of constraints. As Gmsh is primarily developed for automatic mesh generation, it does not have options for manual cell placement, and any manual refinements must be done through workarounds such as embedding points and lines, adjusting the mesh size field and creating transfinite meshes.

4 Combining MRST and Gmsh

As MRST and Gmsh both have their limitations, adopting Gmsh into an MRST workflow may be beneficial. This section will discuss an existing method for loading Gmsh meshes, `gmshToMrst`, and introduce a new package for integrating the two, `gmsh4mrst`.

4.1 `gmshToMrst`

Originally developed by Johansson [11] as a stand-alone method for loading Gmsh meshes into MRST, `gmshToMrst` became part of MRST's module library with release

2022a. The module reads the mesh from an .m-file and performs the necessary computations for converting the mesh to an MRST grid. While it does this job well, one key requirement of gmshToMrst is that the Gmsh mesh has been computed beforehand. This requires users to manually create the Gmsh mesh, slowing down the rapid prototyping MRST is designed for, while forcing its users to learn an entirely new program in order to generate the grids.

4.2 gmsh4mrst

To help with this, I have developed a new software module, gmsh4mrst, to enable automatic Gmsh mesh generation from MATLAB. By abstracting most of the manual work required for generating meshes in Gmsh, gmsh4mrst is designed to enable users to use Gmsh as a backend for mesh creation, speeding up the mesh generation of detailed domains, without the user having to spend time learning a new software. The goal of gmsh4mrst is to enable a near drop-in replacement of distmesh with Gmsh-created meshes.

4.2.1 Installation of gmsh4mrst

In order to integrate the two systems, gmsh4mrst is split in two parts, one written in Python and one in MATLAB. The Python package is hosted on PyPi [1], and can easily be installed from there. The MATLAB package must be manually downloaded from Github [2], and the files must be added to the MATLAB path.

The Python package works as a stand-alone package. The MATLAB package, however, uses Python to create base meshes, and therefore requires that the Python package is installed to run. MATLAB must be run from an environment with the Python package installed, whether that is through a virtual environment or the base Python installation on the computer.

4.2.2 Features of gmsh4mrst

The primary target of gmsh4mrst is to automate Gmsh mesh creation for use in MRST grids, while maintaining the flexibility needed for optimal grid creation. Care has been taken to ensure gmsh4mrst is as robust as possible, especially when it comes to connecting MATLAB and Python. The module is designed to easily handle complex, non-convex domains, and to precisely and quickly adapt the grid to all given face- and cell constraints.

One key feature of gmsh4mrst is its many user-settable arguments and parameters. By leaving most grid-refinement decisions available to the user, gmsh4mrst can be used for almost every grid creation necessary, while reasonable defaults ensure the module can be used for simple grids without much time spent on refinement. An overview of the available arguments is given in Appendix B.

4.2.3 Using `gmsh4mrst` in Python

The Python part of `gmsh4mrst` is where most of the grid generation is done. While the actual implementation and specifications may change over time, the Python package currently implements three methods.

`background_grid_2D`: As arguably the most basic of the implemented methods, `background_grid_2D` uses Gmsh to create a simple background triangulation without any embedded points or lines, but with refinement around face- and cell constraints. This results in a uniform mesh, and can be used as a straight replacement of the Distmesh algorithm. The user has detailed control of the mesh, including any refinement done along the constraints. An example of a grid produced using `background_grid_2D` is shown in Figure 21a.

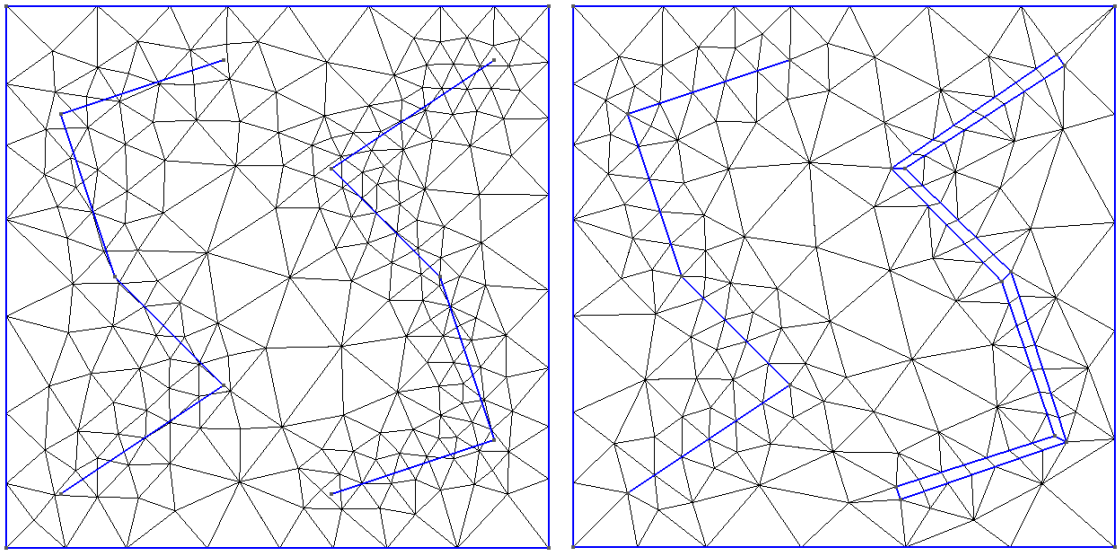
`delaunay_grid_2D`: Going one step further, `delaunay_grid_2D` creates a background triangulation, but include the constraints in the generated mesh. Face constraints are embedded directly as lines or points, ensuring that the faces of the resulting Delaunay triangulation align with the constraints. Each cell constraint is wrapped in a transfinite mesh, ensuring the constraints are traced with cells in the triangulation. As a result of using transfinite meshes, `delaunay_grid_2D` fails if any cell constraints intersect either each other or any face constraints. An example of a grid produced using `delaunay_grid_2D` is shown in Figure 21b.

`pebi_base_2D`: The most complex of the implemented methods, `pebi_base_2D` creates a background triangulation, but include both face- and cell constraints as transfinite meshes. The idea behind this is that it ensures a distribution of Delaunay vertices – and by extension PEBI sites after converting the triangulation – around the constraints, ensuring face constraints are traced by PEBI faces and cell constraints by PEBI sites. As a result of using transfinite meshes, `pebi_base_2D` fails if any constraints intersect. An example of a grid produced using `pebi_base_2D` is shown in Figure 21c.

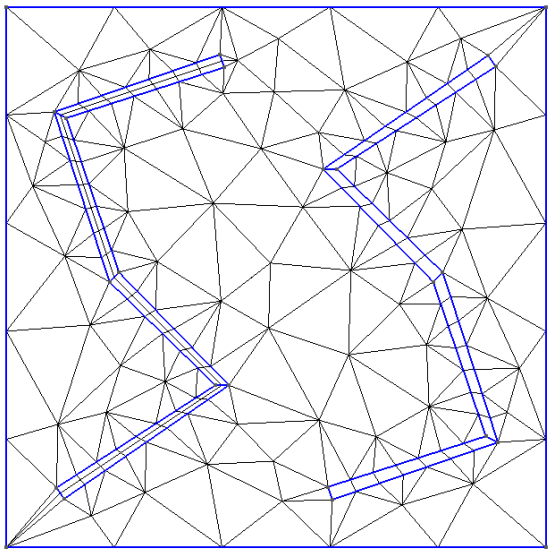
All methods implemented in the Python package output Delaunay triangulations. The difference in outputs of the methods is shown in Figure 21, where the left sides of the grids contain face constraints, and the right sides contain cell constraints.

4.2.4 Using `gmsh4mrst` in MATLAB

The MATLAB part of `gmsh4mrst` contains all code for converting the Gmsh meshes to MRST grids, as well as some code for creating PEBI sites. While the actual implementation and specifications may change over time, the MATLAB package currently implements three methods.



(a) `background_grid_2D` (b) `delaunay_grid_2D`



(c) `pebi_base_2D`

Figure 21: Comparison of the Python methods implemented in `gmsh4mrst`. The grids have a fault on the left side, and a well on the right side.

`pebiGrid2DGmsh:` Designed to be a semi-direct replacement of MRST's `pebiGrid2D`, `pebiGrid2DGmsh` uses `background_grid_2D` to create a background grid, then creates well and fracture sites in MATLAB. This gives it the same flexibility and robustness as `pebiGrid2D`, while avoiding the slow site distribution of Distmesh. An example of a grid produced using `pebiGrid2DGmsh` is shown in Figure 22a.

`delaunayGrid2DGmsh:` As a simple function wrapper, `delaunayGrid2DGmsh` uses `delaunay_grid_2D` to create a Delaunay triangulation capturing the supplied constraints, then converts it to an MRST data type without converting it to a PEBI grid. As no conversion is done, this leaves the output grid as a Delaunay triangulation, but it avoids the extra work of converting without losing constraint information. Due to using `delaunay_grid_2D`, no cell constraints may intersect. An example of a grid produced using `delaunayGrid2DGmsh` is shown in Figure 22b.

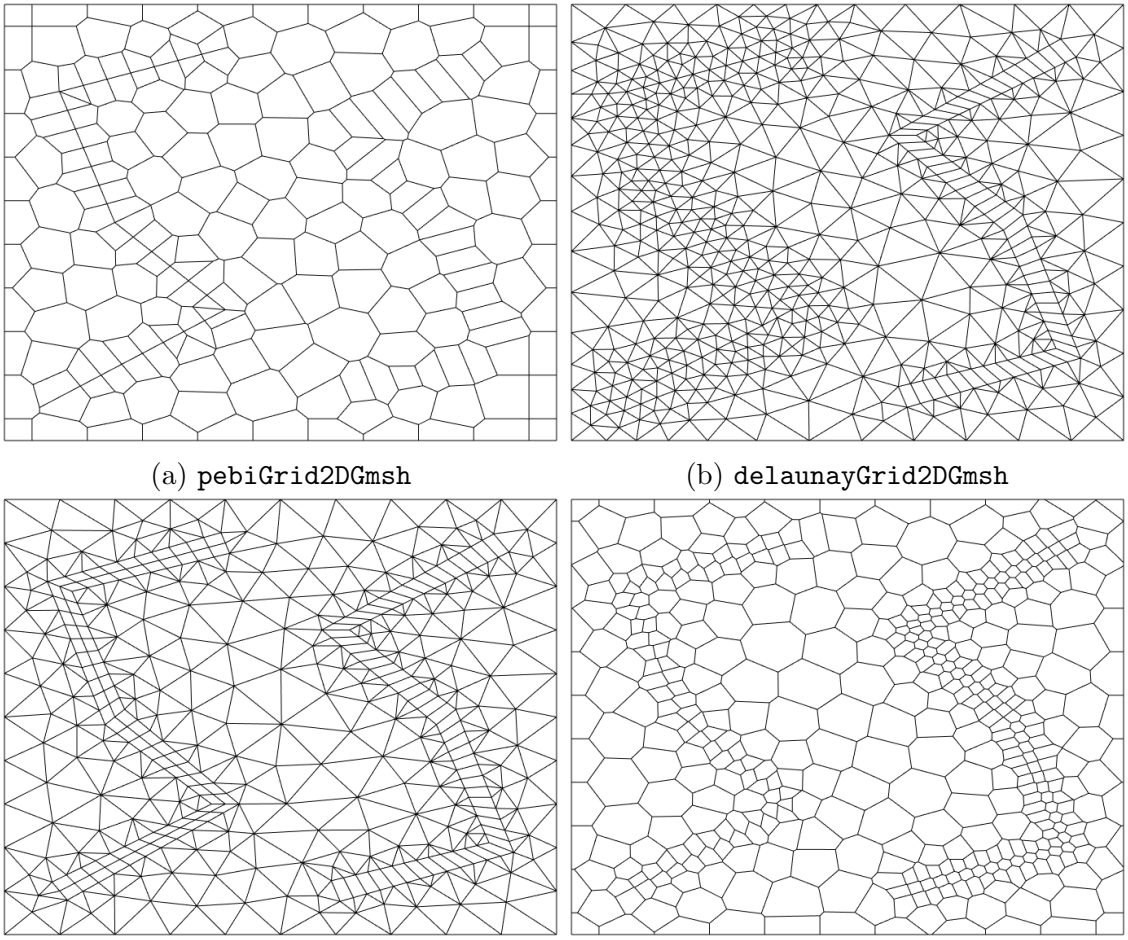
`pebiGrid2DGmshBase:` A somewhat experimental method, `pebiGrid2DGmshBase` uses `pebi_base_2D` to create a triangulation with constraints embedded as transfinite meshes. The user can then choose whether to convert the triangulation to a PEBI grid or not. This results in a relatively flexible method, but due to using `pebi_base_2D`, no constraints may intersect. An example of a grid produced using `pebiGrid2DGmshBase` is shown in Figure 22c and Figure 22d.

All the MATLAB methods gives the user complete control of all parameters of the underlying Python grid creation, with `pebiGrid2DGmsh` additionally letting the user control the fault- and well site creation done in MATLAB. The difference in outputs of the methods is shown in Figure 22, where the left sides of the grids contain face constraints, and the right sides contain cell constraints.

4.2.5 Mechanics of `gmsh4mrst`

Faults as embedded lines: Faults are implemented differently in the three available methods, but here we discuss faults implemented as embedded face constraint lines in Gmsh, which is how it is done in `delaunayGrid2D`. In Gmsh, embedding a line is a relatively simple process, but extra care is taken to ensure intersections between faults are modelled correctly. If two constraint segments intersect, both segments are split at the segment, creating four sub-segments – two starting at the intersection and two ending at the intersection.

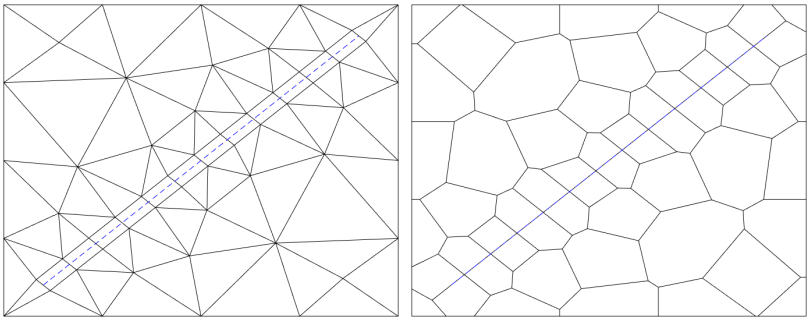
Faults as transfinite grids: In `pebiGrid2DGmshBase`, faults are implemented as transfinite grids. By creating a structured grid around the fault, we can manually control the site distribution around the line, ensuring it ends up as a face in the final grid. We can create protective sites around the faults by increasing the number of perpendicular nodes used in the transfinite grid. When the grid is converted to PEBI, each fault grid should have an even number of perpendicular nodes to make it a face in the grid, while they should have an odd number of nodes to stay as a face in the Delaunay triangulation. An illustration of this process is shown in Figure 23.



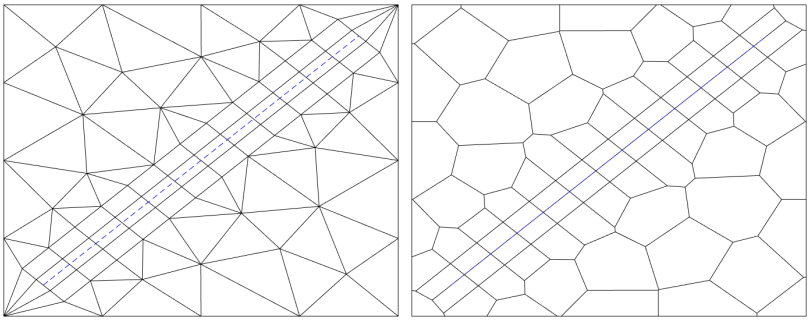
(a) `pebiGrid2DGmsh` (b) `delaunayGrid2DGmsh`

(c) `pebiGrid2DGmshBase` (d) `pebiGrid2DGmshBase` as PEBI

Figure 22: Comparison of the MATLAB methods implemented in `gmsh4mrst`. The grids have a fault on the left side, and a well on the right side.



(a) Fault represented by 2-width transfinite grid.



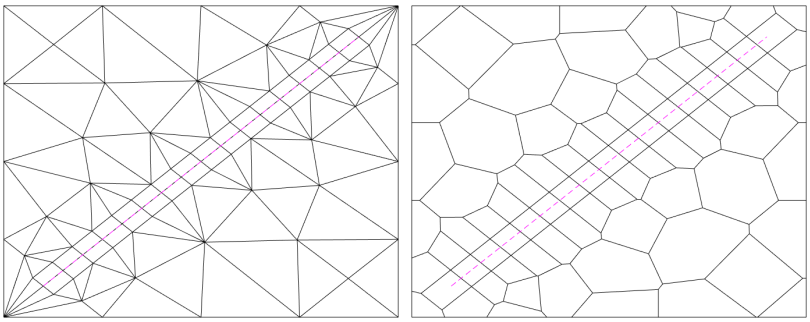
(b) Fault represented by 4-width transfinite grid.

Figure 23: Faults represented as transfinite grids. The blue line represents a fault. The plots to the left are Delaunay grids before conversion, and the plots to the right are the resulting PEBI grids after conversion.

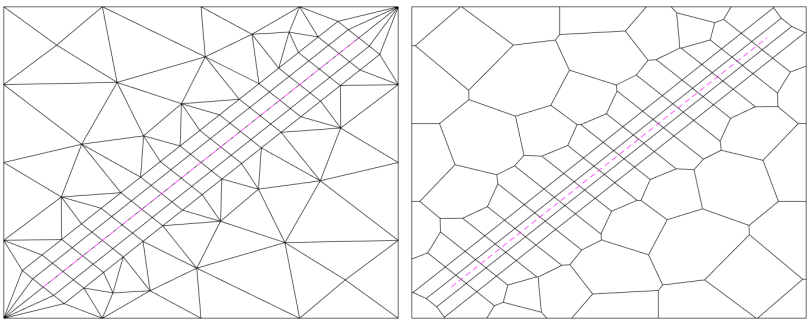
Wells as transfinite grids: Wells are implemented differently in the three methods as well, but here we discuss wells implemented as transfinite grids in Gmsh, which is how it is done in `delaunayGrid2D` and `pebiGrid2DGmshBase`. Much like for faults, we can manually control the site distribution around wells by using transfinite grids, ensuring they end up as PEBI sites after conversion. By increasing the number of perpendicular nodes used in the transfinite grid, we can also create protective sites around the well lines. When the grid is converted to PEBI, each well grid should have an odd number of perpendicular nodes to make it a site in the grid while they should have an even number of nodes to stay as sites in the Delaunay triangulation. An illustration of this process is shown in Figure 24.

Faults and wells in MATLAB: Fault and well creation in `pebiGrid2DGmsh` is done in MATLAB, and follows the same process as `pebiGrid2D` from Berge [3]. In short, this process places well sites and fault sites where they should be, and then later remove conflicting sites from the background grid. This means the Gmsh grid is constructed without having to worry about constraints, and then cleaned up later. An illustration of this process is shown in Figure 25.

Mesh refinement: The mesh refinement in `gmsh4mrst` is done by using mesh size fields, as introduced in Section 3.2.1. Faults are used as inputs for one distance field, and wells are used as inputs for another. These fields are then used as input in their own threshold fields, and the minimum of these threshold fields are used as the final

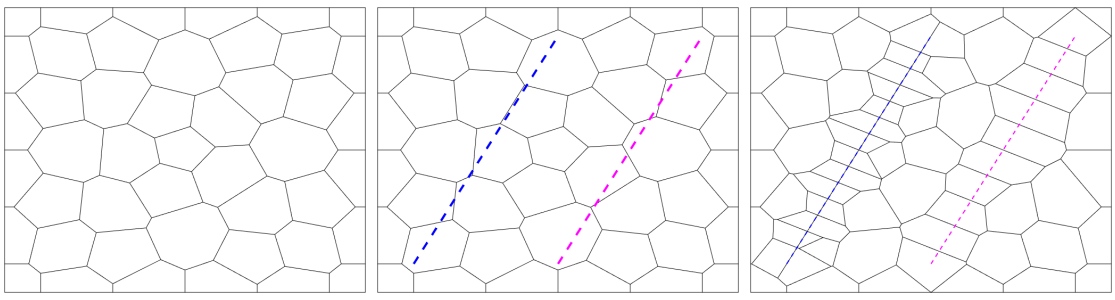


(a) Well represented by 3-width transfinite grid.



(b) Well represented by 5-width transfinite grid.

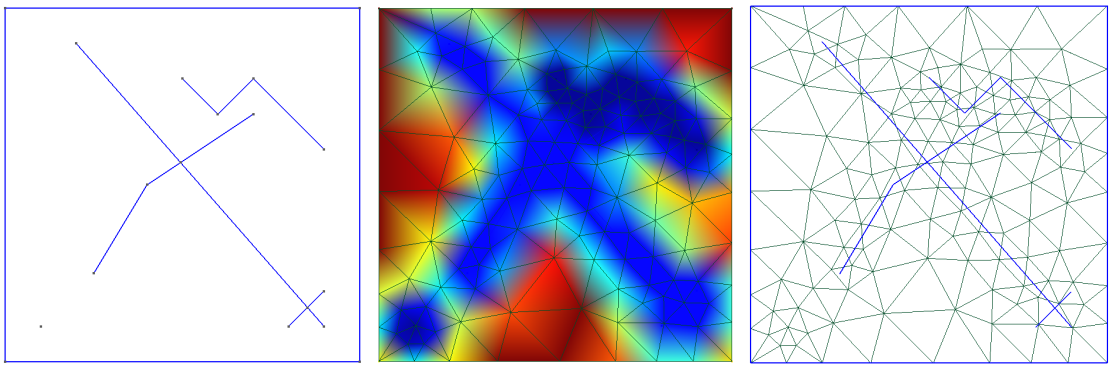
Figure 24: Wells represented as transfinite grids. The magenta line represents a well. The plots to the left are Delaunay grids before conversion, and the plots to the right are the resulting PEBI grids after conversion.



(a) Background grid. (b) Add constraints. (c) Remove conflict points.

Figure 25: Creating faults and wells in MATLAB. The mesh in (a) is created using Gmsh. The constraints in (b) follow a familiar pattern – the blue line is a fault, and the magenta line is a well. In (c), we have removed the conflict points and are left with a conforming grid.

mesh size field. The distance constraints and the minimum size of the threshold fields are controlled by user-settable parameters, as discussed in Appendix B. The maximum size of the threshold fields is set to the default cell size. An illustration of this process is shown in Figure 26.



(a) Constraints. (b) The mesh size field. (c) The produced mesh.

Figure 26: Mesh refinement in `gmsh4mrst`. The input of the mesh refinement is shown as constraints in (a). Based on this, `gmsh4mrst` creates the mesh size field shown in (b). The result is shown in (c) as a Delaunay triangulation, but the method still applies when the triangulation is later converted to PEBI.

Combining MATLAB and Python: While MATLAB has a decent interface for calling Python methods, it is strict on the data conversion, and does not work when trying to send multi-dimensional- or cell arrays to Python. In order to bypass this, both the shape and constraints are converted to structs before being sent to Python, with constraints being converted to a multi-level struct. The result is that Python receives a MATLAB struct, which it reads as a Python dictionary, with an `x`- and `y`-key for each constraint. These keys point to a one-dimensional array, holding the x- and y-components of each constraint, respectively.

4.2.6 Limitations of `gmsh4mrst`

As `gmsh4mrst` uses Gmsh to produce its meshes, the primary limitation of Gmsh – that it produces Delaunay triangulations rather than PEBI grids – creates limitations for `gmsh4mrst` as well. While this has been worked around in `pebiGrid2DGmsh`, it increases the work needed to produce good PEBI grids from Gmsh, and is the reason why we cannot encode faults and wells directly in the Gmsh mesh.

One key limitation of `delaunayGrid2D` and `pebiGrid2DGmshBase` arise from their use of transfinite grids. These grids cannot intersect, limiting the usability of the two methods – `delaunayGrid2D` will not work for cases where wells cross either other wells or faults, and `pebiGrid2DGmshBase` will not work for cases where any constraints cross each other. While the methods work well when this is not the case, this places strict limitations on the usability of the methods.

Another thing worth mentioning is the structuredness of the transfinite grids produced in the above-mentioned methods. Due to the nature of transfinite grids, they

produce highly structured subgrids. This is beneficial because it gives greater control of the produced grid, and thus great control of how it is converted into a PEBI grid, but may also be suboptimal if the grid is to be used for simulations or other work. In cases where this may be an issue, it would be better to use `pebiGrid2DGmsh`, or any of the methods of the UPR module of MRST.

One key design goal of `gmsh4mrst` was to create a drop-in replacement of Distmesh in `pebiGrid2D`. While this is somewhat complete in `pebiGrid2D`, it does not have complete API compitability, with many more arguments, as well as some that are not included in `pebiGrid2DGmsh`. Some work is therefore needed if complete API compitability is wanted.

The final limitation worth mentioning is a distinct lack of feedback to the user, especially if something crashes. While basic argument parsing is done, any other errors are unhandled, creating confusing error messages. This is doubly true if anything goes wrong on the Python-side of things, where I have experienced everything from the connection between MATLAB and Python abruptly closing, to MATLAB crashing without any error message. This makes it hard for the user to understand what has gone wrong, and makes the program less robust.

4.2.7 Examples of `gmsh4mrst`

A key design choice when creating `gmsh4mrst` was to have an easy-to-use interface. This section will present some examples of `gmsh4mrst` programs, as well as their outputs. Source code for the examples is shown in Appendix C.

Complex domains When working with real-life domains, we sometimes have domains that are not perfectly rectangular. To handle this, `gmsh4mrst` supports using any polygon as domain. Code Segment 4 shows how to create a star-shaped domain in `gmsh4mrst`. The output is shown in Figure 27.

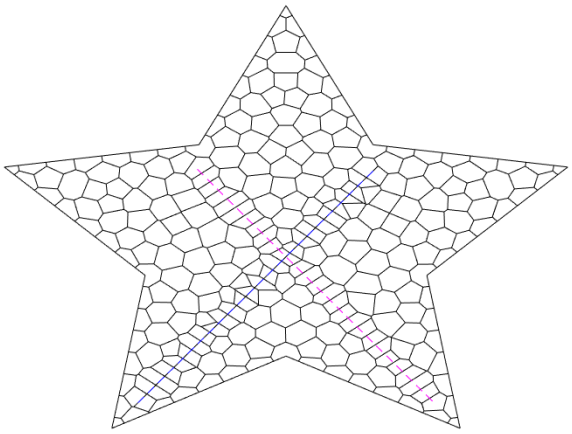


Figure 27: PEBI grid with complex domain, generated using `gmsh4mrst`. The blue line represents a fault, and the magenta line represents a well. Output from Code Segment 4.

Intersecting constraints Real-life geological features may intersect, something `gmsh4mrst` is designed to handle this with ease. Code Segment 5 shows how to create a grid with intersecting constraints. The output is shown in Figure 28.

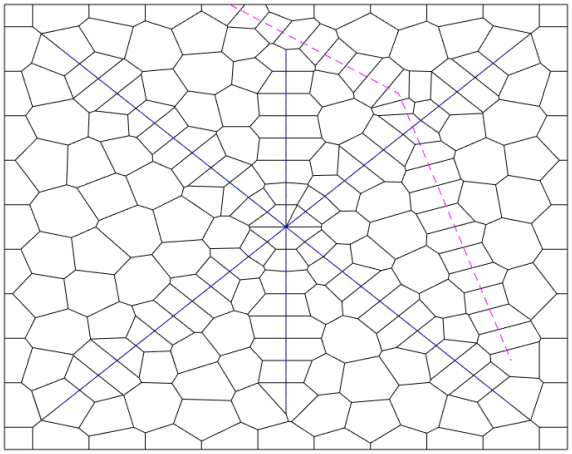


Figure 28: PEBI grid with intersecting constraints, generated using `gmsh4mrst`. The blue lines represent faults, and the magenta line represents a well. Output from Code Segment 5.

Grids with fine details One key drawback of UPR using the `distmesh` algorithm for generating sites is its slowness when creating grids with fine details. This is something `gmsh4mrst` is designed to handle. Code Segment 6 shows how to create a grid spanning a square kilometer, with a 10 centimeter well passing through. The output is shown in Figure 29.

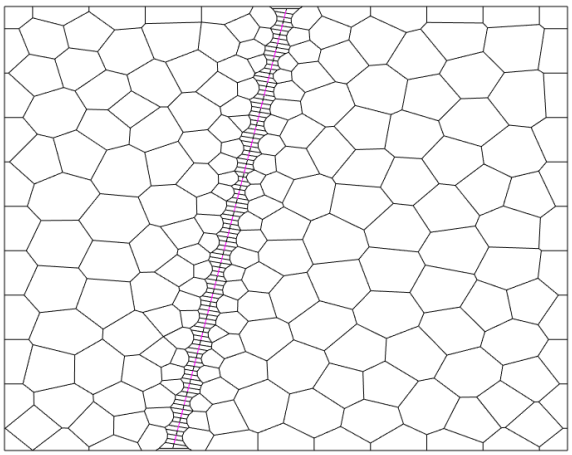


Figure 29: A PEBI grid with a domain of one square kilometer, with a 10 centimeter well passing through the domain. The magenta line represents a well. Output from Code Segment 6.

5 Future work

With the large user base and feature set of Gmsh, there is significant potential for using `gmsh4mrst` in interesting applications. Whether through advanced mesh refinement, improved triangulation algorithms, or any other features of Gmsh, the potential of expanding the use of Gmsh is definitely present.

One key requirement for future improvement of `gmsh4mrst` is by gaining a better understanding of the requirements, wants and needs of potential users. As my domain knowledge is limited, it is hard to fully understand what features are and are not useful, so the first step in any future development is simply to gain users and collect feedback. This way, any developers can ensure that `gmsh4mrst` is optimized for the right work, while any unused features or unneeded parameters may be removed.

One interesting topic to consider is using Gmsh to create structured background grids, instead of the unstructured triangulations `gmsh4mrst` currently produced. This could be achieved using a combination of embeddings and transfinite grids, but would likely require significant manual work in order to make robust.

When looking forward, one clear development of `gmsh4mrst` is to expand the grid generation to 3D. This is something both Gmsh and MRST can handle, and while it would require at least some adaptations of the code, it could potentially open up a new dimension of opportunities when it comes to combining the two tools. As modelling targets usually are three-dimensional, this would also expand the usability of the package significantly.

Finally, MATLAB provides functionality for calling code written in C++, and Gmsh provides a C++ API. Although most of the functionality for Gmsh is already written in C++ and only called from Python, porting `gmsh4mrst` to C++ could potentially lead to increased speed. While this is not needed for models on a smaller scale, it could be beneficial for larger, complex domains with numerous faults and wells. As `gmsh4mrst` provides a MATLAB package separate from the Python package, porting of the Gmsh-calling code could be done without changing anything in how the package is used from MATLAB.

One goal of `gmsh4mrst` was to create an open-source project that can be continuously developed as the needs of the software develop. The software is currently distributed with the same license as Gmsh, which allows anyone to use, change and distribute the software as they want, as long as future distributions are also open-source. With this in mind, I hope this project can be used as a starting point for future development, and that `gmsh4mrst` will become a useful tool for researchers using MRST.

Bibliography

[1] Berg, Andreas B. *gmsh4mrst*. URL: https://pypi.org/project/gmsh4mrst/.

[2] Berg, Andreas B. *gmsh4mrst - Source code*. URL: https://github.com/BollaBerg/gmsh4mrst.

[3] Berge, Runar L. *Unstructured PEBI-grids Adapting to Geological Features in Subsurface Reservoirs*. NTNU, June 2016. URL: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2411565.

[4] Berge, Runar L., Klemetsdal, Øystein S. and Lie, Knut-Andreas. 'Unstructured PEBI Grids Conforming to Lower-Dimensional Objects'. In: *Advanced Modeling with the MATLAB Reservoir Simulation Toolbox*. Ed. by Knut-Andreas Lie and Olav Møyner. Cambridge University Press, 2021, pp. 3–45. DOI: 10.1017/9781009019781.005.

[5] Delaunay, Boris. 'Sur la sphère vide'. In: *Bulletin de l'Académie des Sciences de l'URSS. Classe des sciences mathématiques et na* 6 (1934), pp. 793–800.

[6] Dyken, Christopher and Floater, Michael S. 'Transfinite mean value interpolation'. In: *Computer Aided Geometric Design* 26.1 (2009), pp. 117–134. ISSN: 0167-8396. DOI: https://doi.org/10.1016/j.cagd.2007.12.003. URL: https://www.sciencedirect.com/science/article/pii/S0167839607001240.

[7] Fawzy, Samer et al. 'Strategies for mitigation of climate change: a review'. In: *Environmental Chemistry Letters* 18 (6 Nov. 2020), pp. 2069–2094. DOI: 10.1007/s10311-020-01059-w. URL: https://doi.org/10.1007/s10311-020-01059-w.

[8] Geuzaine, Christophe and Remacle, Jean-François. *Gmsh*. Presentation. URL: https://gmsh.info/doc/course/general_overview.pdf (visited on 25th May 2022).

[9] Geuzaine, Christophe and Remacle, Jean-François. 'Gmsh: A three-dimensional finite element mesh generator with built-in pre- and post-processing facilities'. In: *International Journal for Numerical Methods in Engineering* 79.11 (2009), pp. 1309–1331. DOI: 10.1002/nme.2579. URL: https://gmsh.info/doc/preprints/gmsh_paper_preprint.pdf.

[10] Gmsh. *Gmsh 4.10.2*. URL: https://gmsh.info/doc/texinfo/gmsh.html (visited on 25th May 2022).

[11] Johansson, August. *gmsh_to_mrst*. 2021. URL: https://github.com/augustjohansson/gmsh_to_mrst.

[12] K. Ponting, D. 'Corner Point Geometry in Reservoir Simulation'. In: cp-234-00003 (1989). ISSN: 2214-4609. DOI: https://doi.org/10.3997/2214-4609.201411305. URL: https://www.earthdoc.org/content/papers/10.3997/2214-4609.201411305.

[13] Lie, Knut-Andreas. *An Introduction to Reservoir Simulation Using MATLAB / GNU Octave: User Guide for the MATLAB Reservoir Simulation Toolbox (MRST)*. Cambridge University Press, 2019. DOI: 10.1017/9781108591416.

[14] Liu, Yang et al. 'On Centroidal Voronoi Tessellation—Energy Smoothness and Fast Computation'. In: *ACM Trans. Graph.* 28.4 (Sept. 2009). ISSN: 0730-0301. DOI: 10.1145/1559755.1559758. URL: https://doi.org/10.1145/1559755.1559758.

[15] NASA. *NASA: Climate Change and Global Warming*. URL: https://climate.nasa.gov/ (visited on 30th May 2022).

[16] Persson, Per-Olof and Strang, Gilbert. 'A Simple Mesh Generator in MATLAB'. In: *SIAM Review* 46.2 (2004), pp. 329–345. DOI: 10.1137/S0036144503429121. URL: https://doi.org/10.1137/S0036144503429121.

[17] Shaffer, Gary. 'Long-term effectiveness and consequences of carbon dioxide sequestration'. In: *Nature geoscience* 3.7 (2010), pp. 464–467. ISSN: 1752-0894.

[18] SINTEF. *MRST*. URL: https://www.sintef.no/projectweb/mrst/ (visited on 24th May 2022).

[19] Voronoi, Georges. 'Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Deuxième mémoire. Recherches sur les parallélloèdres primitifs.' In: *Journal für die reine und angewandte Mathematik (Crelles Journal)* (1908), pp. 198–287.

Appendix

A Gmsh examples

This appendix contains the code behind the Gmsh examples listed in Section 3.2.2.

Code Segment 1: Gmsh: Simple square mesh

```
import gmsh

# Always initialize gmsh
gmsh.initialize()

# Create corner points
# Note that the mesh may look different if corners are
# defined clockwise instead of counter-clockwise
p1 = gmsh.model.geo.add_point(0, 0, 0)
p2 = gmsh.model.geo.add_point(1, 0, 0)
p3 = gmsh.model.geo.add_point(1, 1, 0)
p4 = gmsh.model.geo.add_point(0, 1, 0)

# Create line segments
l1 = gmsh.model.geo.add_line(p1, p2)
l2 = gmsh.model.geo.add_line(p2, p3)
l3 = gmsh.model.geo.add_line(p3, p4)
l4 = gmsh.model.geo.add_line(p4, p1)

# Create a curve loop of the edge lines
curve = gmsh.model.geo.add_curve_loop([l1, l2, l3, l4])

# Create a surface of the curve loop
surface = gmsh.model.geo.add_plane_surface([curve])

# Synchronize the CAD kernel to create Gmsh data structures
gmsh.model.geo.synchronize()

# Generate 2D mesh
gmsh.model.mesh.generate(2)

# Show the model by running the GUI
gmsh.fltk.run()

# Always finalize when done using the API
gmsh.finalize()
```

Code Segment 2: Gmsh: Adjusting the mesh size field

```
import gmsh

# Always initialize gmsh
gmsh.initialize()

# Gmsh can easily use different CAD kernels
# In this case, we use OpenCASCADE to create our base domain,
# starting at (0, 0, 0), with sides dx = dy = 1
surface = gmsh.model.occ.add_rectangle(0, 0, 0, 1, 1)

# Synchronize the CAD kernel to create Gmsh data structures
gmsh.model.occ.synchronize()

# Setup our base mesh size
base_size = 0.2

# Create our line
start = gmsh.model.occ.add_point(0.2, 0.2, 0)
end = gmsh.model.occ.add_point(0.8, 0.8, 0)
line = gmsh.model.occ.add_line(start, end)

# Create the point
point = gmsh.model.occ.add_point(0.3, 0.7, 0)

# Synchronize to ensure the new points are available for Gmsh
gmsh.model.occ.synchronize()

# Create a Distance field, calculating the distance from a line
line_distance = gmsh.model.mesh.field.add("Distance")
# Set the input of line_distance to be our line
gmsh.model.mesh.field.set_numbers(line_distance, "CurvesList", [line])
# Set the sample rate of the field to 100
gmsh.model.mesh.field.set_number(line_distance, "Sampling", 100)

# Now we create a Threshold field, using line_distance as our input
# We want to scale for all cells with a distance in [0.05, 0.2]
# We want the size of the cells to be in [base_size / 10, base_size]
line_thresh = gmsh.model.mesh.field.add("Threshold")
gmsh.model.mesh.field.setNumber(line_thresh, "InField", line_distance)
gmsh.model.mesh.field.setNumber(line_thresh, "SizeMin", base_size / 10)
gmsh.model.mesh.field.setNumber(line_thresh, "SizeMax", base_size)
gmsh.model.mesh.field.setNumber(line_thresh, "DistMin", 0.05)
gmsh.model.mesh.field.setNumber(line_thresh, "DistMax", 0.2)

# To use the point, we must set up another distance field
point_dist = gmsh.model.mesh.field.add("Distance")
gmsh.model.mesh.field.set_numbers(point_dist, "PointsList", [point])
```

```
# We can now create a MathEval field, using the square distance as our
# input. We shift F to so that the minimum mesh size is base_size / 10
point_field = gmsh.model.mesh.field.add("MathEval")
gmsh.model.mesh.field.set_string(point_field, "F",
    f"F{point_dist}^2 + {base_size / 10}"
)

# We can now create a Min field, to calculate the minimum of all the
# fields. We then set it as our mesh size field
min_field = gmsh.model.mesh.field.add("Min")
gmsh.model.mesh.field.setNumbers(min_field, "FieldsList",
    [line_thresh, point_field]
)
gmsh.model.mesh.field.setAsBackgroundMesh(min_field)

# Generate 2D mesh
gmsh.model.mesh.generate(2)

# Show the model by running the GUI
gmsh.fltk.run()

# Always finalize when done using the API
gmsh.finalize()
```

Code Segment 3: Gmsh: Structured meshes

```
import gmsh

# Always initialize gmsh
gmsh.initialize()

# Create the linear transfinite mesh
trans_lines = []        # For storing lines
trans_loops = []        # For storing curve loops
trans_surfaces = []     # For storing surfaces
for start_x in (0.2, 0.6):
    # Create transfinite mesh corners
    corner1 = gmsh.model.geo.add_point(start_x, 0.2, 0)
    corner2 = gmsh.model.geo.add_point(start_x + 0.2, 0.2, 0)
    corner3 = gmsh.model.geo.add_point(start_x + 0.2, 0.8, 0)
    corner4 = gmsh.model.geo.add_point(start_x, 0.8, 0)

    # Create transfinite mesh lines
    lines = [
        gmsh.model.geo.add_line(corner1, corner2),
        gmsh.model.geo.add_line(corner2, corner3),
        gmsh.model.geo.add_line(corner3, corner4),
        gmsh.model.geo.add_line(corner4, corner1)
    ]
    trans_lines.append(lines)

    # Create transfinite mesh curve loop, save it in trans_loops
    trans_loop = gmsh.model.geo.add_curve_loop(lines)
    trans_loops.append(trans_loop)

    # Create the transfinite mesh surface
    trans_surface = gmsh.model.geo.add_plane_surface([trans_loop])
    trans_surfaces.append(trans_surface)

# Make the first mesh transfinite
# We use 11 points along the short sides of the mesh,
# and 31 points along the long sides of the mesh
# This gives us a grid of 10 * 30 cells
gmsh.model.geo.mesh.set_transfinite_curve(trans_lines[0][0], 11)
gmsh.model.geo.mesh.set_transfinite_curve(trans_lines[0][1], 31)
gmsh.model.geo.mesh.set_transfinite_curve(trans_lines[0][2], 11)
gmsh.model.geo.mesh.set_transfinite_curve(trans_lines[0][3], 31)

# We can now make the surface transfinite
# If our surface had more than 4 corners, we would have to manually
# specify the corners to use for the transfinite interpolation
# This is automatic for 3 and 4 corners
gmsh.model.geo.mesh.set_transfinite_surface(trans_surfaces[0])
```

```
# We make the second mesh transfinite
# We keep the number of points, but make the points along the long
# sides follow a geometric progression with power = 1.5
# Note how we reverse the second line, to get a symmetric distribution
gmsh.model.geo.mesh.set_transfinite_curve(trans_lines[1][0], 11)
gmsh.model.geo.mesh.set_transfinite_curve(trans_lines[1][1], 31,
                                          coef=1.5)
gmsh.model.geo.mesh.set_transfinite_curve(trans_lines[1][2], 11)
gmsh.model.geo.mesh.set_transfinite_curve(trans_lines[1][3], 31,
                                          coef=-1.5)

gmsh.model.geo.mesh.set_transfinite_surface(trans_surfaces[1])

# Recombine triangles to get a rectangular grid
gmsh.model.geo.mesh.set_recombine(2, trans_surfaces[0])
gmsh.model.geo.mesh.set_recombine(2, trans_surfaces[1])


# Create a base surface
p1 = gmsh.model.geo.add_point(0, 0, 0)
p2 = gmsh.model.geo.add_point(1, 0, 0)
p3 = gmsh.model.geo.add_point(1, 1, 0)
p4 = gmsh.model.geo.add_point(0, 1, 0)

l1 = gmsh.model.geo.add_line(p1, p2)
l2 = gmsh.model.geo.add_line(p2, p3)
l3 = gmsh.model.geo.add_line(p3, p4)
l4 = gmsh.model.geo.add_line(p4, p1)

curve = gmsh.model.geo.add_curve_loop([l1, l2, l3, l4])

# We create the base surface everywhere except within the
# transfinite meshes we created
surface = gmsh.model.geo.add_plane_surface([curve] + trans_loops)

# Synchronize the CAD kernel to create Gmsh data structures
gmsh.model.geo.synchronize()

# Generate 2D mesh
gmsh.model.mesh.generate(2)

# Show the model by running the GUI
gmsh.fltk.run()

# Always finalize when done using the API
gmsh.finalize()
```

B Parameters in `gmsh4mrst`

This appendix gives a brief description of the available user-settable parameters of `gmsh4mrst`. The parameters are formatted as they are used in MATLAB, but all parameters – unless otherwise noted – are available in Python as well. Unless specifically stated, all Python parameters have the same name as their MATLAB equivalents, only written in `snake_case` instead of `camelCase`.

B.1 Background grid refinement

Most of the available arguments control the refinement of the background grid produced in Python, and are passed directly through to the Python package. These arguments are listed in Table 1.

Table 1: Arguments controlling background grid refinement in `gmsh4mrst`.

Argument	Description
`resGridSize`	The default size of each cell in the grid. In Python, this argument is named `cell_dimensions`.
`faceConstraintFactor`	Controls the size of the cells close to the face constraints. The minimum size of the face constraint threshold field is given as `faceConstraintFactor` * `resGridSize`.
`faceConstraintRefinementFactor`	The cell size in the background refinement done around the face constraints. Only available in `pebiGrid2DGmsh` and `pebiGrid2DGmshBase`.
`minFCThresholdDistance`	Distance from face constraints where cell dimensions will start increasing. Used as the minimum distance of the face constraint threshold field.
`maxFCThresholdDistance`	Distance from face constraints where cell dimensions will be back to their default value. Used as the maximum distance of the face constraint threshold field.
`FCMeshSampling`	How many points along each face constraint line to sample for distance calculation.
`cellConstraintFactor`	Same as `faceConstraintFactor`, but for cell constraints.
`cellConstraintLineFactor`	Overrides `cellConstraintFactor` for cell constraint lines. Only available in `pebiGrid2DGmsh`.
`cellConstraintPointFactor`	Overrides `cellConstraintFactor` for cell constraint points.

Continued on next page.

Argument	Description
`cellConstraintRefinementFactor`	The cell size in the background refinement done along the cell constraints. Only available in `pebiGrid2DGmshBase` and `delaunayGrid2DGmsh` and `pebiGrid2DGmshBase`.
`minCCThresholdDistance`	Same as `minFCThresholdDistance`, but for cell constraints.
`maxCCThresholdDistance`	Same as `maxFCThresholdDistance`, but for cell constraints.
`CCMeshSampling`	Same as `FCMeshSampling`, but for cell constraints.
`faceIntersectionFactor`	Same as `faceConstraintFactor`, but for intersections of constraints. Only available in `pebiGrid2DGmsh` and `delaunayGrid2DGmsh`.
`minIntersectionDistance`	Same as `minFCThresholdDistance`, but for intersections of constraints. Only available in `pebiGrid2DGmsh` and `delaunayGrid2DGmsh`.
`maxIntersectionDistance`	Same as `maxFCThresholdDistance`, but for intersections of constraints. Only available in `pebiGrid2DGmsh` and `delaunayGrid2DGmsh`.
`meshAlgorithm`	Which Gmsh meshing algorithm to use.
`recombinationAlgorithm`	Which Gmsh recombination algorithm to use.

B.2 Constraint creation in `pebiGrid2DGmsh`

The MATLAB method `pebiGrid2DGmsh` creates constraints in MATLAB instead of Python, and therefore accepts arguments that influence how this creation is done. These arguments do the same as in `pebiGrid2D`, and are kept to keep the replacement as simple as possible. The arguments are listed in Table 2.

Table 2: Arguments controlling constraint creation in `pebiGrid2DGmsh`.

Argument	Description
`interpolateCC`	Whether any interpolation should be done along the cell constraint lines.
`CCRefinement`	Whether refinement should be done around the cell constraints.
`CCEps`	The refinement transition around the cell constraints.
`CCRho`	Controls the distance between the cell constraint sites. The distance between the sites is given as `CCRho` * `CCFactor` * `resGridSize`.

Continued on next page.

Argument	Description
protLayer	Whether a protection layer should be added around the cell constraints.
protD	The distance between the cell constraint and protection sites.
interpolateFC	Same as interpolateCC, but for face constraints.
circleFactor	The ratio between the radius and distance between the circles used to create the face constraints.
FCRho	Same as CCRho, but for face constraints.
sufFCCond	Whether the fault condition (Definition 3.2) should be enforced, instead of a less strict condition.

B.3 Transfinite grid control

Several arguments are available for controlling the transfinite grids used to produce cell constraints in delaunay_grid_2D and both face- and cell constraints in pebi_base_2D. Unless explicitly stated, all arguments are available in both these methods. These arguments are listed in Table 3.

Table 3: Arguments controlling transfinite grid creation in gmsh4mrst.

Argument	Description
faceConstraintParallelFactor	Overrides faceConstraintFactor along the face constraint lines, i.e. sets the length of the transfinite cells along the constraints. Only available in pebiGrid2DGmshBase.
faceConstraintPerpendicularFactor	Overrides faceConstraintFactor across the face constraint lines, i.e. sets the width of the transfinite cells across the constraints. Only available in pebiGrid2DGmshBase.
faceConstraintPointFactor	Overrides faceConstraintFactor for cell constraint points. Only available in pebiGrid2DGmshBase.
faceConstraintPerpendicularCells	The number of transfinite nodes should be placed across the end-segment of the face constraint transfinite grids, i.e. how many cells wide should the face constraint grid be. Only available in pebiGrid2DGmshBase.
cellConstraintParallelFactor	Overrides cellConstraintFactor along the cell constraint lines, i.e. sets the length of the transfinite cells along the constraints.

Continued on next page.

Argument	Description
`cellConstraintPerpendicularFactor`	Overrides `cellConstraintFactor` across the cell constraint lines, i.e. sets the width of the transfinite cells across the constraints.
`cellConstraintPerpendicularCells`	The number of transfinite nodes should be placed across the end-segment of the cell constraint transfinite grids, i.e. how many cells wide should the cell constraint grid be. Only available in `pebiGrid2DGmshBase`.

B.4 Miscellaneous arguments

Some arguments are not part of the above groups, but still worth mentioning. These arguments are listed in Table 4.

Table 4: Miscellaneous arguments in `gmsh4mrst`.

Argument	Description
`shape`	The shape of the domain. Can either be a 2D array of points or a size $[x, y]$. If a size is passed, the domain will be a square between $[0, 0]$ and $[x, y]$.
`faceConstraints`	The face constraints the grid should adapt to.
`cellConstraints`	The cell constraints the grid should adapt to.
`convertToPEBI`	Whether the grid should be converted to PEBI. Only available in `pebiGrid2DGmshBase`.
`savename`	A name the Gmsh mesh should be saved as. Only available in the Python package, as it is set automatically in MATLAB.
`run_frontend`	Whether the Gmsh frontend should be run after creating the grid. Only available in the Python package.

C gmsh4mrst examples

This appendix contains the code behind the gmsh4mrst examples listed in Section 4.2.7.

Code Segment 4: gmsh4mrst: Complex domain

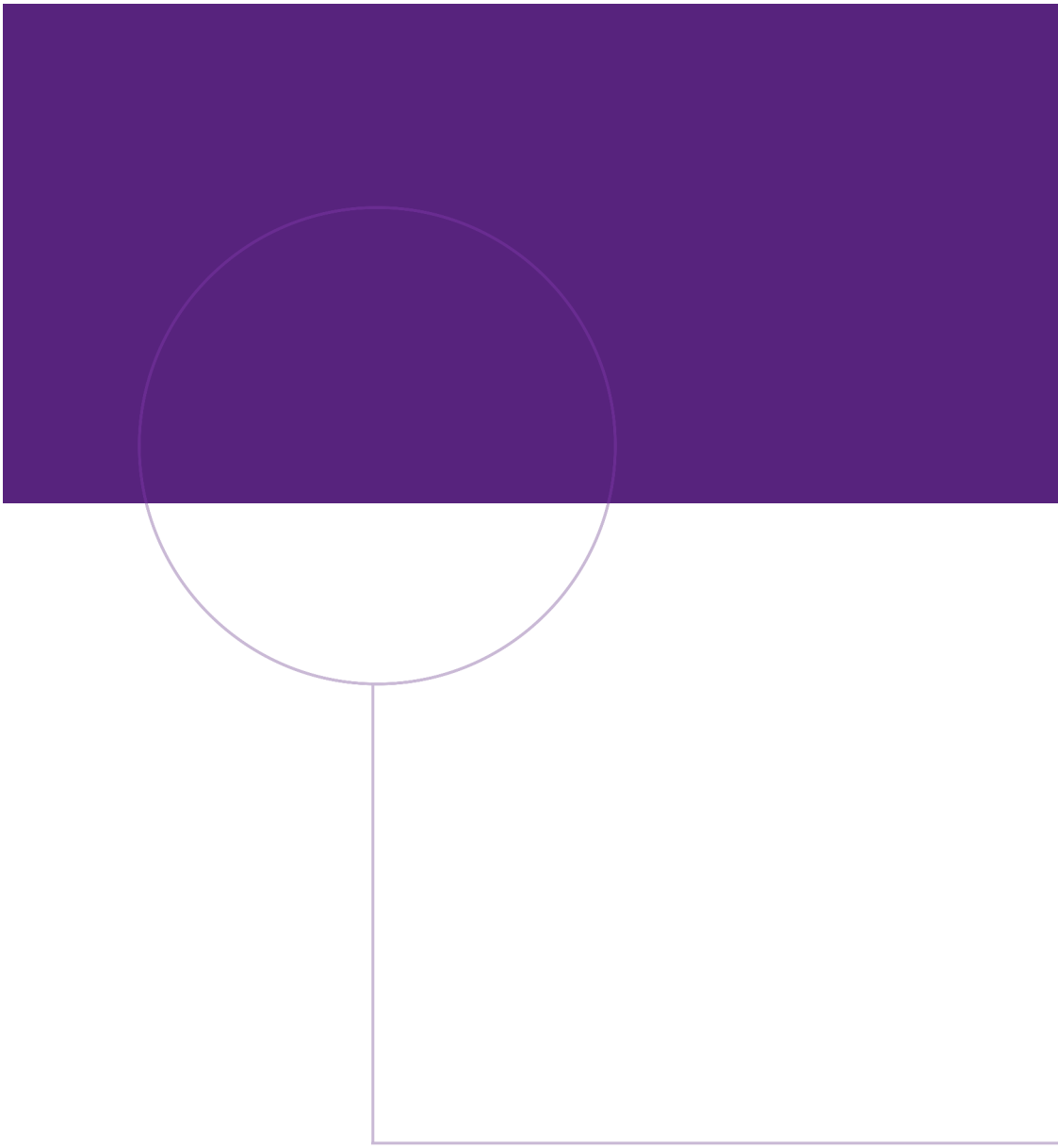
```
% Set the default grid size
resGridSize = 0.1;

% Set the domain to a star-shape
outer = 1;
inner = 0.5;
domain = [
   outer*cosd(18) outer*sind(18);
   inner*cosd(54) inner*sind(54);
   outer*cosd(90) outer*sind(90);
   inner*cosd(126) inner*sind(126);
   outer*cosd(162) outer*sind(162);
   inner*cosd(198) inner*sind(198);
   outer*cosd(234) outer*sind(234);
   inner*cosd(270) inner*sind(270);
   outer*cosd(306) outer*sind(306);
   inner*cosd(342) inner*sind(342);
];

% Create a single fault and well
faceConstraints = {[-0.5 -0.7; 0.3 0.3]};
cellConstraints = {[-0.3 0.3; 0.5 -0.7]};

% Create the PEBI grid
G = pebiGrid2DGmsh( ...
   resGridSize, ...
   domain, ...
   'faceConstraints', faceConstraints, ...
   'cellConstraints', cellConstraints ...
);
```

Code Segment 5: `gmsh4mrst`: Intersecting constraints

```
% Set the default grid size
resGridSize = 0.1;

% Set the domain to the unit square
domain = [1 1];

% Create intersecting constraints
faceConstraints = {
   [0.1 0.1; 0.9 0.9], ...
   [0.5 0.1; 0.5 0.9], ...
   [0.9 0.1; 0.1 0.9], ...
};
cellConstraints = {
   [0.4 1; 0.7 0.8; 0.9 0.2] ...
};

% Create the PEBI grid
G = pebiGrid2DGmsh( ...
   resGridSize, ...
   domain, ...
   'faceConstraints', faceConstraints, ...
   'cellConstraints', cellConstraints ...
);
```

Code Segment 6: `gmsh4mrst`: Detailed grid

```
% Set the default grid size
resGridSize = 0.1;

% Set the domain to the unit square
% We use kilometers as units
domain = [1 1];

% Set the size of cells near our well
cellConstraintFactor = 0.1;

% Set the width of the well
cellConstraintPerpendicularFactor = 0.0001; % 10 cm

% Create a simple well passing through the domain
cellConstraints = {
   [0.5 0.99; 0.3 0.01] ...
};

% Create the PEBI grid
% We can use pebiGrid2DGmshBase as we have no crossing constraints
G = pebiGrid2DGmshBase( ...
   resGridSize, ...
   domain, ...
   'cellConstraints', cellConstraints, ...
   'cellConstraintFactor', cellConstraintFactor, ...
   'cellConstraintPerpendicularFactor', ...
   cellConstraintPerpendicularFactor ...
);
```



NTNU

Norwegian University of
Science and Technology