NTNU

Norwegian University of
Science and Technology

Unstructured PEBI-grids Adapting to Geological Features in Subsurface Reservoirs

Runar Lie Berge

Master of Science in Physics and Mathematics
Submission date: June 2016
Supervisor: Knut Andreas Lie, MATH

Norwegian University of Science and Technology
Department of Mathematical Sciences

Abstract

Extensive research has been done on the generation of unstructured grids in reservoir simulation, with the aim of better representing the geology. We introduce `UPR` (Unstructured PEBI-grids for Reservoirs), a free, open source module for the Matlab Reservoir Simulation Toolbox. This module automates the generation of grids that conform to structures in subsurface reservoirs. The module implements the new methods presented in this thesis. These methods generate PEBI-grids that conform to wells and faults. The grids honor faults exactly. By using our novel method for treating intersections, it handles several hard cases robustly, including, intersection of multiple faults, intersections of wells and faults, and faults intersecting at sharp angles. We have also generalized our method to three dimensions, and present how one can create unstructured 3D PEBI-grids that conform exactly to faults.

Samandrag

Det er blitt gjort omfattande forsking på å generere ustrukturerte grid for reservoar simulasjoner, der målet er å forbetre representasjonen av geologien. Vi introduserer `UPR` (Unstructured PEBI-grids for Reservoirs), ein gratis, open kjeldekode modul for Matlab Reservoir Simulation Toolbox. Denne modulen automatiserer generering av grid som tilpassar seg strukturar i underjordiske reservoar. Modulen brukar dei nye metodane presentert i denne oppgåva for å generere grid som tilpassar seg eksakt til forkastingar. Ved å bruke vår nyskapande metode for å behandle kryssingar kan den generere grid av mange vanskelege tilfelle, mellom anna kryssingar av fleire forkastingar, kryssingar av brønnar og forkastingar, og forkastingar som krysser ved ein skarp vinkel. Vi har også generalisert metoden vår til å behandle forkastingar i tre dimensjonar.

Preface

This master's thesis completes my Master's degree in the field of Industrial Mathematics at NTNU, Department of Mathematical Sciences. The work in this thesis was done in the months from January 2016 to June 2016, and builds on the work done in the course TMA4500.

I would like to thank everyone at SINTEF ICT Applied Mathematics for their interest in my work. Special thanks goes to my supervisor Professor Knut-Andreas Lie, for suggesting this problem to me, and thoroughly answering my questions. Your knowledge and experience have been an invaluable asset. I would also like to thank my fellow students at Industrial Mathematics at NTNU, for keeping my motivation high. In particular, I would like to thank Øystein Strengehagen Klemetsdal for testing my routines, using his VEM implementation.

Trondheim, June 19, 2016

Runar Lie Berge

Contents

CHAPTER 1

Introduction

The energy consumption in the world is increasing every year. While new sources of renewable energy are becoming more available, oil and gas will be the dominant source of energy in the foreseeable future. Since Edwin Drake drilled the first commercial oil well, humans have found oil in more and more inaccessible places. However, despite extensive search activity, the rate of new petroleum discoveries has declined. We therefore need to utilize new and existing fields in an optimal way.

The burning of fossil fuels releases huge amounts of CO2 into the atmosphere. The climate change is maybe the largest challenge of our generation. We can already see the effects; 2015 was reported as the warmest year in recorded history [18, 23]. A proposed solution to improve the imbalance in the carbon cycle is to store CO2 in subsurface reservoirs. A concern about storing CO2 is that it might leak back to the atmosphere. Before we can try to store CO2, we therefore need a thorough understanding of how the CO2 flows in subsurface reservoirs.

The cases above have one thing in common; they need extensive mathematical modeling and an understanding of flow in porous media. A tool we have for this is flow simulations. One of the many challenges of simulating a subsurface reservoir is to transform the physical reservoir into a computational domain. A reservoir is highly inhomogeneous, and creating a grid of it is very challenging. A reservoir can contain many layers of rock with very different properties. Further, geological processes complicate matters and

can create cases such as faults. To accurately estimate the flow in a reservoir, we need to be able to represent these features in a grid. By creating a grid that conforms to features, we can also improve numerical simulations without increasing the computational cost. In modern reservoirs, wells are typical long and perforating the reservoir horizontally. The wells can have complex geometries, such as branching. The wells are an important part of the flow characteristics and it is crucial to also be able to represent them accurately.

The first attempts to simulate oil reservoirs were done in the 1950's. The early methods created 2D slices of the reservoir and created Cartesian grids of these slices. The grids would have of the order 10^2 to 10^3 cells and give very coarse approximations of the flow in a reservoir. New numerical methods and the exponentially growth in computer power have enabled us to create very complicated models of reservoirs. As an example, a modern reservoir grid can have millions of cells.

To improve the representation of the geology, corner point grids gained popularity in reservoir simulations [13, 44]. These grids are similar to 3D Cartesian grids, but the hexagons are not regular. A side of a hexagon is even allowed to have zero area. These grids are able to adapt to many of the features seen in a subsurface reservoir, but still keep the simple index relationship between cells. Corner point grids are today the industry standard, and supported by most reservoir simulation softwares.

The use of unstructured grids in reservoir simulation was introduced in the late 1980s and early 1990s [16, 21, 24, 41]. Unstructured grids are interesting because they are much more flexible than corner point grids. The earliest techniques would embed refinements in a structured background grid in areas of interest. A popular unstructured grid is the perpendicular bisector (PEBI) grid. The properties of PEBI-grids used for reservoir simulations are discussed by Verma and Aziz [51]. Courrioux et al. [9] use a PEBI-grid to create a representation of a full scale reservoir. The main drawback of these first attempts is the inability to represent complex structures, such as pinchouts and intersections of multiple faults. Later, Branets et al. [6] proposed a method that handles intersection of multiple faults, and faults intersecting at sharp angles. A similar method is also presented by Toor et al. [50]. These methods create a protection layer around the features using constrained Delaunay triangulation and recover the faults exactly. Pinchouts and intersecting faults are treated by mirroring Voronoi sites around the features. A disadvantage with these methods is that they often lead to congested Voronoi sites around the features. In an attempt to

deal with these problems, Ding and Fung [12] introduced a conflict-point removal scheme. First, a structured background grid is created. A set of Voronoi sites are placed equidistant around each fault. Each Voronoi site is given a priority, and when two sites are too close, the site with lowest priority is removed. The generated grid conforms to faults and has fairly uniform cells, but fails to treat intersections and pinchouts. A different approach is taken by Merland et al. [37, 38], who suggest to place the Voronoi sites by an optimization method that minimizes the volume of the cells that are cut in two by a fault. This method is promising, but one often needs to treat the grid manually after the optimization. Especially cells at fault intersections can be bad.

PEBI-grids are not the only unstructured grids that have received attention. There have been several attempts at creating triangular grids that adapt to faults and fractures. Brewer et al. [7] present a method for exactly representing fractures by a triangulation. Methods for approximating faults and fractures by triangles have also been investigated [25, 39]. Another method that has gained popularity in the latest years is the cut-cell method [19, 20, 35]. This method generates a grid by creating a mapping from a Cartesian grid to the physical domain, and then creates general polyhedrons by cutting the cells by crossing faults.

A great software for simulating oil reservoirs is the Matlab Reservoir Simulation Toolbox (MRST) [32]. MRST is a Matlab toolbox that allows for rapid prototyping of new ideas in reservoir simulation. The software is released open source. The main goal of this thesis has been to develop a module for unstructured gridding in MRST. As of today, MRST only offers limited automation for unstructured gridding, and by expanding these features a large part of the researchers working with reservoir simulations can take advantage of an even more powerful software. We will create routines for unstructured gridding in both 2D and 3D. The gridding routines must have the ability to conform to structures. We will look at two different conformity requirements: (i) structures that should be traced by faces of the grid, and (ii) structures that should be traced by cell centroids. Typically, wells should be traced by cell centroids, whereas internal boundaries should be traced by faces.

In a discrete fracture model (DFM), fractures are represented explicitly by setting the permeability higher in the fractures. DFM is an excellent example where the use of grids adapting to structures (in this case fractures) are useful. When modeling the fractures we need to represent them, either by grid cells or grid faces. MRST already has a DFM module which was

created by Tor Sandve and Eirik Keilegavlen, and our module works great together with this. The performance of multi-point flux approximation for DFM is discussed in [45, 46].

The rest of this thesis is organized as follows. Chapter 2 goes through the needed background to understand PEBI-grids. We present two optimization algorithms, one that places a set of points along a line path, whereas the other is the well known Limited-Memory BFGS algorithm. In Chapter 3, we discuss two methods for creating an optimal Voronoi diagram. We also introduce the notion of clipped Voronoi diagrams. A method for creating a clipped Voronoi diagram is then presented. Chapter 4 presents the algorithms used to create grids that conform to faults and wells, whereas Chapter 5 discusses the most important features of the implementation of these algorithms. Chapter 6 shows some numerical experiments. We create some example grids and compare them to grids found in the literature. Finally, we give a summary and draw conclusions in Chapter 7.

CHAPTER 2

Background

2.1 2D Delaunay Triangulation

This thesis is the continuation of the work I did in my specialization project [4]. The specialization project discussed 2D Delaunay triangulations in details, and the presentation of this section will follow that of the specialization project.

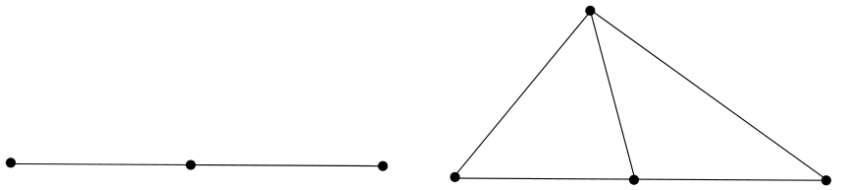
A *simplex* is a generalization of the notation of a triangle and tetrahedron to arbitary dimensions. In 2D, we define three different types of of simplices: a point, a line segment, and an ordinary triangle. Later, we will look at simplices in any dimensions, but for now we stay in 2D. One of the reasons for this generalization is to simplify notation and theorems.

Definition (Triangulation)**.** A triangulation $\mathcal{T}$ of the finite point set P is the set of simplices $\mathcal{T}$, such that

i) The set of vertices in $\mathcal{T}$ equals the point set P.

ii) The convex hull of P equals the union of all simplices in $\mathcal{T}$.

Shewchuk et al. [47] show that there exists a triangulation for any point set. Note that this does not necessary imply the existence of any triangles in $\mathcal{T}$ as shown in Figure 2.1. In the following we will assume that all sets are finite unless otherwise is said. The convex hull of a point set $\{\boldsymbol{p}_i\}_{i=1\ldots n} = P$



(a) (b)

Figure 2.1: (a) A triangulation of three points. This triangulation contains three vertices, two edges, but no triangles. (b) A triangulation of four points.

is the smallest convex set containing all the elements in P. Formally, we define it as

$$K(P) = \left\{ \sum_{i=1}^{n} \lambda_i \boldsymbol{p_i} : \ \boldsymbol{p_i} \in P, \ \lambda_i \geq 0, \ \sum_{i=1}^{n} \lambda_i = 1 \right\}.$$

A point set does in general have several possible triangulations. The left triangulation shown in Figure 2.2 is of a very special class, called the *Delaunay triangulation*. It was introduced in 1934 by the Russian Boris N. Delaunay [11], and is the most popular of all triangulations. The Delaunay triangulation has several nice properties. Arguably, the most important is that it maximizes the minimum angle of the triangles. Before we give the definition of a Delaunay triangulation, we will define what it means for a simplex to be *Delaunay* [47]:

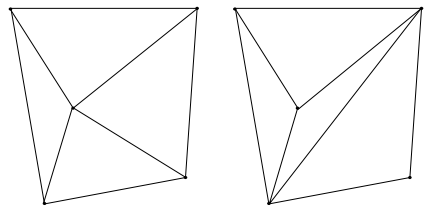


Figure 2.2: Two different triangulations of the same five points. The left triangulation is Delaunay, while the right is not.

Definition 2.1. Let $\mathcal{T}$ be a triangulation of the point set P. We say that a triangle in $\mathcal{T}$ is Delaunay if the interior of the unique circle intersecting the vertices of the triangle does not contain any points in P (the empty circumcircle property). We say that an edge in $\mathcal{T}$ is Delaunay if the interior of *some* circle intersecting its vertices does not contain any points from P. Finally, any vertex in $\mathcal{T}$ is Delaunay.

The definition of a Delaunay triangulation is just an extension of this definition and follows naturally:

Definition 2.2 (Delaunay triangulation)**.** A triangulation $\mathcal{T}$ of a point set P is a *Delaunay triangulation* if all simplices in $\mathcal{T}$ are Delaunay.

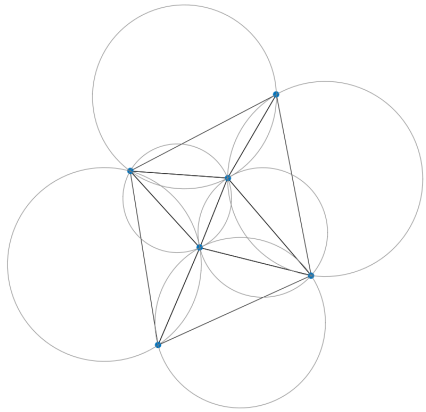


Figure 2.3: A Delaunay triangulation of a point set. The gray circles demonstrate the empty circumcircle principle for Delaunay triangulation; the interior of each circle should not contain any vertices.

An example of a Delaunay triangulation and the empty circumcircle property is shown in Figure 2.3. A question one might ask oneself is if there exists a Delaunay triangulation for all point sets. It turns out, not only does it exist, it is also unique up to degenerate points. A proof of existence and uniqueness is given by Shewchuk et al. [47]. The proof builds on the *Delaunay lemma* proved in Delaunay's original paper.

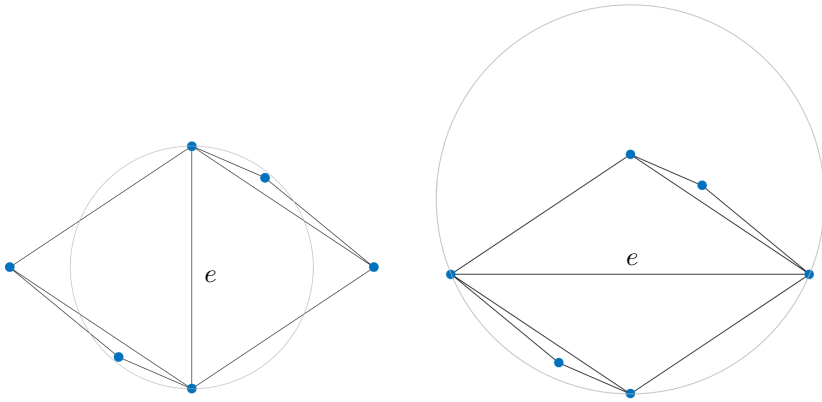
Definition 2.3 (Locally Delaunay)**.** Let e be an edge in a triangulation $\mathcal{T}$. If e is the edge of one or fewer triangles, it is *locally Delaunay*. If e is the edge of the two triangles t_1 and t_2, we say that e is locally Delaunay if the interior of some circumcircle of e does not contain any vertices from t_1 or t_2.

Note that a locally Delaunay edge is not necessary Delaunay. An example, where an edge is locally Delaunay, but not Delaunay is shown in Figure 2.4. The implication is true the other direction, per definition are all Delaunay edges also locally Delaunay.

Lemma 2.1 (Delaunay Lemma)**.** *Let $\mathcal{T}$ be a triangulation of the point set P in the plane. The following statements are equivalent*

i) *$\mathcal{T}$ is a Delaunay triangulation.*

ii) *All triangles in $\mathcal{T}$ are Delaunay.*



(a) (b)

Figure 2.4: A triangulation of four points. The edge e in (a) is locally Delaunay, while the edge e in (b) is not. Note that the edge e is not Delaunay in either cases.

iii) *All edges in $\mathcal{T}$ are Delaunay.*

iv) *All edges in $\mathcal{T}$ are locally Delaunay.*

As mentioned, the Delaunay triangulation is unique up to degenerate points. We say that four points are degenerate if they all lie on the same circumcircle. The simplest example is to let P be the corners of the unit square. Regardless which way you draw the diagonal, you have a valid Delaunay triangulation.

Theorem 2.1 (Uniqueness of the Delaunay Triangulation)**.** *Let P be a point set in the plane. Suppose no four points in P lie on the same circumcircle. Then P has a unique Delaunay triangulation.*

The requirement that no four points can lie on the same circumcircle is slightly stronger than the empty circumcircle property. We will generalize this requirement to all types of simplices, and call this *strong* Delaunay.

Definition 2.4 (Strong Delaunay)**.** Let P be a point set in the plane. We say that a *triangle* t is strong Delaunay if the vertices in t are points in P. Further, the *closed* circumdisk intersecting the vertices of t does not contain any points in P, except the vertices of t. We say that an *edge* e is strong Delaunay if the endpoints of e are points in P, and some *closed* circumdisk of e does not contain any points from P, except the endpoints of e. Finally, any point in P is strong Delaunay.

As a consequence of the uniqueness of the Delaunay triangulation, Shewchuk et al. [47] prove the following proposition

Proposition 2.1. *Every Delaunay triangulation of a point set contains all strong Delaunay simplices.*

There are several algorithms for finding the Delaunay triangulation of a set of points. Dufourd and Bertot [15] use the Delaunay lemma to create an edge-flipping algorithm and give a proof of correctness. Guibas et al. [22] present a randomize incremental algorithm for constructing Delaunay triangulations, while Cignoni et al. [8] present a divide and conquer algorithm. All Delaunay triangulations in this thesis are made by the Matlab function `delaunay`.

2.2 Higher Dimension Delaunay Triangulation

In Section 2.1, we defined a triangulation of a point set as a simplical complex whose vertices are points in the point set, and the convex hull of the point sett equals the union of all simplices. This definition generalizes to any dimension. We define the k-simplex to be the convex hull of $k+1$ affinely independent points. E.g., the 2-simplex is a triangle, and the 3-simplex is a tetrahedron. Let P be a point set in $\mathbb{R}^d$. The triangulation of P consists of the k-simplices for $k \leq d$ with vertices equal points in P and union equal the convex hull of P. If all points in P are on a line, the triangulation will only contain lines (1-simplices) and vertices (0-simplices). If the points lie in a plane, the triangulation will also contain triangles. In general, if the affine hull of P has k dimensions, the triangulation will contain at least one k-simplex, but no higher orders. The affine hull of P is the smallest affine set containing P,

$$A(P) = \left\{ \sum_{i=1}^{k} \lambda_i \boldsymbol{p}_i \ : \quad k > 0,\ \boldsymbol{p}_i \in P,\ \lambda \in \mathbb{R},\ \sum_{i=1}^{k} \lambda_i = 1 \right\}.$$

The generalization of Delaunay triangulation to higher dimensions is straight forward. We will sum up the most important properties, as they are presented by Shewchuk et al. [47]. Let P be a point set in $\mathbb{R}^d$. A simplex is Delaunay if an open circumball of the simplex does not contain any points in P. Equivalently to 2D, a simplex is strongly Delaunay if the

closed circumball does not contain any points in P, except the vertices of the simplex.

Definition 2.5 (Delaunay Triangulation)**.** Let $\mathcal{T}$ be a triangulation of the point set P in $\mathbb{R}^d$. The dimension of the affine hull of P is $k \leq d$. The triangulation $\mathcal{T}$ is Delaunay if all k-simplices in $\mathcal{T}$ are Delaunay.

Remember that the Delaunay triangulation in the plane is unique if no four points in P lie on the same circumcircle. For higher dimensions this property generalizes, and we call the set P *generic*.

Definition 2.6 (Generic)**.** Let P be a point set in $\mathbb{R}^d$. If the dimension of the affine hull of P is k, the set P is said to be *generic* if no $k+2$ points lie on the same ball.

As for the Delaunay triangulation in the plane, the Delaunay triangulation in higher dimensions is unique if the point set is generic.

Proposition 2.2. *Let P be a point set in $\mathbb{R}^d$. If P is generic, there exists exactly one Delaunay triangulation of the set.*

Shewchuk et al. [47] give several other properties of higher-order Delaunay triangulations. We will, however, only need one more property

Theorem 2.2. *Let $\mathcal{T}$ be a Delauany triangulation of the point set P. All strongly Delaunay simplices of P are in the triangulation $\mathcal{T}$.*

2.3 Voronoi Diagrams

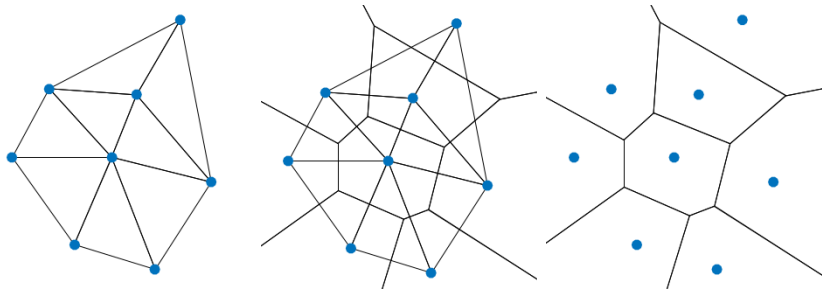
A grid that is closely related to the Delaunay triangulation is the *Voronoi diagram*. The Voronoi diagram has been known for several centuries, but was first formally described in Georges Voronoi's paper from 1908 [52].

Definition 2.7 (Voronoi Diagram)**.** Let $P = \{\boldsymbol{p}_i\}_{i=1\ldots n}$ be a point set in $\mathbb{R}^d$. We call each point for a *site*. We say that a point $\boldsymbol{x}$ belongs to the *Voronoi cell* v_{p_i}, if it is at least as close to $\boldsymbol{p}_i$ as any other sites in P. That is,

$$v_{p_i} = \{\boldsymbol{x} : \quad \boldsymbol{x} \in \mathbb{R}^d,\ |\boldsymbol{x} - \boldsymbol{p}_i| \leq |\boldsymbol{x} - \boldsymbol{p}_j| \ \ j = 1 \ldots n\}.$$

The *Voronoi diagram* $\mathcal{V}$ is the set of all Voronoi cells v_p.

Note that in the Voronoi diagram shown in Figure 2.5, some of the cells extend to infinity. This is true in general; the Voronoi diagram will contain all points in $\mathbb{R}^d$, because all points have at least one closest site in P.



(a) Delaunay triangulation. (b) Delaunay triangulation and Voronoi diagram. (c) Voronoi Diagram.

Figure 2.5: The Delaunay triangulation and the dual Voronoi diagram of the same point set.

The faces of a Voronoi cell are the points that are shared by one or more cells. The faces that have dimension $d-1$, 2, 1, and 0 are called for Voronoi facets, Voronoi polygons, Voronoi edges, and Voronoi vertices, respectively. To simplify the notation in the following theorems, we also include the cell itself as a face. All faces can be described by the intersection of Voronoi cells. We define the intersection of the cells $v_{p_1}, \ldots, v_{p_j}$ as

$$v_{p_1 \ldots p_j} = v_{p_1} \cap \ldots \cap v_{p_j}.$$

If v_{pq} is none-empty, the points in v_{pq} are equidistant from both sites p and q, and at least as close to them as to any other site. In other words, the open ball centered at any point in v_{pq} with p and q on the boundary, does not contain any sites from P. If P is generic, v_{pq} will have dimension $d-1$. In general, the intersection of j cells will have dimension $d+1-j$, but if P is not generic it can be higher or lower [47]. In the 2D Voronoi diagram in Figure 2.5, the sites are generic, thus, the intersection of two cells is an edge, and the intersection of three cells is a vertex.

There is a close relationship between the Delaunay triangulation and Voronoi diagram. They are often called dual of each other, in the sense that the topology of one is defined by the topology of the other. The duality is defined by a bijection between the faces of the Delaunay triangulation and the faces of the Voronoi diagram. We state this duality precisely in the following theorem [47].

Theorem 2.3 (Duality of Delaunay triangulation and Voronoi diagram)**.** *Let P be a generic point set in $\mathbb{R}^d$. Let $\mathcal{V}$ and $\mathcal{T}$ be the associated Voronoi diagram and Delaunay triangulation, respectively. Let $S = \{\boldsymbol{s}_1, \dots \boldsymbol{s}_j\} \subseteq P$ be a subset of the sites in P. The convex hull of S is a k-face of $\mathcal{T}$ if and only if $v_{s_1,\dots s_j}$ is a $(d-k)$-face of $\mathcal{V}$.*

Proof. First, assume that the convex hull of S is a k-face of $\mathcal{T}$. Then there exists a closed ball B that intersects $\boldsymbol{s}_1, \dots, \boldsymbol{s}_j$, but does not contain any sites from $P \setminus S$. The center of this ball is equidistant to all sites in S, hence, the intersection $v_{s_1 \dots s_j}$ is not empty; i.e., it is a Voronoi face of P. Let Π be the affine space that is orthogonal to the affine space of S and contains the center of B. The space Π has dimension $(d-k)$ because the dimension of $A(S)$ is k. All points in Π are equidistant to all sites in S, and no points in $\mathbb{R}^d \setminus \Pi$ are equidistant to all sites in S, thus, $v_{s_1 \dots s_j} \subseteq \Pi$. Let $0 < \varepsilon = \min_{\boldsymbol{p} \in P \setminus S} d(B, \boldsymbol{p})$ be the minimum distance from the ball B to any sites in $P \setminus S$. Any points in Π that are closer to the center of B than $\frac{1}{2}\varepsilon$ are on the face $v_{s_1 \dots s_j}$, hence, the dimension of $v_{s_1 \dots s_j}$ is the same as Π, that is $(d-k)$.

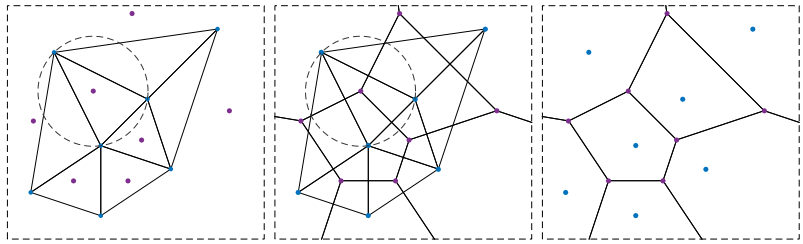
Now assume that $v_{s_1 \dots s_j}$ is a Voronoi $(d-k)$-face. Since P is generic, there is no $s_{j+1} \in P \setminus S$ such that $v_{s_1 \dots s_j} = v_{s_1 \dots s_j s_{j+1}}$. In fact, the number of cells must equal $j = k+1$ if $v_{s_1 \dots s_j}$ is to have dimension $(d-k)$. We can therefore find a closed ball centered at some point in $v_{s_1 \dots s_j}$ that has $s_1, \dots, s_j$ on its boundary and does not contain any sites $P \setminus S$. The convex hull of the $k+1$ sites in S is a k-simplex and it is strongly Delaunay, hence, it is a k-face in the Delaunay triangulation.

□

The main results of the duality theorem is for $j = 2$ and $j = d+1$. For $j = 2$ the theorem says that Voronoi cell v_{s_1} and v_{s_2} share a Voronoi facet if and only if there is a Delauany edge between site s_1 and s_2. For $j = d+1$ the theorem says that all Voronoi vertices are the center of a circumball of a Delaunay $(d+1)$-simplex. Figures 2.5 and 2.6 show the duality in 2D.

Heinemann et al. [24] introduced Voronoi diagrams to reservoir simulations in 1991. In reservoir simulation it is common to call Voronoi diagrams for PEBI-grids (PErpendicular BIsector). We will give a new definition of Voronoi diagrams, which makes it obvious why they are also given this name.

Definition 2.8 (PEBI-Grid)**.** Let P be a finite point set in $\mathbb{R}^d$, and let $\mathcal{B}$ be its PEBI-grid. A site $\boldsymbol{p}_i \in P$ generates a cell b_{p_i} in the following way. Create the perpendicular bisector planes of $\boldsymbol{p}_i$ and all other sites in P. Each



(a) Delaunay triangulation. (b) Delaunay triangulation and Voronoi diagram (c) Voronoi Diagram.

Figure 2.6: The duality of a Delaunay triangulation and a Voronoi diagram in 2D. Delaunay vertices correspond to Voronoi cells (blue points). Delaunay edges are perpendicular to the corresponding Voronoi edges. The circumcenter of a Delaunay triangle corresponds to a Voronoi vertex (purple points).

bisector cuts $\mathbb{R}^d$ in two. Let $H(\boldsymbol{p}_i, \boldsymbol{p}_j)$ be the half-space that contains the site $\boldsymbol{p}_i$. The cell b_{p_i} is the intersection of all of these half-spaces,

$$b_{p_i} = \bigcap_{\boldsymbol{p}_j \in P \setminus \boldsymbol{p}_i} H(\boldsymbol{p}_i, \boldsymbol{p}_j).$$

That $\mathcal{B}$ in fact is a Voronoi diagram is easily shown. Consider any point $\boldsymbol{x} \in \mathbb{R}^d$, and suppose it belongs to cell v_{p_i} in the Voronoi diagram. Since it is at least as close to $\boldsymbol{p}_i$ as any other point $\boldsymbol{p}_j$, it will be in the half-space $H(\boldsymbol{p}_i, \boldsymbol{p}_j)$ for all j. Hence, $\boldsymbol{x}$ is in the intersection of all of these half-spaces. Similarly, if $\boldsymbol{x}$ is in the PEBI-cell b_{p_i}, it is in all half spaces $H(\boldsymbol{p}_i, \boldsymbol{p}_j)$. That it is in the half space $H(\boldsymbol{p}_i, \boldsymbol{p}_j)$ means that it is closer to $\boldsymbol{p}_i$ than $\boldsymbol{p}_j$. This is valid for all j, hence, it is in the Voronoi cell v_{p_i}.

Because of the close connection between Voronoi diagrams and Delaunay triangulations, we expect there to be a connection between the definition of PEBI-grids and Delaunay triangulations. It turns out that the only perpendicular bisectors we need to create a PEBI-cell b_{p_i}, are of those sites connected to $\boldsymbol{p}_i$ by edges in the Delaunay triangulation.

Proposition 2.3. *Let $\mathcal{T}$ and $\mathcal{B}$ be the Delaunay triangulation and the PEBI-grid of a point set P. Let $\boldsymbol{p}_i$ be a vertex in the triangulation $\mathcal{T}$, and $\boldsymbol{p}_i^1, \boldsymbol{p}_i^2, \ldots, \boldsymbol{p}_i^k$ be all vertices connected to $\boldsymbol{p}_i$ by an edge in $\mathcal{T}$. Let $\tilde{b}_{p_i} = H(\boldsymbol{p}_i, \boldsymbol{p}_i^1) \cap \ldots \cap H(\boldsymbol{p}_i, \boldsymbol{p}_i^k)$ be the cell created from $\boldsymbol{p}_i$ and its Delaunay neighbors. Then $\tilde{b}_{p_i}$ equals the PEBI-cell b_{p_i}.*

Proof. We will only give a proof for when P is generic, but it can be

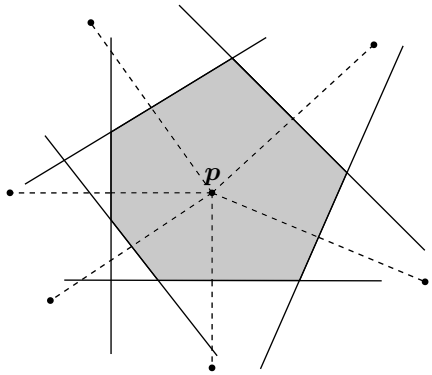


Figure 2.7: Creation of a PEBI cell given a Delaunay triangulation. The dotted lines are edges in the Delaunay triangulations connected to the site $\boldsymbol{p}$. The solid lines are bisectors of these edges. The shaded area is the PEBI-cell b_p.

extended to non-generic cases also. Assume $v_{p_i p_l} = \emptyset$, that is, cell $\boldsymbol{p}_i$ and $\boldsymbol{p}_l$ are not neighbors. The half-space $H(\boldsymbol{p}_i, \boldsymbol{p}_l)$ can not be a restriction when creating the cell v_{p_i}. If it was, there would be a point on the bisector of $\boldsymbol{p}_i$ and $\boldsymbol{p}_l$ that is closer to the two sites than any other sites. This is a contradiction since we assumed $v_{p_i p_l} = \emptyset$. We can therefore remove the half-space from the intersection without affecting the cell $v_{p_i} = \bigcap_{\boldsymbol{p}_j \in P \setminus \boldsymbol{p}_i} H(\boldsymbol{p}_i, \boldsymbol{p}_j) = \bigcap_{\boldsymbol{p}_j \in P \setminus \{\boldsymbol{p}_i, \boldsymbol{p}_l\}} H(\boldsymbol{p}_i, \boldsymbol{p}_j)$. In a similar way, we can remove all sites that are not neighbors of v_{p_i}. Let $S = \{\boldsymbol{s}_1, \dots \boldsymbol{s}_j\} = \{\boldsymbol{s} \in P \setminus \boldsymbol{p}_i \ : \ v_{p_i s} \neq \emptyset\}$. We can define the PEBI-cell as $b_{p_i} = \bigcap_{\boldsymbol{p}_j \in P \setminus \boldsymbol{p}_i} H(\boldsymbol{p}_i, \boldsymbol{p}_j) = \bigcap_{\boldsymbol{p}_j \in S} H(\boldsymbol{p}_i, \boldsymbol{p}_j)$. All the faces $v_{p_i s_1}, v_{p_i, s_2}, \dots, v_{p_i, s_j}$ are $(d-1)$ faces of the Voronoi diagram $\mathcal{B}$, and by the duality theorem, are all edges between $\boldsymbol{p}_i$ and $\boldsymbol{s} \in S$ edges in the Delaunay triangulation.

□

Figure 2.7 shows the construction of a PEBI-cell from a Delaunay triangulation.

The most popular softwares for creating Voronoi diagrams are arguably Qhull [3] and CGAL [27]. Most of the 2D Voronoi diagrams in this thesis are created by the MRST function `pebi`. This function creates the Voronoi diagram as the dual of the Delaunay triangulation. In Section 3.4 we will discuss clipped Voronoi diagrams. To create such diagrams, we have implemented our own routine `clippedPebi2D`. This routine generates Voronoi diagrams by calculating the half-space intersections from the PEBI-grid definition. The 3D Voronoi diagrams are created using Qhull.

2.4 Limited-Memory BFGS

One of the most popular quasi-Newton algorithms is the Limited-Memory BFGS (L-BFGS) method. It is simple and easy to implement (our Matlab implementation is less than 80 lines of code). The L-BFGS method is a variant of the BFGS method. We will explain the differences later, but the big advantage of quasi-Newton methods, compared to Newton's method, is that we never have to calculate the Hessian of the objective function. Instead, it is approximated by looking at the curvature of the previous steps taken. We will give a brief summary of the L-BFGS method. A detailed derivation is given by Nocedal and Wright [40].

A general unconstrained optimization problem can be formulated as

$$O = \min_{\boldsymbol{x} \in \mathbb{R}^d} f(\boldsymbol{x}),$$

where $f(\cdot)$ is a function mapping from $\mathbb{R}^d$ to $\mathbb{R}$. The point that obtains the optimal value O is called the optimal point $\boldsymbol{x}^*$. Finding an analytic solution to an optimization problem can be extremely hard, even impossible, and numerical methods must therefore be used. A well known class of numerical methods for solving an optimization problem is the quasi-Newton methods. The methods are iterative, and require the evaluation of both the objective function and its gradient. At each iterate, the objective function is approximated by a quadratic function

$$m_k(\boldsymbol{p}) = f(\boldsymbol{x}_k) + \nabla f(\boldsymbol{x}_k)^\top \boldsymbol{p} + \frac{1}{2}\boldsymbol{p}^\top H_k^{-1}\boldsymbol{p}, \tag{2.1}$$

where H_k^{-1} is an approximation of the Hessian at the point $\boldsymbol{x}_k$. The minimizer of the quadratic approximation $m_k(\cdot)$ is

$$\boldsymbol{p}_k^* = -H_k \nabla f(\boldsymbol{x}_k). \tag{2.2}$$

This optimum is used as a search direction for the new iterate

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha \boldsymbol{p}_k^*.$$

The constant α_k is a step length parameter chosen to satisfy the Wolfe conditions [53][40]:

$$\begin{aligned} f(\boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k) &\leq f(\boldsymbol{x}_k) + c_1 \alpha_k \nabla f(\boldsymbol{x}_k)^\top \boldsymbol{p}_k, \\ \nabla f(\boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k)^\top \boldsymbol{p}_k &\geq c_2 \nabla f(\boldsymbol{x}_k)^\top \boldsymbol{p}_k, \end{aligned}$$

with $0 < c_1 < c_2 < 1$. If H_k^{-1} is chosen to be the exact Hessian at each iteration, we obtain Newton's method. The beauty of the quasi-Netwon methods is that we do not need to compute the Hessian, but still obtain superlinear convergence.

Arguably, the most popular quasi-Newton method is the BFGS method. Instead of approximating the Hessian, such as the DFP method [10], it estimates the inverse Hessian directly. We then avoid inverting any matrices and only need to calculate matrix vector products. Define $\boldsymbol{s}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$, $\boldsymbol{y}_k = \nabla f(\boldsymbol{x}_{k+1}) - \nabla f(\boldsymbol{x}_k)$ and $\rho_k = (\boldsymbol{y}_k^\top \boldsymbol{s}_k)^{-1}$. At each step we update the inverse Hessian as

$$H_{k+1} = (I - \rho_k \boldsymbol{s}_k \boldsymbol{y}_k^\top) H_k (I - \rho_k \boldsymbol{y}_k \boldsymbol{s}_k^\top) + \rho_k \boldsymbol{s}_k \boldsymbol{s}_k^\top. \tag{2.3}$$

We need to specify some initial H_0 in order to solve the first step. There is not one H_0 that works best in all cases, the choice is very problem dependent. A common choice is the identity matrix, or some scaled version of it.

For large systems, the cost of storing and computing the Hessian approximation H_k might be very large. As a solution to this problem the L-BFGS method is introduced. The BFGS method uses all previous iteration steps to approximate the inverse Hessian, but is this necessary? As the name suggest, the L-BFGS method only remembers the last m iterations. It uses these steps to update the inverse Hessian in the same way as BFGS. This means that L-BFGS and BFGS take exactly the same steps the first m iterations, but after that they will start to differ. A larger m will usually result in fewer iterations to convergence, but it is a trade-off with increased computational cost at each iteration. In practice m, is usually set to a value between 3 and 20.

The inverse Hessian approximation H_k will in general be dense. Instead of storing the full matrix, we store the latest m vectors $\{\boldsymbol{s}_i\}_{i=k-m}^{k-1}$ and $\{\boldsymbol{y}_i\}_{i=k-m}^{k-1}$ and reconstruct the inverse Hessian H from these. We define $V_k = I - \rho_k \boldsymbol{y}_k \boldsymbol{s}_k^\top$. By unraveling the approximated inverse Hessian from Equation (2.3) we can express the L-BFGS approximation as a sum of

vector and matrix products

$$\begin{aligned} H_k = &(V_{k-1}^\top \dots V_{k-m}^\top) H_k^0 (V_{k-m}^\top \dots V_{k-1}^\top) \\ &+\rho_{k-m}(V_{k-1}^\top \dots V_{k-m+1}^\top)\boldsymbol{s}_{k-m}\boldsymbol{s}_{k-m}^\top(V_{k-m+1}^\top \dots V_{k-1}^\top) \\ &+\rho_{k-m}(V_{k-1}^\top \dots V_{k-m+2}^\top)\boldsymbol{s}_{k-m+1}\boldsymbol{s}_{k-m+1}^\top(V_{k-m+2}^\top \dots V_{k-1}^\top) \\ &\vdots \\ &+\rho_{k-1}\boldsymbol{s}_{k-1}\boldsymbol{s}_{k-1}^\top. \end{aligned}$$

As for the BFGS method we need to choose an initial H_0, however, we allow $H_0 = H_k^0$ to vary at each iteration. It is not trivial to choose a H_k^0, but a method that has proven effective in general is to choose $H_k^0 = \frac{\boldsymbol{s}_{k-1}^\top \boldsymbol{y}_{k-1}}{\boldsymbol{y}_{k-1}^\top \boldsymbol{y}_{k-1}} I$. This scaled version of the identity matrix tries to approximate the size of the true Hessian. By this choice of H_k^0, the search direction $\boldsymbol{p}_k^*$ will be reasonable scaled, and a step length of $\alpha_k = 1$ will be accepted in most cases.

2.5 Intersection of Geometrical Objects

In this section, we will compute the intersection of various geometrical objects. These intersections are used by the algorithms described later in this thesis. Specifically, the line-plane intersection is used by the polygon clipping algorithm in Section 3.3, and the circle and sphere intersections are used by the fault conformity algorithms in Chapter 4.

Line-Plane Intersection

The intersection of a line segment and a plane is easily calculated. Let the line segment be defined by its two endpoints $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$. We describe the plane by its normal vector $\boldsymbol{n}$ and a point $\boldsymbol{x}$ on the plane. Let $\boldsymbol{I}$ be the intersection point of the line segment and the plane. The two triangles in Figure 2.8 are similar, thus, the ratios of the side lengths are equal

$$\frac{d(\boldsymbol{v}_1, \boldsymbol{I})}{d(\boldsymbol{v}_1, \boldsymbol{r}_1)} = \frac{d(\boldsymbol{v}_2, \boldsymbol{I})}{d(\boldsymbol{v}_2, \boldsymbol{r}_2)}. \tag{2.4}$$

The points $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$ can be found by projecting $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ onto the plane. We define t as the relative length from $\boldsymbol{v}_1$ to the intersection

$$t = \frac{d(\boldsymbol{v}_1, \boldsymbol{I})}{d(\boldsymbol{v}_1, \boldsymbol{I}) + d(\boldsymbol{v}_2, \boldsymbol{I})}.$$

Dividing over and under by the length from $\boldsymbol{v}_2$ to the intersection yields,

$$t = \frac{d(\boldsymbol{v}_1, \boldsymbol{I})/d(\boldsymbol{v}_2, \boldsymbol{I})}{(d(\boldsymbol{v}_1, \boldsymbol{I}) + d(\boldsymbol{v}_2, I))/d(\boldsymbol{v}_2, \boldsymbol{I})} = \frac{d(\boldsymbol{v}_1, \boldsymbol{r}_1)}{d(\boldsymbol{v}_1, \boldsymbol{r}_1) + d(\boldsymbol{v}_2 \boldsymbol{r}_2)}. \tag{2.5}$$

Here, we have used the identity from Equation (2.4) to simplify the expression. The intersection is found by taking a t step length along the line segment:

$$\boldsymbol{I} = t(\boldsymbol{v}_2 - \boldsymbol{v}_1) + \boldsymbol{v}_1.$$

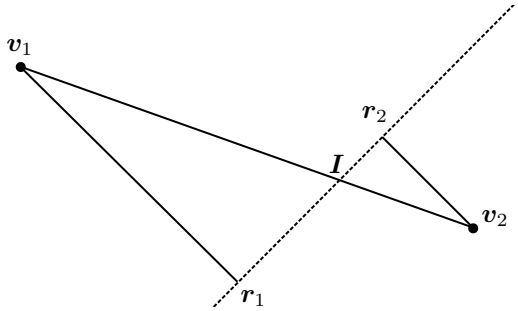


Figure 2.8: The two triangles, $(\boldsymbol{v}_1 \boldsymbol{r}_1 \boldsymbol{I})$ and $(\boldsymbol{v}_2 \boldsymbol{r}_2 \boldsymbol{I})$ are similar.

There are a few special cases that the above calculation does not take into consideration. If $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ are on the same side of the plane, the intersection is empty. If $d(\boldsymbol{v}_1, \boldsymbol{r}_1) = d(\boldsymbol{v}_2, \boldsymbol{r}_2) = 0$ the line coincides with the plane, and the intersection is the whole line. If $d(\boldsymbol{v}_1, \boldsymbol{r}_1) = 0$ and $d(\boldsymbol{v}_2, \boldsymbol{r}_2) \neq 0$, the intersection is at the end point $\boldsymbol{v}_1$ and vice versa.

Circle-Circle Intersection

The intersection of two circles is either two real points, one real point, or two imaginary points. For notation see Figure 2.9. The known variables are the circle centers $\boldsymbol{C}_0$ and $\boldsymbol{C}_1$ together with the circle radii R_0 and R_1. The distance d between the circles is implicitly given. First, we shift and rotate our coordinate system such that the first circle is centered at the origin and the second circle is centered at $(d, 0)$. The intersection points in the new coordinate system will have coordinates $(a, \pm h)$. At the intersection, the equations for both circles

$$\begin{aligned} R_0^2 &= a^2 + h^2 \\ R_1^2 &= (a - d)^2 + h^2 = a^2 - 2ad + d^2 + h, \end{aligned}$$

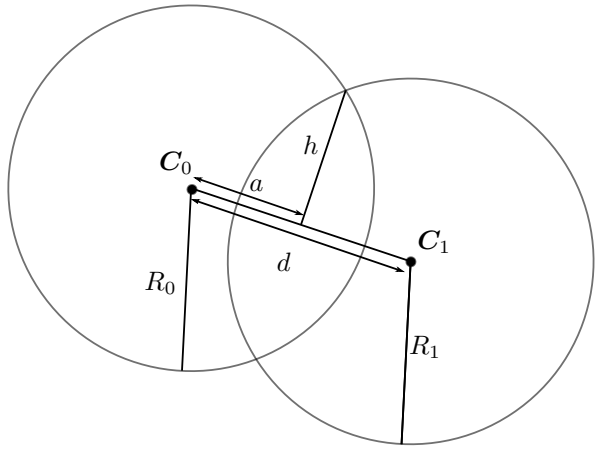


Figure 2.9: Intersection of two circles with centers in $\boldsymbol{C}_1$ and $\boldsymbol{C}_2$ and radii R_1 and R_2.

must be satisfied. If $d = 0$ the intersection is either empty, or $R_1 = R_2$ and the circles coincide. Otherwise, we substitute the first equation into the second and obtain an expression for a

$$a = \frac{d^2 + R_0^2 - R_1^2}{2d}.$$

Putting this back into the first equation we get

$$h = \pm\sqrt{R_0^2 - a^2}.$$

If $a^2 < R_0^2$ we have to real solutions. If $a^2 = R_0^2$ there is only one intersection point, and if $a^2 > R_0^2$ the intersection points are imaginary.

Let $n_\parallel = \frac{\boldsymbol{C}_0 - \boldsymbol{C}_1}{|\boldsymbol{C}_0 - \boldsymbol{C}_1|}$ be the normal vector pointing from $\boldsymbol{C}_0$ to $\boldsymbol{C}_1$ and $\boldsymbol{n}_\perp$ be the vector orthonormal to $\boldsymbol{n}_\parallel$. Transforming back to the original coordinate system, we obtain the intersection points on vector format

$$\boldsymbol{p}_{int} = \boldsymbol{C}_0 + a\boldsymbol{n}_\parallel \pm h\boldsymbol{n}_\perp.$$

Sphere Intersections

Suppose we have three spheres centered at $\boldsymbol{C}_0$, $\boldsymbol{C}_1$, and $\boldsymbol{C}_2$ with radii R_0, R_1, and R_2. To calculate their intersection we rotate and shift our coordi-

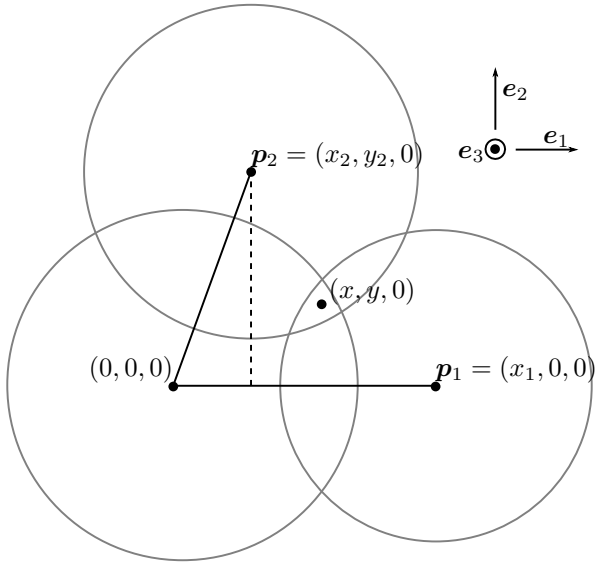


Figure 2.10: The cross section of three spheres. The spheres are centered at origin, $\boldsymbol{p}_1$, and $\boldsymbol{p}_2$. The point $(x, y, 0)$ is the projection of the intersection into the xy-plane.

nate system such that $\boldsymbol{C}_0$ is centered at origin, $\boldsymbol{C}_1$ is centered at $(x_1, 0, 0)$ and $\boldsymbol{C}_2$ is centered at $(x_2, y_2, 0)$. For further notation, see Figure 2.10. The orthonormal basis for our new coordinate system is

$$\begin{aligned}
\boldsymbol{e}_1 &= \frac{\boldsymbol{C}_1 - \boldsymbol{C}_0}{|\boldsymbol{C}_1 - \boldsymbol{C}_0|}, \quad \boldsymbol{v}_2 = \boldsymbol{C}_2 - \boldsymbol{C}_0 - \langle \boldsymbol{C}_2 - \boldsymbol{C}_0, \boldsymbol{e}_1 \rangle \boldsymbol{e}_1 \\
\boldsymbol{e}_2 &= \frac{\boldsymbol{v}_2}{|\boldsymbol{v}_2|} \\
\boldsymbol{e}_3 &= \boldsymbol{e}_1 \times \boldsymbol{e}_2.
\end{aligned}$$

The coordinates of the sphere centers $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ in the new coordinate system is found by first shifting the sphere centers, and then take the inner product with the three basis vectors. By construction, the z-coordinate of all sphere centers will equal 0. Equivalently will the y-coordinate of $\boldsymbol{p}_1$ is also equal 0. The other coordinates are

$$x_1 = |\boldsymbol{C}_1 - \boldsymbol{C}_0|, \quad x_2 = \langle \boldsymbol{C}_2 - \boldsymbol{C}_1, \boldsymbol{e}_1 \rangle, \quad y_2 = \langle \boldsymbol{C}_2 - \boldsymbol{C}_1, \boldsymbol{e}_2 \rangle.$$

At the intersection, all sphere equations must be satisfied:

$$\begin{aligned} R_0^2 &= x^2 + y^2 + z^2 \\ R_1^2 &= (x - x_1)^2 + y^2 + z^2 \\ R_2^2 &= (x - x_2)^2 + (y - y_2)^2 + z^2. \end{aligned}$$

Subtracting the second equation from the first we eliminate both the z and y variable

$$x = \frac{R_0^2 - R_1^2 + x_1^2}{2x_1}.$$

Notice that this equals the x-coordinate for the circle-circle intersection. We have assumed that $x_1 \neq 0$. If x_1 equals zero, sphere one and two either coincide or do not intersect. To find the y-coordinate we subtract the third equation from the second

$$\begin{aligned} 2yy_2 &= R_1^2 + x_2^2 + 2xx_1 - x_1^2 - R_2^2 + y_2^2 - xx_2, \\ y &= \frac{R_0 - R_2^2 + y_2^2}{2y_2} - \frac{x_2}{y_2}x. \end{aligned}$$

If $y_2 = 0$, the equations can only be satisfied if $R_1^2 - (x - x_1)^2 = R_2^2 - (x - x_2)^2$, i.e., the intersection is either empty or a circle.

Substituting back into the first equation we obtain the z-coordinate

$$z = \pm\sqrt{R_0^2 - x^2 - y^2}.$$

Transforming back to the original coordinate system, the sphere intersections are

$$\boldsymbol{p}_{int} = \boldsymbol{C}_0 + x\boldsymbol{e}_1 + y\boldsymbol{e}_2 \pm z\boldsymbol{e}_3.$$

2.6 Placing Points Along a Path

We wish to place a set of points $\{\boldsymbol{c}_i\}_{i=1,\ldots,n} = C \subset \mathbb{R}^d$ along a path. The desired distance between two consecutive points is given by a function $f(\cdot,\cdot) : \mathbb{R}^d \to \mathbb{R}$. We parametrize the path such that $\boldsymbol{l}(t_i) = \boldsymbol{c}_i$. Further, $\boldsymbol{l}(0)$ is the start point of the path, and $\boldsymbol{l}(T)$ is the end point of the path.

This gives the following requirement on the set of points C:

$$\int_{t_i}^{t_{i+1}} |\boldsymbol{l}'(t)| \, \mathrm{d}t = f(\boldsymbol{c}_i, \boldsymbol{c}_{i+1}) \tag{2.6}$$

If f is a constant, the points are placed equidistant along the line. In this case, Equation (2.6) reduces to

$$\int_{t_i}^{t_{i+1}} |\boldsymbol{l}'(t)| \, \mathrm{d}t = d_s,$$

where d_s is the desired distance between the points. The total length of the path is

$$S = \int_{t_1}^{t_n} |\boldsymbol{l}'(\tau)| \, \mathrm{d}\tau.$$

The number of points in the set C will change depending on the ratio of the desired distance and length of the path. To achieve the desired distance, the number of points must be $n = \frac{S}{d_s}$. In general, the fraction $\frac{S}{d_s}$ will not be a integer. There are two solutions to this problem. We can keep the desired distance d_s, but set $0 < t_1$ and $t_n < T$. The start and end points of the path will then not be in the set C. Alternatively, we can give some slack to the desired distance. We adjust the distance between the points to be a factor of the total line length S. The distance between the points will then be either slightly shorter or slightly longer than the desired distance.

We can find the points in C by using Newton's method. We wish to solve Equation (2.6) for the unknown vector $\boldsymbol{t} = [t_1, \ldots, t_n]^\top$. Remember that $\boldsymbol{c}_i = \boldsymbol{l}(t_i)$. There are n unknowns, but only $n - 1$ equations. The last equation is defined depending on the situation, e.g., $t_1 = 0$ if we want to place the first point at the start of the path. We define the vector function $\boldsymbol{g} = [g_1, \ldots, g_{n-1}]^\top$ as

$$g_i(t_1, \ldots, t_n) = \int_{t_i}^{t_{i+1}} |\boldsymbol{l}'(\tau)| \, \mathrm{d}\tau - f(\boldsymbol{l}(t_i), \boldsymbol{l}(t_{i+1})). \tag{2.7}$$

The function $\boldsymbol{g}$ is the difference between the actual distance and the desired distance. When $\boldsymbol{g} = 0$, we have a solution to Equation (2.6). By using the fundamental theorem of calculus, together with the chain rule, we obtain

the partial derivative:

$$\frac{\partial g_i}{\partial t_j} = \begin{cases} |\boldsymbol{l}'(t_{i+1})| - f'(\boldsymbol{l}(t_i), \boldsymbol{l}(t_{i+1}))\boldsymbol{l}'(t_{i+1}), & j = i+1 \\ -|\boldsymbol{l}'(t_i)| - f'(\boldsymbol{l}(t_i), \boldsymbol{l}(t_{i+1}))\boldsymbol{l}'(t_i), & j = i \\ 0, & \text{Otherwise.} \end{cases}$$

The Jacobian of the vector function $\boldsymbol{g}$ is

$$J(\boldsymbol{t}) = \begin{bmatrix} \frac{\partial g_1(\boldsymbol{t})}{\partial t_1} & \frac{\partial g_1(\boldsymbol{t})}{\partial t_2} & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \frac{\partial g_{n-1}(\boldsymbol{t})}{\partial t_{n-1}} & \frac{\partial g_{n-1}(\boldsymbol{t})}{\partial t_n} \end{bmatrix}.$$

To find $\boldsymbol{g} = 0$, we start by making an initial guess $\boldsymbol{t}^0$. Depending on the parametrization, placing $t_1, \dots t_n$ equidistant between 0 and T might be a good choice. At each iteration, the Newton's update is

$$\boldsymbol{t}^{k+1} = \boldsymbol{t}^k - J(\boldsymbol{t}^k)^{-1}\boldsymbol{g}(\boldsymbol{t}^k),$$

which we continue until convergence.

In the case where the desired distance between points is constant, the above method is straight forward. We calculate the number of points needed, make an initial guess, and run the Newton's update until convergence. To find the points for a non-constant function f is harder. Already when deciding the number of points we need, we might run into problems. For some functions f, there might be several solutions to Equation (2.6), and the number of points might be different for each solution. We will present an algorithm for placing the points along the path for a general function f. The algorithm adjusts the number of points on the fly, and does not need to evaluate the gradient of the distance function f.

Initialization

To find an initial set of points, we first assume f is constant. The desired distance between points can by found by estimating the average desired distance, e.g., set $d_s = f(\boldsymbol{l}(0), \boldsymbol{l}(T))$. The initial set of parameter values $\boldsymbol{t}^0$ is chosen such that the points points $\boldsymbol{c}_1^0, \dots, \boldsymbol{c}_{n^0}^0$ are placed equidistant along the path.

Main Loop

For each iteration we either insert a new point, remove a point, or move the points. What we do depends on the difference between the left and right side of Equation (2.6). The vector $\boldsymbol{g}^k$ is found by evaluating Equation (2.7) at iteration k. The vector measures the error of the distances between the points. If a distance error $g_i^k > 0$ is positive, it means that the distance between $\boldsymbol{c}_i$ and $\boldsymbol{c}_{i+1}$ is too long. If a distance error $g_i^k < 0$ is negative, it means that the distance between $\boldsymbol{c}_i$ and $\boldsymbol{c}_{i+1}$ is too short.

There are two cases which add or remove points.

$\sum_{i=1}^{n^k} g_i^k > \min_{i=1,\dots n^k-1} f(\boldsymbol{c}_i^k, \boldsymbol{c}_{i+1}^k)$: A point is added.

$\sum_{i=1}^{n^k} g_i^k < -\max_{i=1,\dots n^k-1} f(\boldsymbol{c}_i^k, \boldsymbol{c}_{i+1}^k)$: A point is removed.

If the sum of the distance errors is greater than the smallest desired distance, we add a point. The point is added on the path segment that has the largest distance error. We let the new point be

$$\boldsymbol{c}_{\text{new}} = \boldsymbol{l}\left(\frac{t_i^k + t_{i+1}^k}{2}\right); \quad g_i^k \geq g_j^k, \ \forall\ j.$$

If the sum of all distance errors is less than negative the longest desired distance, we remove a point. We remove a point on the path segment that is most too short; that is, the most negative g_i^k.

If no points are added or removed, we move the points based on their distance error g_i^k. If a path segment is too short, we wish it to be longer and can imagine it pushes the two points that it is connected to away. If a path segment is too long, we can imagine it pulls the two points that it is connected to towards its center. We define the force on each point as the difference of the distance errors of the two path segments the point is connected to:

$$F_i^k = (g_i^k - g_{i-1}^k).$$

If $F_i^k > 0$ we wish to move point $\boldsymbol{c}_i^k$ towards the end of the path. If $F_i^k < 0$ we wish to move point $\boldsymbol{c}_i^k$ towards the start of the path. The force also gives a good estimate for how long we wish to move the points. However, we are only able to move the points through the parametrization $\boldsymbol{l}(\cdot)$. Changing t_i by a tiny amount might change the position of $\boldsymbol{c}_i$ significantly and vice versa. We therefore scale the force by the factor $\frac{t_{i-1}^k - t_{i+1}^k}{|\boldsymbol{c}_{i-1}^k - \boldsymbol{c}_{i+1}^k|}$ to capture this difference. We move the parametrization values based on the forces acting

on the corresponding points:

$$t_i^{k+1} = t_i^k + \beta^k \frac{t_{i-1}^k - t_{i+1}^k}{|\boldsymbol{c}_{i-1}^k - \boldsymbol{c}_{i+1}^k|} F_i^k.$$

The constant β^k is allowed to vary for each iteration. We could find β^k the same way as for the line search procedure in the L-BFGS algorithm in Section 2.4, but we have experienced that fixing $\beta \approx 10^{-1}$ for all iterations works well.

CHAPTER 3

Grid Optimization and Clipping

In this chapter, we will look at how to create bounded Voronoi diagrams. To this end, an efficient method for computing the intersection of a Voronoi diagram and a surface is presented. Lastly, we present two different approaches for placing Voronoi sites such that the associated Voronoi cells are as uniform as possible.

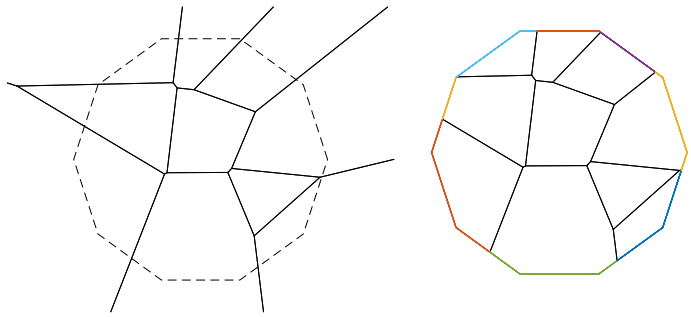
3.1 Restricted Voronoi Diagram

The restriction of the Voronoi diagram to the surface $\partial\Omega$ is called the *restricted Voronoi diagram* (RVD). In the literature it is also called restricted Voronoi *tessellation* [55]. If $\{\boldsymbol{p}_i\}_{i=1,\dots n} = P$ is a set of Voronoi sites, the RVD cell $v_{p_i}^R$ is the subset of the surface $\partial\Omega$ that is at least as close to the site $\boldsymbol{p}_i$ as any other sites,

$$v_{p_i}^R = \{\boldsymbol{x},\ \boldsymbol{x} \in \partial\Omega \subset \mathbb{R}^d, \quad |\boldsymbol{x} - \boldsymbol{p}_i| \leq |\boldsymbol{x} - \boldsymbol{p}_j| \ \forall\ \boldsymbol{p}_j \in P\}.$$

As you can see from Figure 3.1, the RVD cell $v_{p_i}^R$ is intersection of the Voronoi cell v_{p_i} and $\partial\Omega$.

To create the RVD, we need to represent the bounding surface. In the following we will assume the surface is given by a triangulation. Let $\{t_i\}_{i=1}^m$ be the set of triangles that triangulate the surface $\partial\Omega$. The edge-neighborhood of the triangles is also know, that is, for each edge in the



(a) (b)

Figure 3.1: A 2D Voronoi diagram and a corresponding clipped Voronoi diagram. (a) A Voronoi diagram. (b) The Voronoi diagram clipped against a regular dekagon. Each colored boundary edges corresponds to one RVD cell.

triangulation we know which two triangles that have that edge in common. A brute force method for computing the RVD is to compute the intersection of all Voronoi cells and all triangles. This requires the computation of m times n number of intersections, and usually very many of these are empty. We can do much better, and in the following we will present an algorithm that only computes the non-empty intersections. The algorithm is equivalent to the methods presented by Yan et al. [55, 56].

Initialization

First, the Delaunay triangulation of P is created. Remember from Proposition 2.3 that the Voronoi cell k can be defined by the bisectors of p_k and the sites connected to p_k by an edge in the Delaunay triangulation. When we cut a triangle against the bisectors of site $\boldsymbol{p}_k$, we therefore only need to cut against these bisectors. We will assume $P \subset \mathbb{R}^3$. The 2D case can easily be derived from this by thinking of line segments as surfaces in 2D. The algorithm starts by picking a triangle t_i. The Voronoi site $\boldsymbol{p}_k$ that is closest to the centroid of t_i is then found. This will guarantee that t_i and $\boldsymbol{p}_k$ are incident pairs, that is, the intersection of t_i and v_{p_k} is non-empty. We push t_i and $\boldsymbol{p}_k$ into a queue Q.

Main Loop

The triangle site pair in front of Q is popped out. To simplify notation, assume the popped triangle is t_1 and the popped site is $\boldsymbol{p}_1$. Assume also

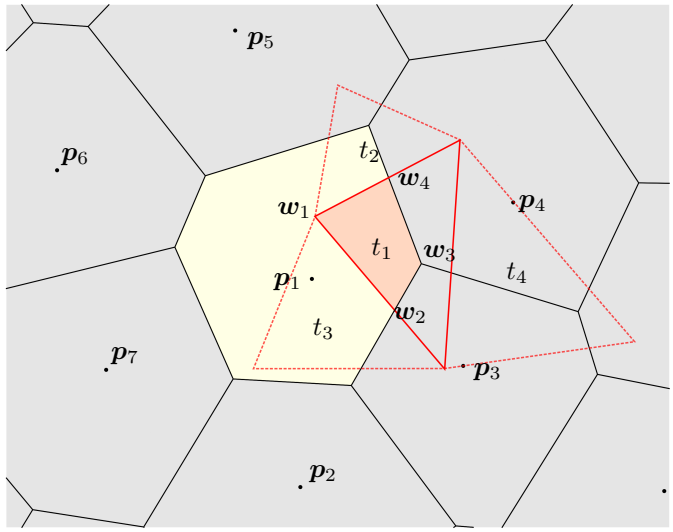


Figure 3.2: Clipping triangle t_1 against the bisectors of site $\boldsymbol{p}_1$. The red shaded area is the intersection of triangle t_1 and cell v_{p_1}. The triangles t_2-t_4 are the neighbor triangles of t_1. The vertex $\boldsymbol{w}_1$ is a type 1 vertex, $\boldsymbol{w}_2$ and $\boldsymbol{w}_4$ are type 2 vertices, and $\boldsymbol{w}_3$ is a type 3 vertex.

the bisectors are ordered such that the k'th bisector is $[\boldsymbol{p}_1, \boldsymbol{p}_k]$. We use the polygon clipping algorithm described in Section 3.3 to clip triangle t_1 against all bisectors of site $\boldsymbol{p}_1$ and its Delaunay neighbors. The normal vector for bisector k is $\frac{\boldsymbol{p}_k - \boldsymbol{p}_1}{|\boldsymbol{p}_k - \boldsymbol{p}_1|}$, and we pick the midpoint $\frac{1}{2}(\boldsymbol{p}_1 + \boldsymbol{p}_k)$ as a point on the bisector. For each vertex $\boldsymbol{w}$ returned from the clipping, we keep track of its bounding planes. There are three different possibilities:

1) $\boldsymbol{w}$ is an original vertex of triangle t_1,

2) $\boldsymbol{w}$ is the intersection of one bisector and a triangle edge,

3) $\boldsymbol{w}$ is the intersection of two bisectors and triangle t_1.

The clipped triangle in Figure 3.2 shows the three different vertex types. The symbolic representation is stored in a vector $symV(\boldsymbol{p}_i) = (k_1, k_2, k_3)$. The symbolic representation of a vertex can always be represented by three integers. A negative index k corresponds to the $(-k)$'th triangle, while a positive index k corresponds to the k'th bisector. When a new vertex is inserted during the clipping procedure, we must find its symbolic represen-

tation. New vertices are only inserted at the intersection of polygon edges and bisectors. Suppose a vertex $\boldsymbol{w}$ is inserted at the intersection of the edge between the vertices $\boldsymbol{v}_i$ and $\boldsymbol{v}_j$ and bisector k. The symbolic representation $symV(\boldsymbol{w})$ is found by taking the intersection of the symbolic representation of $\boldsymbol{v}_i$ and $\boldsymbol{v}_j$ and add the bisector index k:

$$symV(\boldsymbol{w}) = symV(\boldsymbol{v}_i) \cap symV(\boldsymbol{v}_j) \cup k.$$

If $symV(\boldsymbol{v}_i) = (-3, -2, -1)$, $symV(\boldsymbol{v}_j) = (-2, -4, -1)$, and the edge is clipped against bisector 4, the symbolic representation of the new vertex is $symV(\boldsymbol{w}) = (-2, -1, 4)$. The symbolic representation is very useful as it defines the topology of the RVD.

After cell k is clipped, we add new triangle site pairs to the queue based on the symbolic representation. If $symV$ contains a positive index k, the bisector $[\boldsymbol{p}_1, \boldsymbol{p}_k]$ cut the triangle. That means that the triangle t_1 and site $\boldsymbol{p}_k$ are incident pairs, and they are added to the queue. If $symV$ contains a negative index j, this means the triangle t_{-j} and site $\boldsymbol{p}_1$ are incident pairs. In Figure 3.2 the incident pairs $\boldsymbol{p}_1, t_2$ and $\boldsymbol{p}_1, t_3$ are added to the queue because $symV(\boldsymbol{w}_1) = (-1, -2, -3)$. Next, the incident pairs $\boldsymbol{p}_3, t_1$ and $\boldsymbol{p}_4, t_1$ are added to the queue because $symV(\boldsymbol{w}_3) = (-1, 3, 4)$.

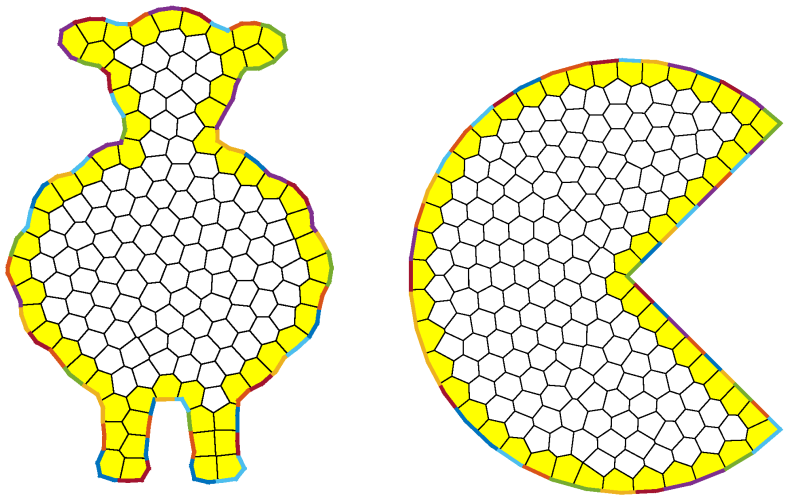
For each cell we have a list of incident triangles. When a triangle site pair is added to Q, the triangle is also added to the cell's incident list. A triangle site pair is not added to the queue if the triangle already is in the cell's incident list.

3.2 Clipped Voronoi Diagram

The Voronoi diagram spans the whole of $\mathbb{R}^d$, and some cells will extend to infinity. In many application we wish to have a bounded diagram. E.g., when the diagram represents a reservoir, it should be bounded by the reservoir boundaries. In computer graphics we might wish to clip the Voronoi diagram against the boundary of the object we try to model. We will call the Voronoi diagram that is restricted to some subdomain Ω of $\mathbb{R}^d$ for the *clipped Voronoi Diagram* (ClVD). Formally we define a clipped Voronoi cell as

$$v_{p_i}^C = \{\boldsymbol{x},\ \boldsymbol{x} \in \Omega \subset \mathbb{R}^d, \quad |\boldsymbol{x} - \boldsymbol{p}_i| \leq |\boldsymbol{x} - \boldsymbol{p}_j| \ \forall\ \boldsymbol{p}_j \in P\}.$$

The RVD and ClVD are connected in the sense that the RVD defines the boundary of the ClVD, that is, the RVD corresponds to all facets of the ClVD that belongs to only one cell. The colored edges in Figures 3.1 and 3.3



(a) (b)

Figure 3.3: A ClVD of a sheep and a Pac-Man shape. The yellow cells are outer ClVD cells, and the white cells are inner ClVD cells. The colored edges corresponds to cells in the RVD.

show a RVD of the corresponding ClVD.

We define two types of ClVD cells. The *inner* ClVD cells are those cells that are not cut by the boundary, and the *outer* ClVD cells are those that are. The inner ClVD cells are equal the corresponding Voronoi cells, and we can compute them by our favorite Voronoi diagram algorithm. The two types of cells are shown in Figure 3.3. If we can locate the outer and inner cells, we only need to treat the outer cells specially by clipping them against the boundary.

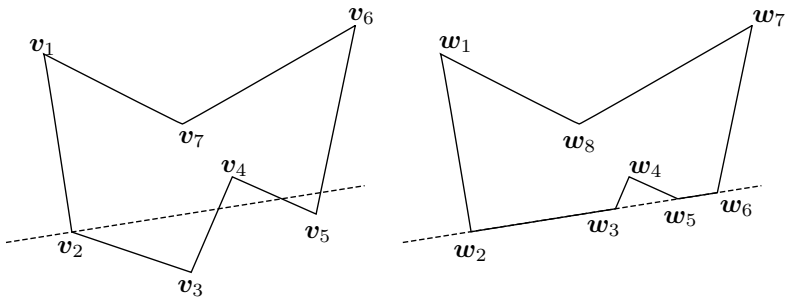
In the following we will present an algorithm that both locates and clips the outer cells simultaneously. The algorithm can be summarized in three steps.

1. Construct the RVD as described in Section 3.1.

2. For all inner cells, construct the Voronoi diagram by any desired method, e.g., the Qhull algorithm.

3. For all outer cells, the RVD defines the outer facets. The remaining vertices are calculated as the dual of the Delaunay triangulation. Any vertices outside the domain Ω are removed.

The main part of this algorithm is the computation of the RVD. Any RVD



(a) (b)

Figure 3.4: (a) Polygon before it is clipped against a plane (dashed line). (b) Polygon after it is clipped.

cells that are non-empty correspond to outer cells. All other sites correspond to inner cells. The inner cells do not cut the boundary and we create them using Qhull. The RVD only gives us the vertices on the boundary. The remaining vertices of the outer cells are computed as the dual of the Delaunay triangulation; a vertex of the Voronoi diagram equals the center of the circumball of a tetrahedron in the Delaunay triangulation. For all outer sites $\boldsymbol{p}_i$ we locate all tetrahedrons that have $\boldsymbol{p}_i$ as a vertex. A vertex is added to the cell $v_{p_i}^C$ at the center of each circumball of the tetrahedrons. We have to be careful since the centers of some circumballs might lay outside our domain. This is not that uncommon, e.g., see Figure 3.1 where one vertex is outside the domain. We check each vertex, and remove any outside the domain.

3.3 Clipping a Polygon

In Section 3.1, we presented an algorithm for creating the restricted Voronoi diagram. The core of this algorithm is to clip polygons against the bisectors in the Delaunay triangulation. To clip a polygon, we use a similar method as the polygon clipping presented by Sutherland and Hodgman [49]. As a reference, see Figure 3.4.

Let $\{\boldsymbol{v}_i\}_{i=1}^n$ be the set of vertices of the polygon. The vertices are ordered counterclockwise such that the line segments $(\boldsymbol{v}_1\boldsymbol{v}_2)$, $(\boldsymbol{v}_2\boldsymbol{v}_3)$, ..., $(\boldsymbol{v}_{n-1}\boldsymbol{v}_n)$, $(\boldsymbol{v}_n\boldsymbol{v}_1)$ equal the edges of the polygon. The polygon is a surface embedded in $\mathbb{R}^3$. We will clip the polygon against a plane, so that the part of the polygon on one side of the plane is kept, while the part on the other side is removed. The plane is described by a normal vector $\boldsymbol{n}$ and a point

$\boldsymbol{x}_0$ on it. We define the distance from a vertex $\boldsymbol{v}_i$ to the plane to be the shortest distance between them

$$d_i = (\boldsymbol{v}_i - \boldsymbol{x}_0)^\top \boldsymbol{n}.$$

We allow for negative distances, and vertices on different sides of the plane will have opposite signs. We assume the normal vector $\boldsymbol{n}$ points outward, so valid vertices will have a negative distance. The clipped polygon has vertices $\{\boldsymbol{w}_1, \dots \boldsymbol{w}_m\} = W$. To find the vertices of the clipped polygon we iterate over all edges of the polygon. At each iteration i, the sign of d_i and d_{i+1} is checked. Depending on the sign of d, there are three cases which will add a vertex to W.

1. If both d_i and d_{i+1} are less than zero, vertex $\boldsymbol{v}_{i+1}$ is added to W.
2. If the signs of d_i and d_{i+1} are opposite, the edge $(\boldsymbol{v}_i\boldsymbol{v}_{i+1})$ crosses the plane. The intersection of $(\boldsymbol{v}_i\boldsymbol{v}_{i+1})$ and the plane is added to W.
3. If d_i is positive and d_{i+1} is negative, the vertex $\boldsymbol{v}_{i+1}$ is added to W.

Notice that case three can only be satisfied if case two is satisfied. What is important is the order in which vertices are added to the cut polygon. If the vertex from case 3 were to be added before the vertex from case two, W would no longer be in a counterclockwise order. The intersection of the line segment and the plane is calculated as in described in Section 2.5. The distance from $\boldsymbol{v}_i$ to the plane $\boldsymbol{r}_i$ needed in Equation (2.5) equals d_i up to a sign. Equivalently for the vertex $\boldsymbol{v}_{i+1}$. The pseudocode is given in Algorithm 1.

3.4 Optimal Voronoi Diagram

The mass center, or *mass centroid*, of a Voronoi cell is given by

$$\boldsymbol{c}_{p_i} = \frac{\int_{V_{p_i}} \boldsymbol{y}\rho(\boldsymbol{y})\,\mathrm{d}\boldsymbol{y}}{\int_{V_{p_i}} \rho(\boldsymbol{y})\,\mathrm{d}\boldsymbol{y}},$$

where $\rho(\boldsymbol{y})$ is a given mass density function. A special class of Voronoi diagrams is one where the sites coincide with the associated mass centroid. If a Voronoi diagram has this property, we call it a *centroidal Voronoi diagram* (CVD). In the literature it is also called for centroidal Voronoi tesselation. The mass center of the boundary cells in a Voronoi diagram will equal infinity. We therefore clip the Voronoi diagram to a subset of $\mathbb{R}^d$.

Algorithm 1 Polygon Clipping

```
v ← polygon vertices
b ← plane
d_i ← distance from v_i to b
for all edges (v_i v_{i+1}) do
  if d_i < 0 and d_{i+1} < 0 then
    W ← v_{i+1}
  end if
  if sign(d_i) ≠ sign(d_{i+1}) then
    W ← (v_i v_{i+1}) ∩ b
    if sign(d_{i+1}) < 0 then
      W ← v_{i+1}
    end if
  end if
end for
return W
```

This is equivalent to the Clipped Voronoi diagram discussed in Section 3.2. Formally we define CVD as follows

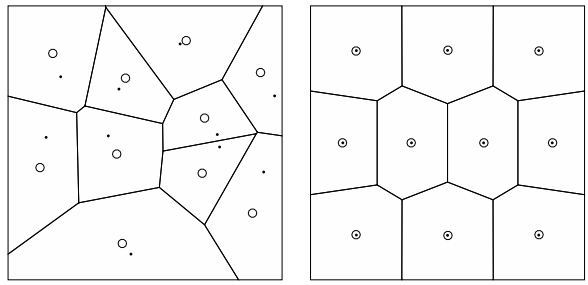
Definition 3.1 (CVD)**.** Let $\Omega \subset \mathbb{R}^d$ be a bounded subset, and let P be a set of sites. The associated Voronoi diagram $\mathcal{V}$ is said to be a *centroidal Voronoi diagram* if

$$\frac{\int_{V_{p_i} \cap \Omega} \boldsymbol{y} \rho(\boldsymbol{y}) \, \mathrm{d}\boldsymbol{y}}{\int_{V_{p_i} \cap \Omega} \rho(\boldsymbol{y}) \, \mathrm{d}\boldsymbol{y}} = \boldsymbol{p}_i \tag{3.1}$$

for all Voronoi cells.

A CVD and a none CVD are shown in Figure 3.5. Notice how the CVD cells are much more regular, and evenly sized. This is one of the reasons CVD is called an optimal Voronoi diagram. A great review about details and applications of CVD is given by Du et al. [14].

How can one generate a CVD? The two sides of Equation (3.1) both depend on the site $\boldsymbol{p}_i$. Further, the set of equations are coupled together by edges in the associated Delaunay triangulation. Finding an analytic expression is extremely hard, if not impossible except for the most simple cases. Therefore, several numerical schemes have been proposed. One of the earliest methods for computing the CVD is using a fixed point iteration scheme. First, we give an initial guess of sites. We construct the Voronoi diagram for these sites, and compute the centroids. These centroids are used as a new guess for the sites, and the process continues until convergence. This scheme is extremely simple and easy to implement, however, linear convergence makes it quite slow.



(a) (b)

Figure 3.5: A Voronoi diagram (a) compared to a centroidal Voronoi diagram (b). Dots are Voronoi sites, while circles are mass centers. Both diagrams have a constant density function and 10 sites.

We define the CVD *energy function* as [14, 26]

$$F(\mathbf{x}) = \sum_{\mathbf{i=1}}^{\mathbf{n}} \int_{\mathbf{V_{P_i}} \cap \mathbf{\Omega}} \rho(\mathbf{y}) |\mathbf{y} - \mathbf{c_i}|^{\mathbf{2}} \mathrm{d}\mathbf{y}. \tag{3.2}$$

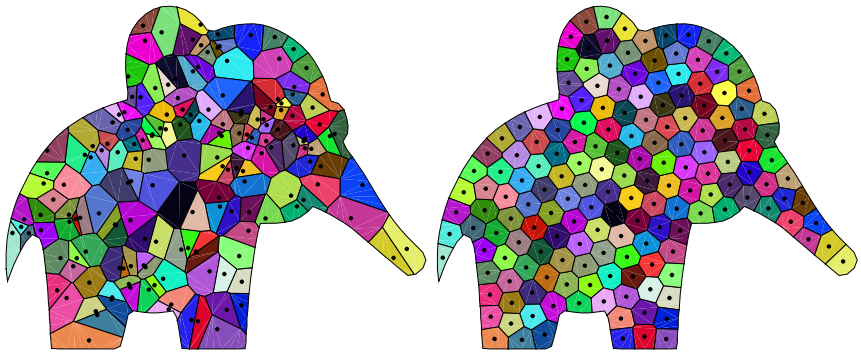
A necessary condition for F to be minimized is $\boldsymbol{p}_i = \boldsymbol{c}_i$, that is, the Voronoi sites coincide with the mass centroids [14]. The gradient of F is [14, 26]

$$\frac{\partial F}{\partial \boldsymbol{p}_i} = 2m_i(\boldsymbol{p}_i - \boldsymbol{c}_i),$$

where m_i is the mass of the associated Voronoi cell. It was long thought that the energy function at most was continuous, due to the change in topology when the sites are moved. However, Liu et al. [33] proved that for density functions $\rho \in \mathcal{C}^2$ the energy function is two times differentiable for convex domains, and almost always two times differentiable for non-convex domains.

The exact Hessian is given explicitly [26, 33], and we can therefore use Newton's method to find the minimizer of the energy function. However, the computation of the Hessian is expensive, and Liu et al. [33] show the advantages of using quasi-Newton methods. Specifically, they show that the L-BFGS algorithm, described in Section 2.4, performs better than both Newton's method and fixed-point iterations.

Figure 3.6 shows two ClVDs of an elephant. The first ClVD is created by placing the set of sites at random. To create the second ClVD we use the randomly generated sites as an initial guess. We find the minimum of the CVD energy function by using the L-BFGS algorithm. At convergence we



(a) (b)

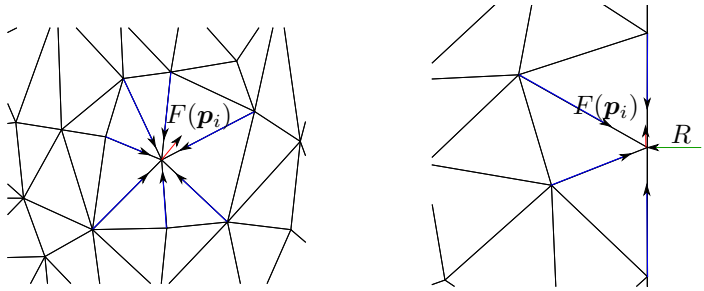
Figure 3.6: A ClVD of an elephant. Black dots are Voronoi sites. Each cell is colored in a unique color, and corresponding cells in the two diagrams have the same color. (a) Initial ClVD. The sites are chosen randomly. (b) The ClVD after the CVD energy function is minimized.

obtain a CVD. Notice that there are only four sites inside the trunk of the elephant in the initial ClVD. The optimization procedure moves sites from the head to the trunk so that all cells in the converged Voronoi diagram have about the same size.

3.5 Optimal Delaunay Triangulation

In Section 3.4, we looked at CVDs and why we call them optimal Voronoi diagrams. Another approach to create a Voronoi diagram with nice properties, is to study the dual Delaunay triangulation. A common measure for the quality of a triangle grid is how uniform the length of the edges are. Persson and Strang [43] present a very successful algorithm for optimizing a Delaunay triangulation based on forces. The Delaunay triangulation is related to a physical truss structure. The edges in the Delaunay triangulation are associated with springs, whereas vertices are associated with the joints connecting the springs. The forces on each joint will depend on the difference between the actual length of the springs and their uncompressed length.

The uncompressed length l_0 of a spring is based on an element size function h. We evaluate the spring at its midpoint. For the domain $[0,1] \times [0,1]$ and element size function $h(x,y) = 1 + x$, the uncompressed length of the springs will be about twice as big in the right side of the domain as the left side.



(a) (b)

Figure 3.7: Forces acting on a joint $\boldsymbol{p}_i$. Blue forces are the repulsive forces from each edge. The red force $\boldsymbol{F}(\boldsymbol{p}_i)$ is the sum of all repulsive forces. The lengths of the force vectors are not proportional to their magnitude. (a) An internal joint. (b) A joint on the boundary. An external force R is acting perpendicular to the boundary. The external force balance the internal forces so the joint will not move across the boundary.

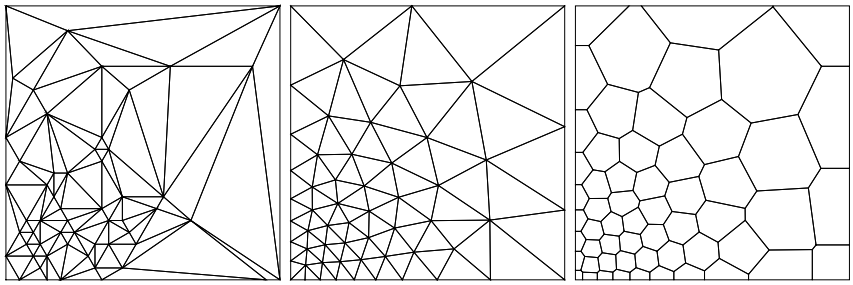
We let the forces from the springs follow Hooke's law; that is, the force is proportional to the difference of its actual length l and its uncompressed length l_0. We assume, however, the springs only have repulsive forces, and no attractive forces. The force f from a spring is:

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

Here, k is a constant of value one that is needed to obtain the correct units.

Let P be the coordinates of all joints. To find the force on a joint $\boldsymbol{p}_i$ we find the force from all springs connected to $\boldsymbol{p}_i$. The total force $\boldsymbol{F}(\boldsymbol{p}_i)$ is the sum of these forces. Figure 3.7a shows seven springs connected to one joint. The repulsive force from a spring acts in the longitudinal direction of the spring. We do not want the joints to move outside the domain we wish to triangulate. Figure 3.7b shows how an external force is added to the boundary joints. The external force is perpendicular to the boundary and balances the repulsive forces of the springs. Boundary joints can therefore only move along the boundary. We also allow for *fixed* joints. The fixed joints can be thought of as glued to their position. They are not allowed to move, no matter how large the forces acting on them are.

The optimization loop of the force based algorithm is very simple. We calculate the Delaunay triangulation of the joints P^k. For each edge in the triangulation, we calculate the repulsive force $f(l, l_0)$. For joints on



(a) Initial Delaunay triangulation. (b) Delaunay triangulation after convergence. (c) Dual Voronoi diagram.

Figure 3.8: Optimization of a triangulation using the force-based algorithm. The element size function is proportional to the distance from the origin squared $h(x, y) \sim x^2 + y^2$.

the boundary we also add an external force to prevent it from passing over the boundary. The total force on a joint is found by summing all repulsive forces and external forces. The total force on a fixed joint is set to zero. All joints are moved a step length ξ along the direction of the total force acting on them:

$$\boldsymbol{p}_i^{k+1} = \boldsymbol{p}_i^k + \xi \boldsymbol{F}(\boldsymbol{p}_i^k).$$

An example of an optimum triangulation and its dual Voronoi diagram is shown in Figure 3.8.

CHAPTER 4

PEBI-Grids Conforming to Wells and Faults

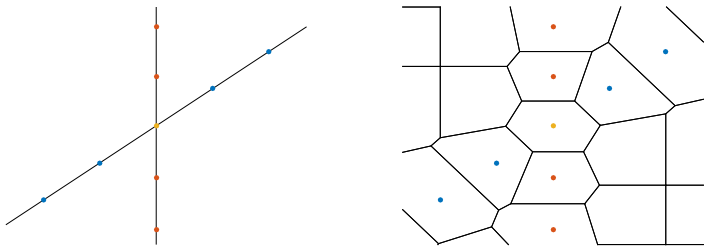
In this Chapter, we will go through the process of generating a PEBI-grid that conforms to wells and faults. We will first present our 2D algorithm, before we discuss how to generalize this to 3D.

We will create three different sets of Voronoi sites: well sites, fault sites, and reservoir sites. Well and fault sites are created to make the grid conform to wells and faults respectively. Reservoir sites are all other sites that create the background grid. We will call any feature that should be traced by centroids of grid cells for wells. Any features that should be traced by faces will be called for faults. Note, these conformity requirements are not exclusive for wells and faults. In some cases, e.g., the simulation case in Section 6.4, a structure can be gridded both as a well and a fault, depending on how we wish to approach the problem.

4.1 2D Algorithms

Generating Well Sites

When generating the well sites, our goal is to trace the wells with cell centroids. In a well-shaped Voronoi grid, the cell-centroids coincide with their respective site [14]. We place a set of well sites along each well as described in Section 2.6. A requirement we put on the well sites is that consecutive well sites should be connected by edges in a Delaunay triangulations of the



(a) (b)

Figure 4.1: Intersection of two wells (black lines). Blue points are well sites for the diagonal well, red points are well sites for the vertical well, and the yellow point is a shared well site. (a) The well paths and well sites. (b) A grid created using a Cartesian background-grid.

sites. From Theorem 2.3, the PEBI-cells of two consecutive well sites will then be neighbors.

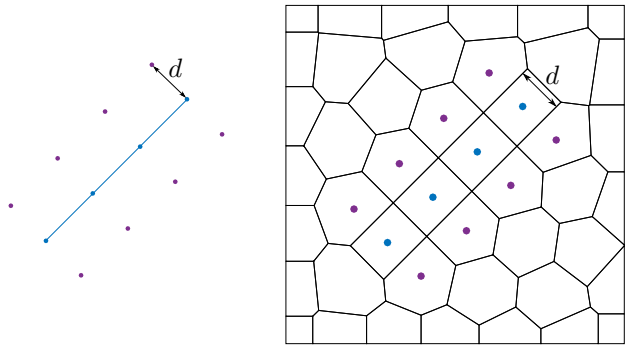
Definition (Well Condition)**.** If $\boldsymbol{p}_1$ is a well site and $\boldsymbol{p}_2$ is a consecutive site, the *well condition* is satisfied if the circle intersecting the two sites centered at their midpoint does not contain any other sites.

By Proposition 2.1, the line segment between $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ will be an edge in the Delaunay triangulation if the well condition is satisfied. In fact, we do not need the circle to be centered at the midpoint, it is enough that *some* circle is empty. To check if there exist an empty circle is tedious, so we have chosen our definition due the its simplicity. Further, if the circle centered at the midpoint is empty, the neighbor edge in the PEBI-grid will contain this point.

When two wells cross, we have to be careful when placing the well sites. If we place the sites of each well independently, consecutive sites will not be connected by Delaunay edges over the intersection. It can also create small and badly shaped cells. To treat these cases, we split all wells at the intersections. A well site is placed at each intersection. A well site at an intersection is shared by all well segments starting or ending in this intersection. Figure 4.1 shows the intersection of two wells. The yellow site is shared by both wells, and the other sites are in this case placed equidistant along the wells. This method ensures a consistent size of the well cells, even at intersections of multiple wells.

Fung et al. [17] suggest to add a protection layer around the well paths to make the shapes of the well cells more regular. Sun and Schechter [48]

show that we can grid the radius of a well explicitly by adding protection sites around the well sites. To add a layer of protection sites, we trace the well paths and place the protection sites normal to the well path. Each well site will have two protection sites, one on each side. Figure 4.2 shows one well with a protection layer. The distance d the protection sites are placed from the well paths also equals the diameter of the corresponding well-cell. For the well sites placed at the intersection of wells, we do not place any protection sites.



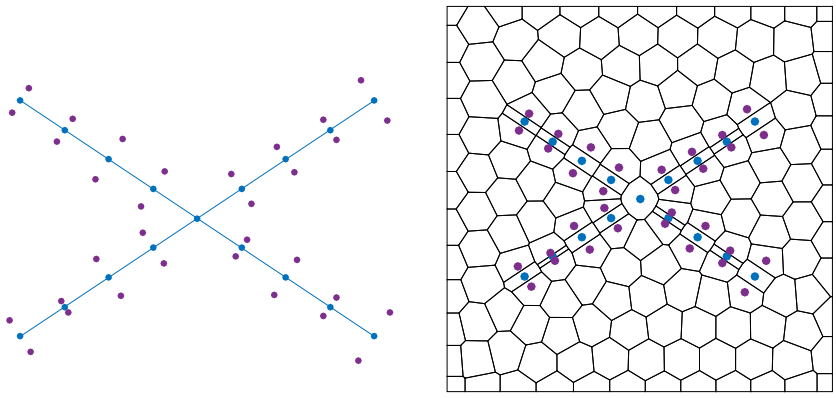
(a) (b)

Figure 4.2: Illustration of a well with a protection layer. Blue points are well sites, while purple points are protection sites. The distance from a well site and its protection sites d also equals the diameter of the corresponding well cell.

We allow for the distance d to vary along the well path. This is practical when we for example wish to create a grid of a fracture with varying width. Figure 4.3 shows two faults intersecting where the distance function is permuted randomly for each set of protection sites. Notice that at the intersection no sites are added. This results in a cell having larger diameter than d.

Generating Fault Sites

To create a grid where edges trace a fault, we have to place fault sites equidistant on each side of the faults. To achieve this, we are inspired by the method described by Ding and Fung [12]. They propose to place fault sites equidistant along a fault by calculating the intersection of circles. They only discuss the specialized case when all circles have the same radius and distance apart. Their method also fails to represent faults exactly at



(a) (b)

Figure 4.3: Intersection of two wells (blue lines). Blue dots are well sites and purple dots protection sites.

intersections. We will generalize the idea of intersecting circles and give a rigorous discussion of how one can guarantee this method to works. We will also present a novel method for handling intersecting faults.

We will generate a set of equidistant sites on both sides of a fault. We start by placing a set of points $C = \{\boldsymbol{c}_i\}$ along the fault as described in Section 2.6. These points will be referred to as circle centers. The distance between two consecutive circle centers $d_i = |\boldsymbol{c}_{i+1} - \boldsymbol{c}_i|$ is set by a density function $\rho(\boldsymbol{c}_i, \boldsymbol{c}_{i+1})$. Draw a circle around each circle center. Two consecutive circles should intersect, which gives us an upper and lower bound on the radii of the circles:

$$d_i \leq R_i + R_{i+1}, \quad |R_i - R_{i+1}| \leq d_i. \tag{4.1}$$

We have chosen to set the radius of a circle as an average of the distance to the circles on either side

$$R_i = c_f \frac{d_i + d_{i-1}}{2}.$$

The constant c_f is called the circle factor, and it controls how far from the fault the fault sites should be placed. Normal values for the constant is in the interval $(0.5, 1)$. If c_f is small, the fault sites will be placed close to the fault. If c_f is large, the fault sites will be placed far from the fault. Figure 4.4 shows the difference of circle factor 0.6 and circle factor 0.9

The fault sites $\{\boldsymbol{f}_j\}$ are placed where two circles intersect. Figure 4.4 shows the fault sites placed around a fault that is described by three circle

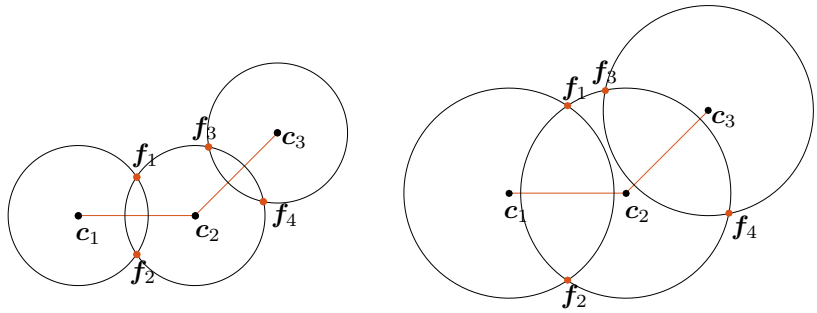


Figure 4.4: The creation of fault sites $\boldsymbol{f}_1$, $\boldsymbol{f}_2$, $\boldsymbol{f}_3$, and $\boldsymbol{f}_4$, from the circle centers $\boldsymbol{c}_1$, $\boldsymbol{c}_2$, and $\boldsymbol{c}_3$. (a) Circle factor is 0.6. (b) Circle factor is 0.9

centers. By construction, the site pairs $\boldsymbol{f}_1\boldsymbol{f}_2$ and $\boldsymbol{f}_3\boldsymbol{f}_4$ are placed equidistant on each side of the fault, which under the following condition will make the PEBI-grid conform to the fault.

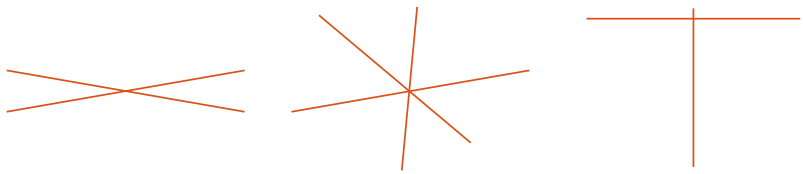
Definition (Fault Condition)**.** Let $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ be two sites from P. If the two sites are placed at the intersection of two circles, the *fault condition* is satisfied if the interior of the two circles contains no sites from P.

The fault condition is a necessary and sufficient condition for the line segment between the two circles to be the neighbor edge between PEBI-cell b_{f_1} and b_{f_2}. All closed circles intersecting the two sites with a center on the line segment are subsets of the two circles centered at $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$. The circles contain no sites, except $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$, thus, the line segment will be an edge in the PEBI-grid.

In a reservoir, it is common to have multiple faults. Creating the fault sites can then be much harder, as the faults may intersect. We present a method that can handle fault intersections, also for the hard cases in Figure 4.5

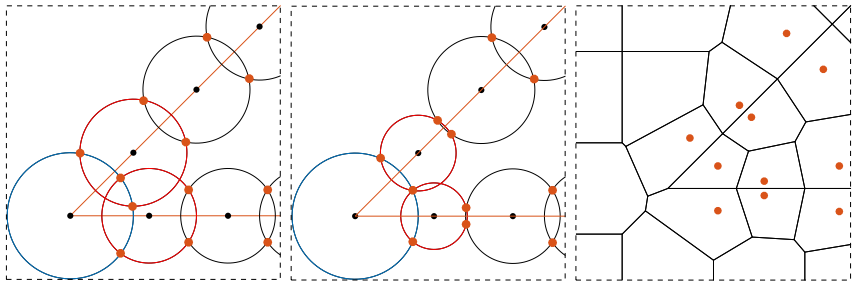
If we place the fault sites for each fault independently, we will in general not be able to represent the faults exactly. At the intersection of two faults, fault sites from either fault may interfere with each other and violate the fault condition. We calculate the intersections of all faults, and the faults are split into fault segments at these intersections. The fault segments will not have any intersections, except possibly at the end-points. This way we do not have to distinguish between different intersection cases, such as, pinchouts or intersection of multiple faults.

At each intersection, we place a circle that is shared by all fault segments



(a) (b) (c)

Figure 4.5: Three hard cases to grid. (a) Faults intersecting at sharp angles. (b) Multiple faults intersecting. (b) Faults that are barely intersecting.

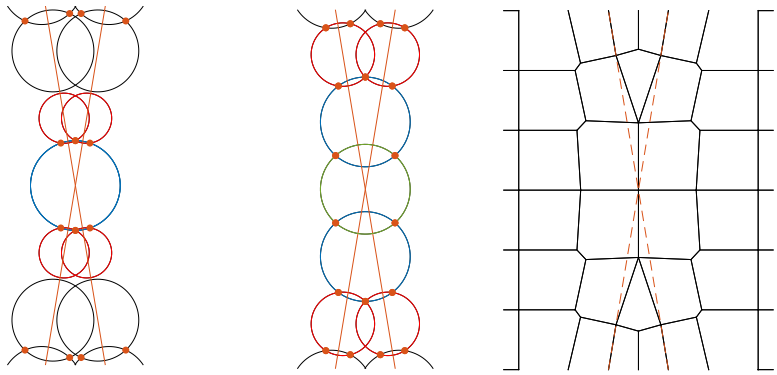


(a) (b) (c)

Figure 4.6: Merging two conflict sites in a pinch out. The orange lines are two intersecting faults and orange points the fault sites. (a) Original sites. (b) Merged sites. (c) A grid created using a Cartesian background-grid.

ending in that intersection. The other circles are placed as normal by using the algorithm in Section 2.6. We color all intersection circles blue, and all neighbor circles of blue circles are colored red. On each red circle one of three actions is performed; (i) nothing is modified, (ii) the radius is changed, (iii) the circle is merged with another red circle. If the interior of a circle does not contain any sites it is not modified. If the interior of a circle contains the fault site $\boldsymbol{f}_i$, we locate the red circle that generated $\boldsymbol{f}_i$. These two circles are tagged as conflict circles. The radii of the conflict circles are shrunk as shown in Figure 4.6. The new radii are chosen such that the blue circle and the two red circles intersect at the midpoint of the two faults. When multiple faults intersect, a circle might have multiple conflict pairs. We then calculate the circle's new radius for each conflict pair and choose the smallest of them.

If the radius of a red circle is shrunk too much, it might violate the radius condition of Equation (4.1). For those cases, we locate the other red conflict circle sharing a fault site with this circle. These two circles

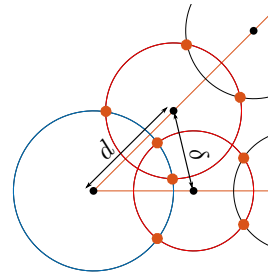


(a) (b) (c)

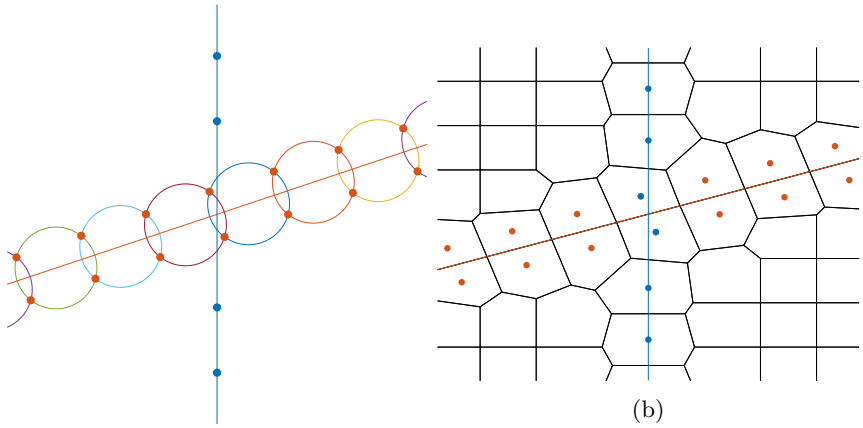
Figure 4.7: The procedure of merging circles. (a) The circle at the intersection is colored blue, and its neighbors are colored red. The radii of the red circles are shrunk until they intersect the blue circle at the same point. The red circles now have an imaginary intersection with its neighbors. They are therefore merged. (b) The merged circles are colored blue, and their neighbors are colored red. The green circle is already processed and is therefore not colored red. The procedure from (a) is repeated until the fault condition is satisfied. (c) An associated grid.

are merged to one circle centered at the midpoint of them. The merged circle is colored blue, and we repeat the procedure above. Figure 4.7 shows one iteration of the merging. In this case two sets of conflict circles are merged, one on each side of the intersection. This is is enough to satisfy the fault condition. If the intersection had been sharper, we might have had to merge more circles recursively.

A second case that trigger two red conflict circles to merge is if they are too close to each other. From the figure on the right; if the ratio $\frac{\delta}{d}$ is smaller than a given tolerance, we merge the circles. By changing the tolerance parameter we can control how we grid sharp intersections. For a large tolerance more circles will be merged than for a small tolerance.



Our method of splitting faults into fault segments and placing circle centers along these segments, makes it easy to handle barely intersecting faults. If a fault segment is shorter than a specified length, we do not place any circles along it. In our implementation we have set this minimum length to be 80% of the desired length between circle centers.



(a) (b)

Figure 4.8: Intersection of a fault (red) and a well (blue). The well sites at the intersection is removed and the fault sites are placed as close as possible to the well. (a) The well and fault sites. (b) A grid created using a Cartesian background grid. The fault sites at the intersection are labeled as well sites.

Special consideration is also taken when a fault intersects a well. All faults and wells are split at the intersections. Figure 4.8 shows the intersection of a well and a fault. The first circle center of a fault segment starting in a well-fault intersection is placed half a step length from the start. Equivalently, the last circle center of a fault segment ending in a well-fault intersection is placed half a step length from the end. The two fault sites created from the circle before and after the well-fault intersection are labeled as well sites. These two sites are the first and last well site for the well segments starting and ending in the intersection, respectively.

Generating Reservoir Sites

The reservoir sites can be placed any way the user may see fit. The most obvious choice is to create a Cartesian grid by placing the sites equidistant in the x and y direction. When placing the reservoir sites, we usually ignore all faults and wells. After the reservoir sites are created, we remove any sites violating the fault or well condition. The resulting grid is then guaranteed to conform to faults and wells. Even if the fault and well conditions are satisfied, we might want to remove a few more reservoir sites. For each well and fault site we define a grid size. For well sites the grid size is the distance between two consecutive well sites. For fault sites the grid size is

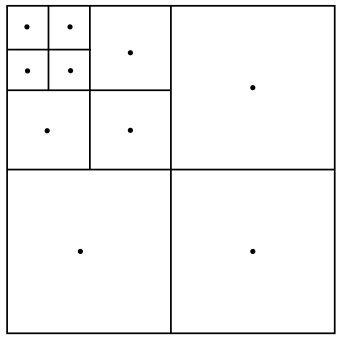


Figure 4.9: A cell refined by three levels.

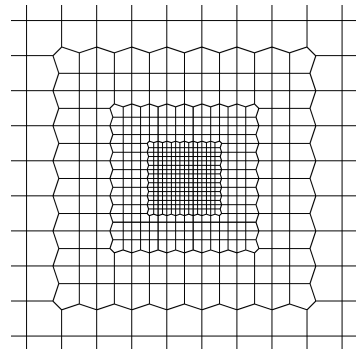


Figure 4.10: A grid generated with three levels of local grid refinement.

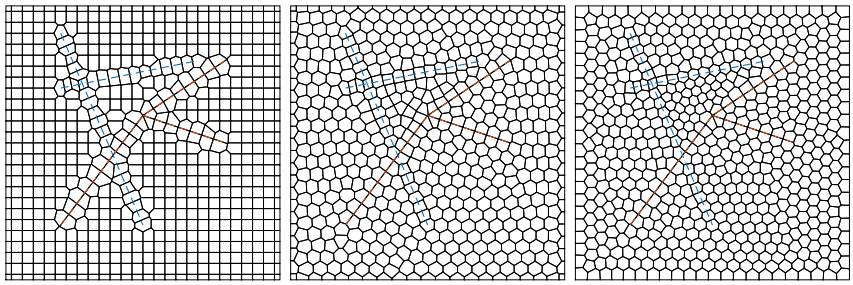
set to the distance between the two sites that are generated by the same two circles. If a reservoir site is closer to a well or fault site than that site's grid size, the reservoir site is removed.

As well as making a grid conforming to faults and wells, one often wants to refine the grid in areas close to the wells. A well-known method for refining a Cartesian grid is Multilevel Quad-Tree Local Grid Refinement [17]. A cell is refined by dividing it in four by connecting the midpoints of the opposing edges, as shown in Figure 4.9. This can be repeated any number of times until a wanted refinement is reached. We can refine the reservoir sites in a similar manner. If a cell is to be refined, replace the original reservoir site by four new sites centered at each quadrant. This will result in non-square cells at the boundary of each refinement level, shown in Figure 4.10.

To create a fully unstructured grid, we can place the reservoir sites using the force-based method in Section 3.5. To achieve refinement towards wells, we create an element size function that decreases towards wells. We let the element size function decrease exponentially to achieve a similar refinement as for the quad-tree refinement:

$$h_r(\boldsymbol{p}) = \min\left[h_{\max}, h_{\min}\exp\left(\frac{d(\boldsymbol{p}, W)}{\varepsilon}\right)\right]. \tag{4.2}$$

The desired grid size of the background grid far from and close to the wells is $h_{\max}$ and $h_{\min}$ respectively. The distance $d(\boldsymbol{p}, W)$ is the closest distance from the point $\boldsymbol{p}$ to the set of well sites W. The constant ε controls the transition region. If ε is small, the refinement happens quickly around the



(a) Cartesian reservoir sites.

(b) Optimized Delaunay triangulation.

(c) Minimized CVD energy function.

Figure 4.11: Three grids of a reservoir. The reservoir has two wells (dotted blue lines) and two faults (dotted orange lines). The well and fault sites are the same for all three grids and are created using the methods described in this chapter. The reservoir sites are created by three different methods; a Cartesian grid, optimizing the dual Delaunay triangulation, and minimizing the CVD energy function.

wells. If ε is large, the transition region is large. When we run the force algorithm, all well and fault sites are set as fixed points.

The last method we will discuss for generating reservoir sites is a variant of the CVD algorithm presented in Section 3.4. This method is similar to the force-based method, but we optimize the Voronoi diagram directly instead of optimizing the dual Delaunay triangulation. We define the fault and well sites as fixed sites, that is, their gradient is zero. The CVD energy function is then minimized, but without moving any of the well or fault sites. A comparison of the three methods for placing reservoir sites is shown in Figure 4.11. The two optimization methods have more uniform cells than the Cartesian background grid. Also, the cell centroids for the optimization methods are very close to the well path. The minimization of the CVD energy function creates better reservoir cells in congested areas, e.g., look at the area between the well intersection and fault intersection.

All the steps for creating a conforming PEBI-grid are summarized in Algorithm 2, and the steps are shown in Figure 4.12 for a Cartesian background grid.

4.2 2.5D Grids

It is not uncommon for an oil reservoir to have very large aspect ratios. An oil reservoir can stretch kilometers in the lateral directions, but only hundred meters in vertical direction. 2.5D grids are 3D grids that take

Algorithm 2 Unstructured Griding.

(1) Create fault and well sites

(1.i) A set of well sites is placed along each well path according to a well cell density function.

(1.ii) A set of circle centers is placed along the faults according to a fault cell density function. Around each circle center, a circle is drawn. The fault sites are placed at the circle intersections.

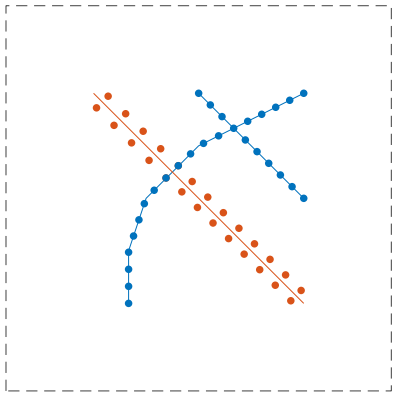
(1.iii) Special care has to be taken when wells and faults intersect. If two or more wells intersect at a point, a well site is placed at the intersection. The rest of the well sites are placed as normal. For fault-fault intersections, a circle is placed at the intersections. The other circles are placed as normal and the fault sites set at the circle intersections. If a well and fault intersect, a fault circle is placed half a step length on both sides of the intersection. The two fault sites generated by these circles are labeled as wells. The other well sites are placed as normal.

(2) A set of reservoir sites are created in the domain.

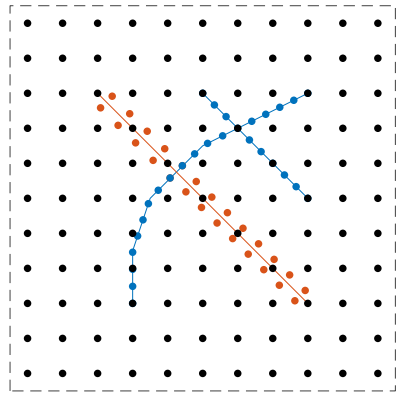
(3) Other types of sites are inserted, e.g., refinements around wells.

(4) All reservoir sites that violate the fault or well condition are removed.

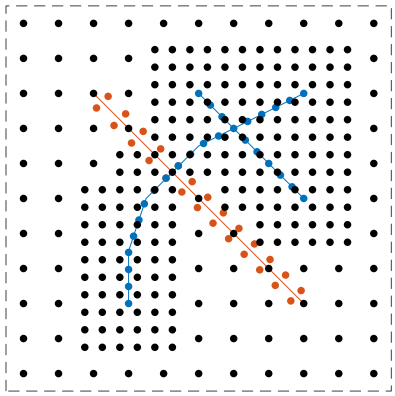
(5) Well and fault sites are assigned a minimum grid size. For well sites the minimum grid size equals the distance between two consecutive well sites. For fault sites the minimum grid size equal the distance between the two fault sites created by the same two circles. If a reservoir site is closer to a fault or well site than the allowed minimum grid size, the reservoir site is removed.



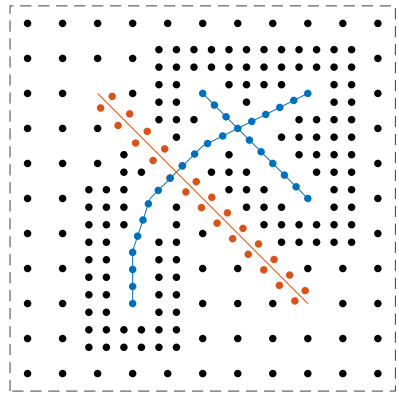
(a) Create well and fault sites.



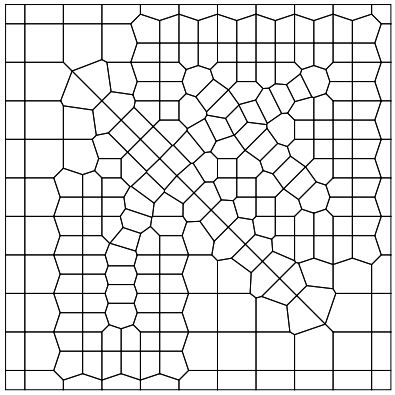
(b) Create reservoir sites.



(c) Refine the reservoir sites.



(d) Remove conflict sites.



(e) Assosiated PEBI-grid.

Figure 4.12: Generating a grid using Algorithm 2. Orange line is a fault and blue lines are two wells. (a) Step (1) creates the well and fault sites (blue and orange dots). (b) Step (2) creates the reservoir sites. (c) Step (3) refines the reservoir sites close to wells. (d) Steps (4) and (5) remove conflicting grid sites. (e) PEBI-grid created from the grid sites.

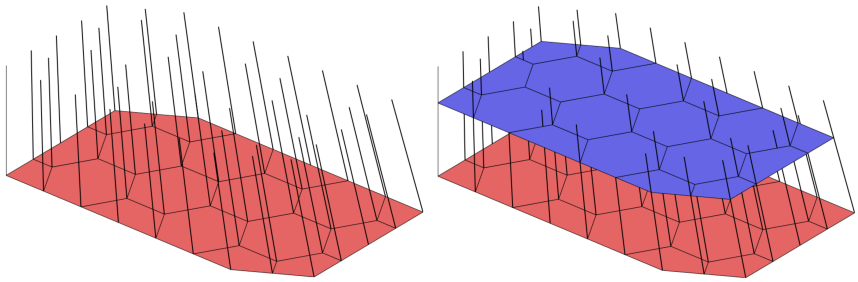


Figure 4.13: The creation of a 2.5D grid. First, a 2D layer is gridded (red layer) and a set of pillars is placed through all vertices. Then, the grid is extruded along the pillars. Figures from Lie [32].

advantage of the flexibility of 2D griding to create complex grids in lateral direction, and the simplicity of a Cartesian grid in the vertical direction. The rock in oil reservoirs is formed by a sedimentation process that creates natural horizontal layers. These layers are often captured very well by 2.5D grids, which have made these types of grids are very popular in reservoir simulation.

A 2.5D grid is created by generating a 2D grid of a layer in lateral direction, or aligning with some major horizon. A set of pillars is created, one pillar going through each vertex. The grid is extruded along these pillars, as shown in Figure 4.13. The pillars can be vertical or inclined such that they align with faults. The hard part of creating a 2.5D grid is the choice of pillars and the length each cell is extruded along them. An example of a 2.5D grid that aligns with an inclined fault is shown in Figure 4.14.

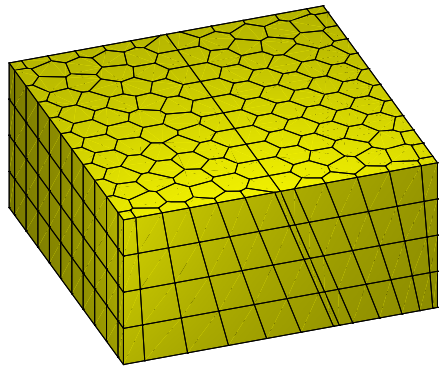


Figure 4.14: A 2.5D grid aligned with an inclined fault.

4.3 3D Algorithms

Creating PEBI-grids conforming to faults is much harder in 3D than in 2D. Instead of dealing with lines, one has to consider surfaces, and the complexity increases drastically. In this section, we will present a generalization of the 2D method presented in Section 4.1. This is a novel method, which to the best of our knowledge is new in the literature.

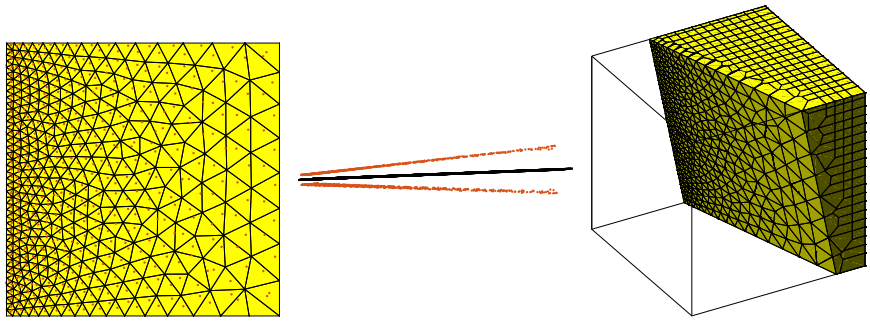
Generate Fault Sites

In 3D, faults are represented by surfaces. This added degree of freedom (compared to 2D) increase the complexity drastically. In 2D, it was sufficient to use two circles to create equidistant fault sites on each side of the fault. One might to try the same in 3D by taking the intersection of two spheres, but the intersection will give a circle instead of two points. To define two unique points, one possibility is to choose the two points on the circle that are furthest from the fault surface. Another method is to take the intersection of a third sphere. The intersection of three spheres yields two unique points equidistant from the plane defined by the sphere centers. This method is the natural extension of intersecting circles in 2D, which is why we have opted to use it to generate the fault sites.

To generate the fault sites, we will represent a fault by a surface triangulation. A triangle is the equivalent to a line segment in the 2D gridding method. Around all vertices in the triangulation, we draw a sphere. We let the radius of the spheres vary based on a fault cell density function. The radius of a circle should be approximately the same as the length of the edges in the corresponding triangle. For each triangle, we find the intersection of the three spheres centered at the triangle's vertices. This intersection gives two points equidistant on each side of the triangle, and we place one fault site at each of these points. The resulting PEBI-grid will have faces tracing the fault. In fact, the faces on the fault will equal the triangulation.

The reservoir sites can be created by any preferred method. The methods discussed in Section 4.1 easily generalize to higher dimensions. As long as the fault condition is satisfied for all spheres, the associated PEBI-grid will conform exactly to the faults. Figure 4.15 shows a fault in the unit cube. We create a triangulation of the fault using the force-based algorithm in Section 3.5. The relative element size function is set to $h = 1 + 4x$, and the radii increase at the same rate $R = \frac{1}{40}(1 + 4x)$.

To handle intersections of faults, we introduce a priority scheme. Each



(a) (b) (c)

Figure 4.15: The generation of fault sites in 3D. Orange points are generated fault sites. The domain is the unit cube. (a) A side view of the fault triangulation. The size of the triangle edges increases linearly towards the right end of the fault. The radii of the circles are growing with the same rate. (b) Top view of the fault and fault sites. (b) A grid of the domain.

fault is given a priority by the user. If a fault site is in the interior of a circle of a fault with higher priority, it is removed. We start by finding any sites that violate the fault condition for the fault with highest priority. These sites are removed. We then consider the fault with second highest priority and remove any sites inside the generating circles of this fault. This process continues recursively until we have checked all faults. When we check which sites that violate the fault condition for a fault, we only check sites that have a lower priority than the current fault. If two faults have the same priority, they will not affect each other. A grid of two intersecting faults is shown in Figure 4.16. The fault with higher priority is represented exactly, while the fault with lower priority is only approximated over the intersection.

A last example is shown in Figure 4.17. We have created an inclined fault in the unit cube. The fault also curves in the horizontal direction.

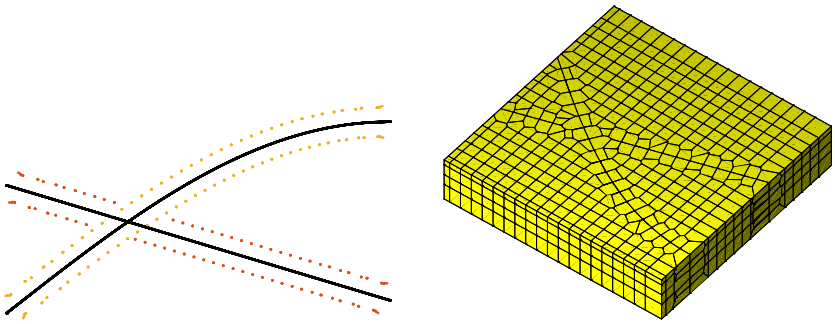


Figure 4.16: Two intersection faults in 3D. (a) Top view of the fault and fault sites. Orange points show sites of the fault with higher priority, while the red points show sites of the fault with lower priority. (b) A grid of the domain.

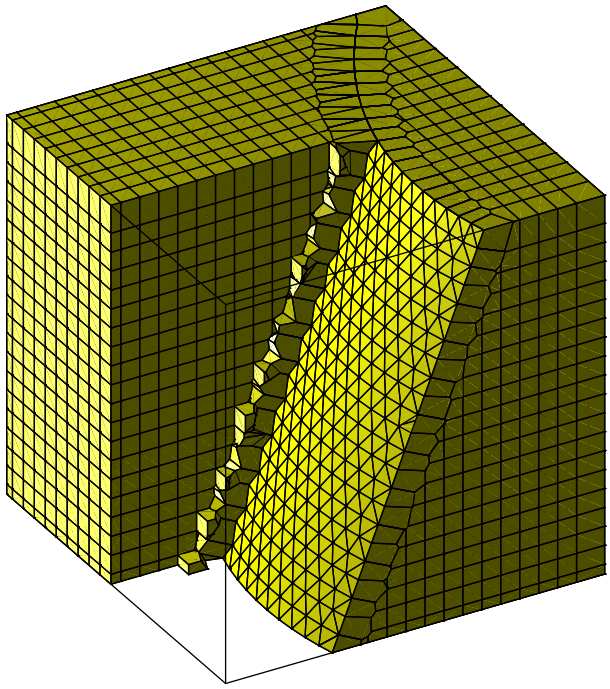


Figure 4.17: An inclined fault that curves in the horizontal direction.

CHAPTER 5

Implementation in Matlab

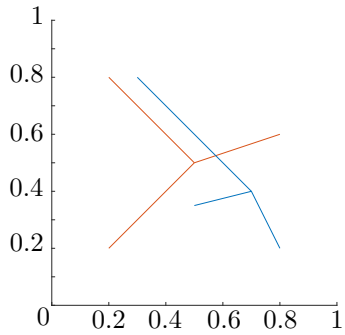
All algorithms from Chapters 3 and 4 have been implemented in Matlab. The implementation is compatible with the Matlab Reservoir Simulation toolbox (MRST) [32]. The focus of the implementation has been to create a basis for generating unstructured PEBI-grids for this toolbox. A set of functions and a number of illustrative examples have been created and can be downloaded from a git repository [5] under the terms of the GNU v3.0. In this chapter, we will discuss the most important features of the implementation and how one can use it efficiently.

5.1 2D Faults and Wells

The griding module contains several routines for creating grids. The most automated routines are `compositePebiGrid` and `pebiGrid`. The two routines create a valid MRST grid in a square domain. These two routines are used to create most 2D grids in this thesis. The two routines handle faults and wells, as well as intersections. The routines are equivalent up to how they generate the reservoir sites. The first routine generates the reservoir sites as a Cartesian background grid, whereas the second routine uses the Delaunay force algorithm from Section 3.5.

As an illustration, we will create a minimalistic test case. First, the wells and faults are defined:

```
f = {[0.2,0.2;0.5,0.5;0.8,0.6], ...
     [0.2,0.8;0.5,0.5]};
w = {[0.8,0.2;0.7,0.4;0.3,0.8], ...
     [0.7,0.4;0.5,0.35]};
```



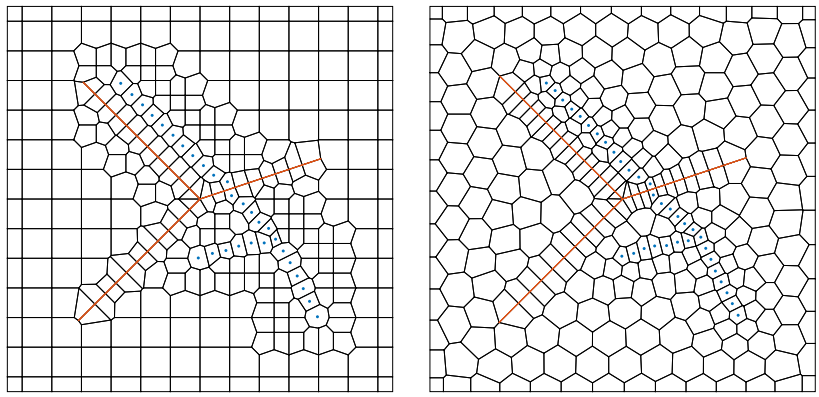
All routines assume that faults and wells are piecewise linear paths. The faults and wells are stored in a cell array, where each element is a set of coordinates describing one path.

The two routines have two required parameters, the grid size and the size of the domain. The routines assume that the domain is a square with lower left corner at the origin and upper right corner given by the domain size. In this example, we will refine the grid close to wells. We set the size of the fault and well cells to half the size of the reservoir cells. The relative size is set by the optional parameters `'faultGridFactor'` and `'wellGridFactor'`. Further, we define one level of local grid refinement for `compositePebiGrid`. For the fully unstructured grid created by `pebiGrid`, we set the refinement constant ε (from Equation (4.2)) to $\frac{1}{5}$.

```
gS = [.08,.08];
G = compositePebiGrid(gS ,[1,1],'faultLines',          f,  ...
                                'wellLines',           w,  ...
                                'faultGridFactor',   0.5,  ...
                                'wellGridFactor',    0.5,  ...
                                'mlqtMaxLevel',        1);

Gp = pebiGrid(gS(1), [1,1],     'faultLines',          f,  ...
                                'wellLines',           w,  ...
                                'faultGridFactor',   0.5,  ...
                                'wellGridFactor',0.8*0.5,  ...
                                'wellRefinement',   true,  ...
                                'wellEps',          1/5);
```

The two routines tag the well cells and fault faces. The logical array `G.cells.tag` has a length equal the number of cells. The element `G.cells.tag(i)` is true if cell `i` is a well cell. Equivalently, the array `G.faces.tag` is a logical array of length equal the number of faces. The element `G.faces.tag(i)` is true if face `i` is a fault face. The two grids and the tagged cells and faces are shown in Figure 5.1.



(a) `compositePebiGrid`. (b) `pebiGrid`.

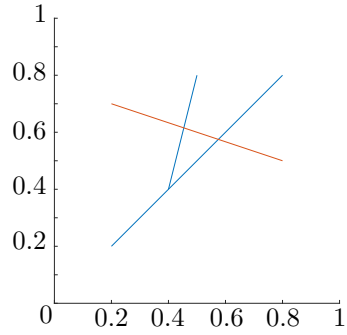
Figure 5.1: Two grids generated by the routines `compositePebiGrid` and `pebiGrid`. Blue dots are centroids of cells tagged as well cells. Orange edges are edges tagged as fault edges.

5.2 Working with Lower-Level Routines

In some cases, the routines used in Section 5.1 might not be enough to create the grid we wish. We might not want a square domain, or maybe we would like to create the reservoir sites with a different algorithm. To this end, we discuss how we can use the lower-level routines in our module. Creating a grid from these routines requires a bit more work, but we are even more flexible to achieve the results we wish.

We create a fault intersected by two wells:

```
well  = {[0.2,0.2; 0.8,0.8], ...
         [0.4,0.4; 0.5,0.8]};
fault = {[0.2,0.7; 0.8,0.5]};
```



This is a quite simple case, with one fault that is intersected by a branching well.

The next step is to find the intersection of all wells and faults. At each intersection we split the paths. The new set of paths has no intersections,

except possibly at the endpoints. To split the paths, we use the routine `splitAtInt`:

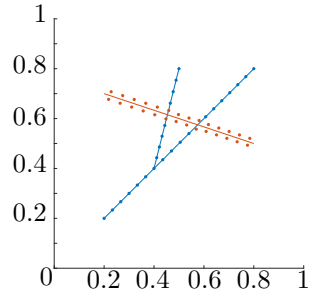
```
[sWell, wCut, wfCut] = splitAtInt(well, fault);
[sFault,fCut, fwCut] = splitAtInt(fault, well);
```

The routine `splitAtInt` takes in two cell arrays of paths and calculates all intersections between the paths in the first argument. Then, it calculates the intersection of all paths in the first argument and the second argument. Finally, it returns the paths of the first argument, which are now split at all intersections. The routine also returns two additional vectors. The first vector gives information about the intersections between the paths in the first cell array. The second vector gives information about the intersection of the paths of the first cell array and the paths of the second cell array. If `fCut(i)=1`, it means that `sFault(i)` has a fault intersection at the end. If `fCut(i)=2`, it means that `sFault(i)` has a fault intersection at the start. If `fCut(i)=3`, it means tha `sFault(i)` has a fault intersection at both the start and end. Equivalently for `fwCut`, but this gives the fault-well intersections. In this example we have the following:

```
sFault =                                fCut =      fwCut =
   {[0.2,0.7; 0.4538, 0.6154],             0           1
   [0.4538, 0.6154; 0.575, 0.575],         0           3
   [0.575, 0.575; 0.8,0.5]}                0           2
```

We will use the routine `createFaultGridPoints` to generate the fault sites, and the routine `createWellGridPoints` to generate the well sites. These routines are able to handle intersections of faults and wells and have several options available to tweak the generation of the sites. We will use the preset options, but we have to specify the well-fault intersections for each path. If this is not done, the routines just ignore the intersections.

```
df = 0.05; dw = 0.05; dr = 0.05;
F = createFaultGridPoints(sFault, df, ...
                          'fwCut', fwCut);
fSites = F.f.pts;
fGs    = F.f.Gs;
[wSites, wGs] = ...
   createWellGridPoints(sWell, dw, ...
                        'wfCut', wfCut);
```

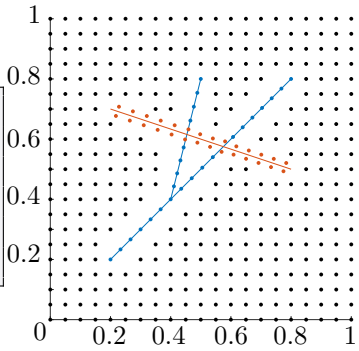


The routine `createFaultGridPoints` returns a struct `F` that contains all

the information about the fault sites. Notably, it contains all generating circles and a mapping to and from the fault sites.

We generate a Cartesian background grid by placing the reservoir sites equidistant in the x and y direction:

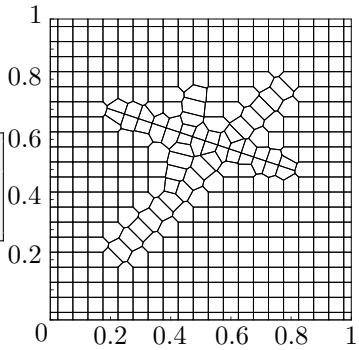
```
[X,Y]  = meshgrid(0:dr:1);
rSites = [X(:),Y(:)];
rSites =                                ...
   removeConflictPoints2(rSites,        ...
                    [fSites;wSites],    ...
                     fGs;wGs]);
```



We use the routine `removeConflictPoints2` to remove any reservoir sites that are too close to well or fault sites.

We have now created all sites for our grid. By using the MRST routine `triangleGrid` we create a Delaunay triangulation of the sites. We then create the PEBI-grid as the dual of the Delaunay triangulation by using the MRST routine `pebi`:

```
sites = [fSites;wSites;rSites];
Gt    = triangleGrid(sites);
G     = pebi(Gt);
```

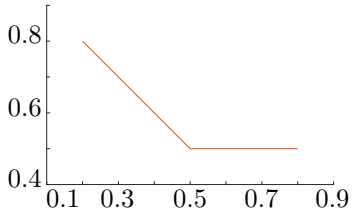


5.3 Generate 2D Fault Sites

In this section we will go through the most crucial part of the implementation that places sites equidistant on both sides of a fault. The following is a very simplified version of the routine `createFaultGridPoints`. We will show a simple example of a single fault with no intersection. It is much simpler to create a grid conforming to wells than faults. Placing well sites is basically equivalent to placing circle centers along the faults, as we will show later. We therefore do not explain it further.

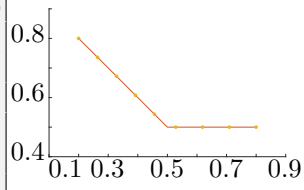
We start by creating a fault path of two line segments. We also set the circle factor and the desired distance between fault circles:

```
fault    = [0.2,0.8; 0.5,0.5; 0.8,0.5];
dFault   = 0.1;
circFact = 0.6;
```



Notice that the second row of the variable `fault` corresponds to the end point of the first line segment *and* the start point of the second line segment. The first step of the algorithm is to place a set of circles equidistant along the fault. We use the built-in Matlab function `interp1` to interpolate the fault:

```
linesDist = sqrt(sum(diff(fault,[],1).^2,2))
linesDist = [0; linesDist];
cumDist   = cumsum(linesDist);
dt        = cumDist(end)/ ...
              ceil(cumDist(end)/dFault);
newPtsT   = 0:dt:cumDist(end);
CC        = interp1(cumDist,fault,newPtsT);
```



This will create a set of equidistant circle centers if you measure the distance along the fault path. For two consecutive circle centers that lie on different line segments, the Euclidean distance may therefore be slightly shorter. This difference is usually not big, but we have to be careful and take it into account when calculating the circle intersections.

Now that we have placed the circle centers, we need to set the radii of the circles. The radius of circle i has to be bigger than one-half of the distance to the circles on either side. We have chosen to set the radius to be the circle factor times the average distance to the circle centers on either side. This works for almost all cases, but if one for some reason places the circle centers very irregularly, one may need to set a different radius to make sure the circles intersect.

```
d  = sqrt(sum((CC(2:end,:) - CC(1:end-1,:)).^2, 2));
CR = circFact*[d(1); (d(1:end-1) + d(2:end))/2; d(end)];
```

To calculate the intersecting points we use the method described in Section 2.5. The basis vectors of the new coordinate system are calculated,

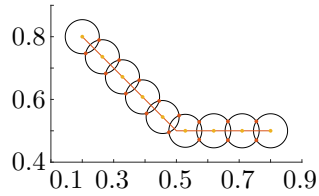
```
n1 = (CC(2:end,:)-CC(1:end-1,:))./repmat(d,1,2); %Unit vector
n2 = [-n1(:, 2), n1(:,1)];                       %Unit normal
```

together with the coordinates:

```
a = (d.^2 - CR(2:end).^2 + CR(1:end-1).^2)./(2*d);
h = sqrt(CR(1:end-1).^2 - a.^2);
```

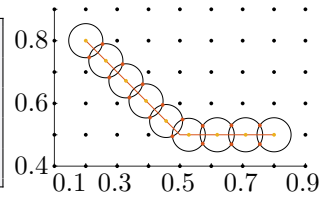
Transforming back to the original coordinate system, we obtain the intersections:

```
lPts = CC(1:end-1,:)+bsxfun(@times,a,n1)...
                    +bsxfun(@times,h,n2);
rPts = CC(1:end-1,:)+bsxfun(@times,a,n1)...
                    -bsxfun(@times,h,n2);
pts  = [lPts; rPts];
```



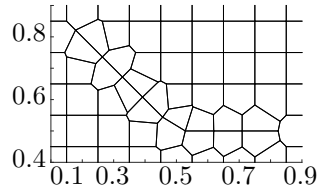
We now create a Cartesian background grid, pretending the fault does not exists. The Cartesian grid is created by placing reservoir sites equidistant in the unit square. After the reservoir sites are generated, we remove any sites inside one of the circles created above.

```
[X,Y]   = meshgrid(0:dFault:1);
backPts = [X(:),Y(:)];
CRrep   = repmat(CR',size(backPts,1),1);
removed = any(pdist2(backPts,CC)<CRrep,2);
backPts = backPts(~removed,:);
```



Now that all sites are created we can generate the MRST-grid:

```
Gt = triangleGrid([pts;backPts]);
G  = pebi(Gt);
```



5.4 Creating a 3D PEBI-Grid in MRST

Unlike in 2D, MRST does not have any routines for creating a 3D PEBI-grid. Instead of reinventing the wheel, we wish to take advantage of the build-in routines in Matlab for generating Voronoi diagrams. Matlab uses the Qhull algorithm, which is fast, robust, and well tested. Our routine `voronoi2mrst` transform the output from the Qhull algorithm to a valid MRST grid. In this section, we will go through the most important details of this routine.

The grid structure in MRST is very general. This has the advantage that it is able to represent almost any kind of grids. On the other hand, it can not take advantage of the structure that some grids have, which makes storing the grid more expensive. The Qhull algorithm for creating Voronoi diagrams uses a very simple grid structure. It stores two sets: (i) a set of vertices coordinates, and (ii) a mapping from cells to the set of vertices. This representation is very compact, but does not contain much information. To generate a valid MRST, grid we need to create several mappings, e.g., which faces belong to which cells. First, we present all fields required for a valid MRST grid. Then, we present how one can convert a Qhull grid structure into a MRST grid structure. For a full explanation of the MRST grid structure, we refer the reader to Lie [32].

A grid in MRST is stored as a struct which we will call `G`. The grid has three substructs, `G.cells`, `G.faces`, and `G.nodes`. The first substruct `G.cells` contains information about the cells and mappings from cells to `G.faces`. The second substruct contains information about the facets. In 3D, a facet is a surface, whereas in 2D it is a line segment. The last substruct contains the information about the vertices. The substruct `G.cells` must contain at least the following fields:

- `num`: A scalar of the total number of cells n_c in the grid.
- `facePos`: A $n_c+1\times 1$ array that maps from cells to half-faces. The index of all half-faces belonging to cell i is: `G.cells.facePos(i):G.cells.facePos(i+1)-1`. A face can be imagined to consist of two half faces; each half face belonging to one of the two cells sharing the face. The number of half faces equal $n_h =$ `facePos(end)-1`.
- `faces`: A $n_h \times 1$ array that maps from half-faces to faces. The index of the face that belongs to half-face i is found at `faces(i)`.

Figure 5.2 shows a grid with the corresponding mappings.

```
cells.facePos =    cells.faces =
   1                  5
   4                  3
   7                  1
   10                 8
   13                 2
   16                 3
   19                 7
                      10
                      5
                      :
```

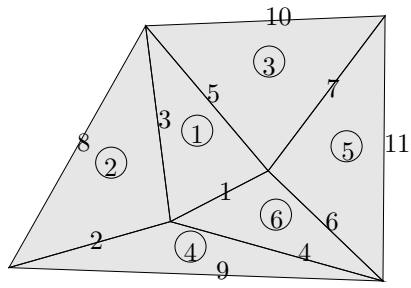


Figure 5.2: A grid and the corresponding mappings `G.cells`. Numbers with circle around them are cell indices, while the other numbers are face indices.

The substruct `G.faces` contains the following mandatory fields:

- `num`: A scalar of the total number of faces n_f in the grid.
- `nodePos`: A $n_f+1\times 1$ array that maps from faces to the half-nodes. The index of all half-nodes belonging to face i is: `G.faces.nodePos(i):G.faces.nodePos(i+1)-1`.
- `nodes`: A mapping from half-nodes to nodes. The index of the node that belongs to half-node i is found at `nodes(i)`.

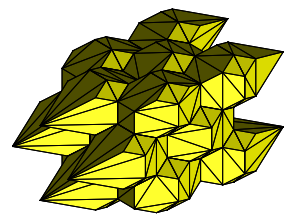
The substruct `G.nodes` contains the following mandatory fields

- `num`: A scalar of the total number of nodes n_n in the grid.
- `coords`: A $n_n \times d$ array containing the coordinates of all nodes. The column size d equals the dimension. The coordinate of node i is `G.nodes.coords(i,:)`.

We will show a simple example of how one can generate all necessary maps. We start by generating a Voronoi diagram using Qhull:

```
n   = 5;
[X,Y,Z] = ndgrid(linspace(0,1,n));
pts = [X(:),Y(:),Z(:)];
pts(1:2:end,1) = pts(1:2:end,1) + 0.1;
[V,C] = voronoin(pts);

% Remove infinity cells
rem = cellfun(@(c) any(isinf(V(c,1))), C);
C   = C(~rem);
```



In the Qhull grid structure, the first vertex is always equal infinity. To obtain a bounded grid, we have removed any cells mapping to this vertex.

The first step is to generate all half-faces. Qhull only contains mappings from cells to vertices, and to obtain the half faces we compute the convex hull of each cell. This will give us a mapping `hf2n` from the half-faces to nodes.

```
cNum = numel(C);          % Number of cells
hf2n    = [];             % Map from half-face nodes to nodes
hf2nPos = 1;              % Map from half-faces to hf2n
facePos = ones(cNum,1);
for i = 1:cNum;
  hull          = convhull(V(C{i},:));
  [hull, localPos] = remParFaces(V(C{i},:), hull);
  hf2n          = [hf2n; C{i}(hull)'];
  hf2nPos       = [hf2nPos; (hf2nPos(end) - 1 + localPos(2:end))];
  facePos(i+1) = numel(hf2nPos-1);
end
```

The Matlab function `convhull(V)` creates a triangulation of the convex hull of the vertices `V`. In a Voronoi diagram, the faces (and therefore half-faces) are in general polygons. We have made a function `remParFaces` that merges all triangles that lie in the same plane to one polygon. It returns an array, `hull`, of size (number of nodes in cell) $\times$ 1 and an array, `localPos`, of size (number of half-faces in cell + 1) $\times$ 1. Specifically, the local nodes of half-face j is `hull(localPos(j):localPos(j+1)-1)`. By local nodes we mean the nodes of cell i.

The next step is to create the mapping from half-faces to faces. If a face is shared by two cells, there are two half-faces mapping to the face. We will find these mappings by finding half-faces that contain the same nodes. First, we find the number of vertices of each half-face.

```
fSize   = diff(hf2nPos);     % Number of nodes of each half face
[~,ias,ics] = unique(fSize); % The unique sizes
```

If two half faces share the same face, they must have the same number of vertices. We therefore iterate over all half face sizes and find the half faces of that size.

```
nodes   = [];
nodePos = 1;
faces   = zeros(size(hf2nPos,1)-1,1);
for i = 1:numel(ias)
  testPos  = fSize(ias(i))==fSize;
```

To find the corresponding nodes, we use the MRST routine `mcolon(A,B)`. This is a generalization of the Matlab operator `colon` that works on arrays. For two arrays we have `mcolon(A,B) = [A(1):B(1), A(2):B(2), ..., A(end):B(end)]`.

```
  from     = hf2nPos([testPos;false]);
  to       = hf2nPos([false;testPos]) - 1;
  map      = mcolon(from, to);
  nTmp     = reshape(hf2n(map),fSize(ias(i)),[])';
```

Each row in the matrix `nTmp` is a mapping from a half face to its nodes.

We now find the unique rows of the matrix `nTmp`, and let this be the mapping from faces to nodes. The information about which half faces maps to which face is found at the third output of the `unique` routine in Matlab.

```
  [~,ia,ic]= unique(sort(nTmp,2), 'rows');
  newNodes = nTmp(ia,:)';
  nodes    = [nodes; newNodes(:)];
  faces(testPos) = ic+numel(nodePos)-1;
  locPos   = cumsum(repmat(fSize(ias(i)),[size(newNodes,2),1]));
  nodePos  = [nodePos; nodePos(end) + locPos];
end
```

The only mapping we now lack is the mapping `G.faces.neighbors`. We use a MRST function to create the mapping from half faces to cells:

```
cellNo = rldecode(1:cNum, diff(facePos), 2).';
```

Half face i belongs to cell number `cellNo(i)`. If two half-faces j, k maps to the same face this means that cell `cellNo(j)` and `cellNo(k)` are neighbors.

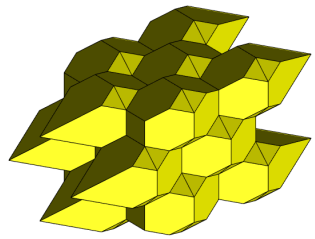
```
fNum = numel(nodePos)-1;
for i = 1:fNum
  neigh = faces==i;
  if sum(neigh)==2
    neighbors(i,:) = cellNo(neigh);
  else
    neighbors(i,:) = [cellNo(neigh),0];
  end
end
```

We have now created all mappings, and we put it together to make the final grid:

```
G.cells.num       = cNum;
G.cells.facePos   = facePos;
G.cells.faces     = faces;

G.faces.nodePos   = nodePos;
G.faces.num       = fNum;
G.faces.nodes     = nodes;

G.nodes.num       = size(V,1);
G.nodes.coords    = V;
G.faces.neighbors = neighbors;
```



3D Fault

An example of 3D reservoir with a single fault is shown in Figure 5.4. The reservoir contains three layers of different types of rock, with the highest porosity in the middle layer. The rock has shifted diagonally along a fault. We create a grid of the reservoir using the method described in Section 4.3. We start by generating the fault sites:

```
F = createFaultGridPoints3D(fDt,rho);
```

The syntax here is equivalent as for the 2D case. The first argument `fDt` is a cell array of surface triangulations. In this case the cell array only contains one triangulation. The second argument `rho` is a function that gives the radii of the circles. We create the reservoir sites as the Cartesian grid, and then remove any sites inside the generating circles:

```
rx = 0:dt:xmax; ry = 0:dt:ymax; rz = 0:dt:zmax;
[X,Y,Z] = ndgrid(rx,ry,rz);
rSites  = [X(:), Y(:), Z(:)];
[rSites,removedRes] = faultSufCond(rSites,F.c.CC,F.c.R);
```

Putting all sites together we can generate the grid. To generate a clipped Voronoi diagram we use our implementation of the clipping algorithm from Section 3.2:

```
sites = [F.f.pts; rSites];
G = clippedPebi3D(sites,boundary);
```

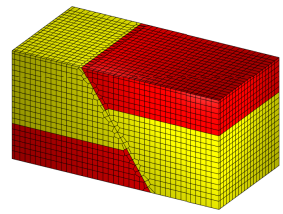


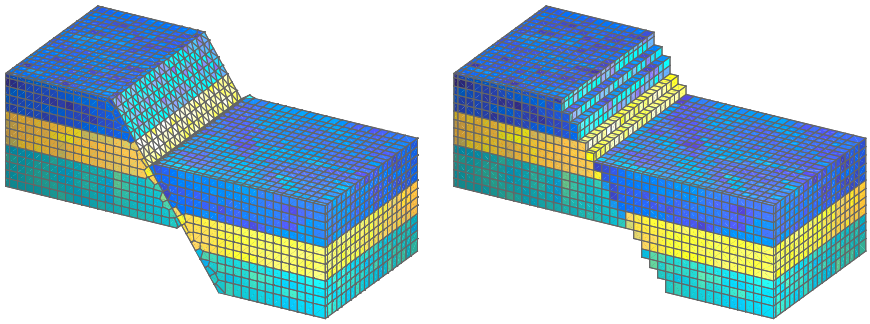
Figure 5.3: A faulted reservoir. Red cells are auxiliary cells.

The grid is shown in Figure 5.3. The red cells are auxiliary cells that are inactive. The yellow cells on the right side of the fault can be imagined to have shifted $\frac{1}{3}$`zmax` down along the fault.

We now set the porosities of the rock. The porosity is set as an approximated Gaussian field. We create a set of normally distributed variables corresponding to each cell. Then we convolve them with a Gaussian kernel. The porosity of each layer is generated separately.

```
nx = numel(rx); ny = numel(ry); nz = numel(rz);
p1 = gaussianField([nx,ny,nz*2/9],[0.05,0.2],3,10)
p2 = gaussianField([nx,ny,nz*2/9],[0.4,0.6], 3,10)
p3 = gaussianField([nx,ny,nz*2/9],[0.2,0.4], 3,10)
```

The thickness of the reservoir is $\frac{2}{3}$`zmax`, disregarding the auxiliary cells. All layers have the same thickness $\frac{2}{9}$`zmax`. A comparison of the grid generated and a corresponding Cartesian grid is shown in Figure 5.4. A detailed study of the effects these two grid types have on flow simulations is given by Wu and Parashkevov [54].



(a) (b)

Figure 5.4: A close up view of a single fault. The reservoir contains three layers with distinct porosities. The cells in the two grids are equal away from the fault. (a) A grid created using our implementation. The fault is exactly represented by the grid. (b) A Cartesian grid. The fault is represented by a zig-zag pattern.

CHAPTER 6

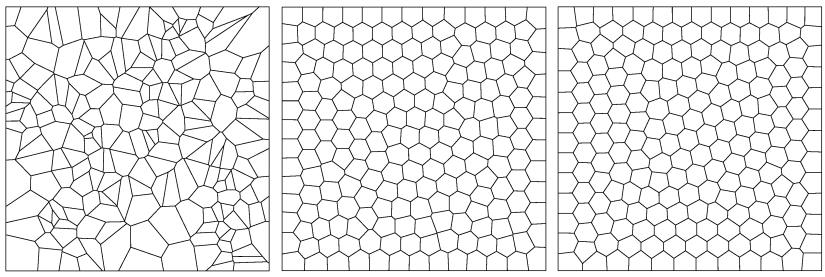
Numerical Tests

6.1 CVD optimization

We create a CVD by finding the minimum of the CVD energy function given by Equation 3.2. We use our Matlab implementation of the L-BFGS method described in Section 2.4 to find the minimum. In the following we present two cases; a CVD for the unit square, and a CVD for the unit Cube. The initial guesses for the sites are found by the random function in Matlab. The convergence tolerance are in the two cases set to $||\nabla f(\boldsymbol{x}_k)|| \leq ||\nabla f(\boldsymbol{x}_0)|| \cdot 10^{-6}$

A 2D PEBI-grid before, during, and after optimization is shown in Figure 6.1. The domain is the unit square. There are 200 sites. The number of stored vectors for the Hessian approximation is set to $m = 10$. We see that the grid is fairly good already after 20 iterations.

In another example we set our domain equal the unit cube. The number of stored vectors for the Hessian approximation is set to $m = 5$. Figure 6.2 shows the converged CVT grid for 100, 500 and 1000 cells. It also shows objective function and the ℓ^2-norm of the gradient for each iteration.



(a) Initial PEBI-grid. (b) PEBI-grid after 20 iterations. (c) Converged PEBI-grid (140 iterations).

Figure 6.1: The optimization of a PEBI-grid using L-BFGS. The initial sites are 200 randomly chosen points in the unit square. The convergence criterion is set to $\nabla f_k < \nabla f_0 \cdot 10^{-6}$, and we use 10 vectors in the Hessian approximation.

6.2 Example Grids

Sharp Intersections

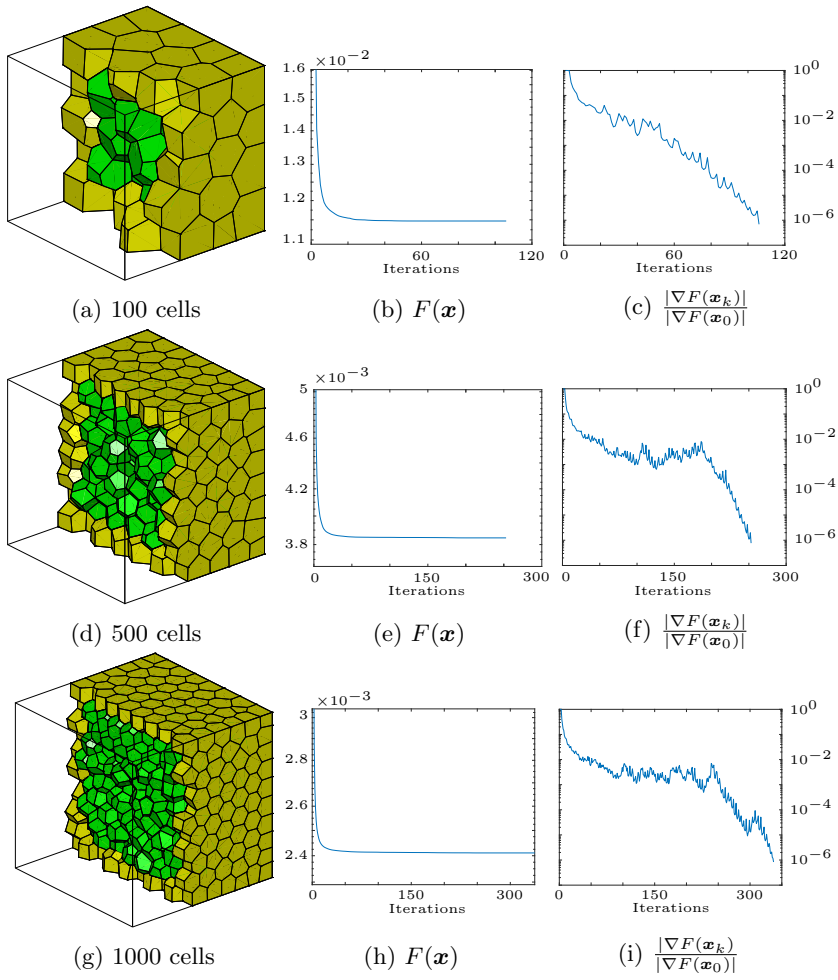
Figure 6.3 shows the intersection of faults at different angles. The first case shows an intersection at 90 degrees. In this case the fault condition is not violated for any circles, thus, no fault sites are merged. In the second case two faults intersect at 45 degrees. This is sharp enough that two fault sites are merged; one on both sides of the intersection. The last intersection is at 20 degrees. This is so sharp that multiple circles are merged. Because circles are merged, the faults are not represented exactly around the intersection, but the error is small compared to the cell sizes.

Three Intersecting Faults

We create a grid with three faults and with two fault intersections. One of the faults ends in the first intersection. At the other intersection, one fault just barely crosses the other, leaving a tiny finger on the other side of the fault. This example grid is taken from Ding and Fung [12], and their grid can be seen in Figure 6.4c. We create the fault paths:

```
l = {[0.1,0.42;   0.4,.55; 0.7,0.65], ...
     [0.8,0.13;   0.6,0.4; 0.55,0.6], ...
     [0.42,1.08; 0.45,0.9;  0.5,0.8; 0.58,0.6]};
gs = 1/30;
```

We generate a grid with a Cartesian background grid using the routine

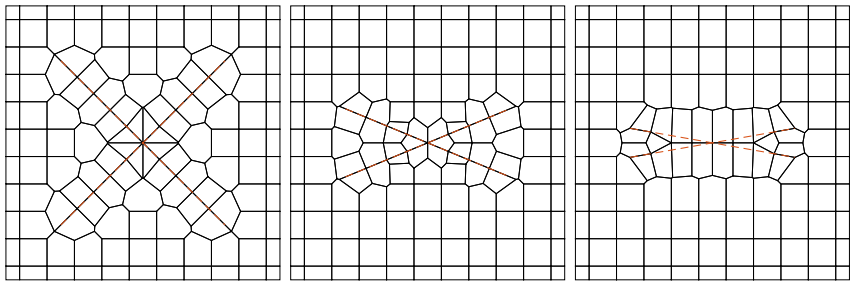


(a) 100 cells (b) $F(\boldsymbol{x})$ (c) $\frac{|\nabla F(\boldsymbol{x}_k)|}{|\nabla F(\boldsymbol{x}_0)|}$

(d) 500 cells (e) $F(\boldsymbol{x})$ (f) $\frac{|\nabla F(\boldsymbol{x}_k)|}{|\nabla F(\boldsymbol{x}_0)|}$

(g) 1000 cells (h) $F(\boldsymbol{x})$ (i) $\frac{|\nabla F(\boldsymbol{x}_k)}{|\nabla F(\boldsymbol{x}_0)|}$

Figure 6.2: CVD grids of a cube with different number of cells. The green cells are inner cells while the yellow cells are boundary cells. Function values and gradient norms for the CVD energy function are shown on the right.



(a) (b) (c)

Figure 6.3: Intersection of faults at different angles. Dashed orange lines are the faults. (a) Faults intersecting at 90°. No circles or fault sites are merged. (b) Faults intersecting at 45°. No circles are merged, but two pairs of fault sites are merged. (c) Faults intersecting at 20°. Four pairs of circles are merged; two on both sides of the intersection.

`compositePebiGrid` with the preset settings:

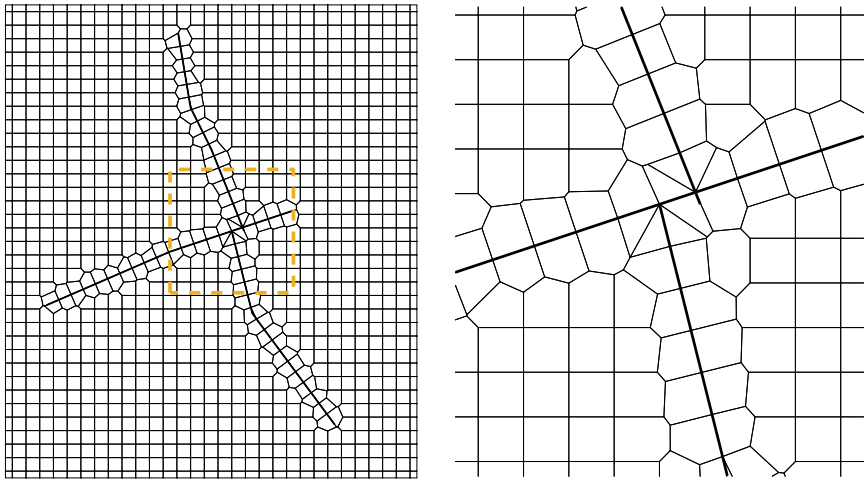
```
G = compositePebiGrid([1/30,1/30],[1,1.15],'faultLines',1);
```

The result is shown in Figure 6.4. The small finger that crosses the fault in one of the intersections is ignored by the gridding algorithm. Any fault paths that are shorter than 80% of the fault grid size is considered to small too matter, and therefore not gridded. This is a fixed value in our implementation. The main difference between our algorithm and the one presented by Ding and Fung [12] is how we handle fault intersections. Notice how all faults are exactly represented in our grid, whereas only one fault is exactly represented over the intersections in the grid by Ding and Fung [12].

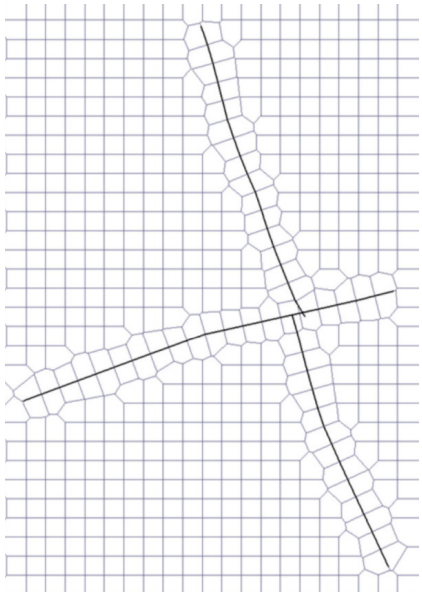
Complex Fault Network

We create a grid of a highly faulted reservoir. Branets et al. [6] create a grid of the same reservoir using constrained Delaunay triangulation algorithm. A comparison of this algorithm and our method is shown in Figure 6.5. Notice how the grid generated by Branets et al. [6] has much more congested cells around the faults.

Since this reservoir does not have a square boundary, we can not use the wrapping function `pebiGrid` to generate our grid. But we are not in vain. We start by cutting the faults at all intersections:

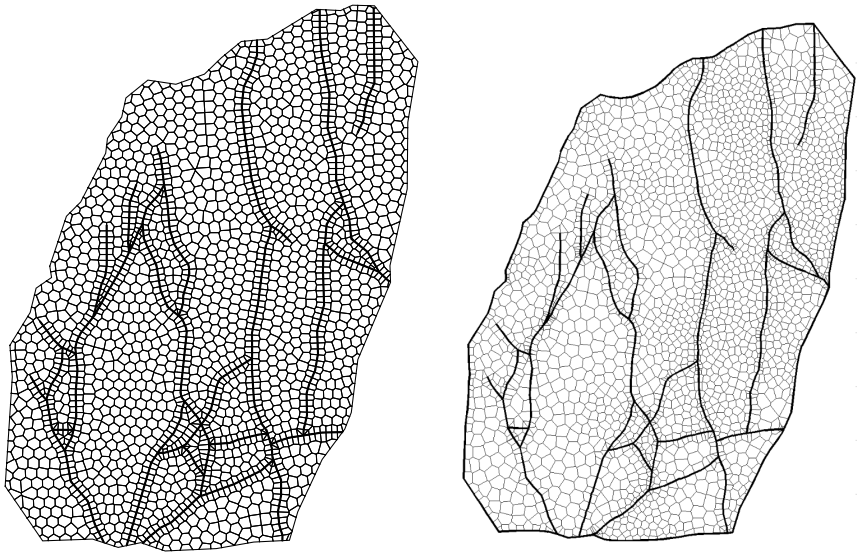


(a) (b)



(c)

Figure 6.4: A reservoir where three faults are intersecting. (a) A grid generated by our implementation. (b) A close up view of the intersection in (a). (c) A grid generated by Ding and Fung [12].



(a) (b)

Figure 6.5: A reservoir with a complex fault network. (a) A grid created using our implementation. (b) A grid generated by Branets et al. [6].

```
[fault, fCut, ~] = splitAtInt(fault, {});
```

The boundary of the reservoir is a polygon stored in the variable `bdr`. We scale the fault grid size by the size of the reservoir and create the fault sites:

```
fGs = max(max(bdr))/70;
F = createFaultGridPoints(fault, fGs,'fCut',fCut);
```

We create the reservoir sites using the third party software DistMesh [43]. To define the boundary, we use a the signed distance function `signDist`. This function returns the distance to the bounding polygon, with a negative sign for points inside the polygon:

```
rectangle = [min(bdr); max(bdr)];
fixedPts = [F.f.pts;bdr];              % Add faults as fixed points
uni = @(p,varargin) 2*ones(size(p,1),1); % Desity function
Pts = distmesh2d(@signDist ,uni, fGs, rectangle,...
                 fixedPts, 'bdr', bdr);
```

To use the MRST routine `pebi`, one needs to create the Delaunay triangu-

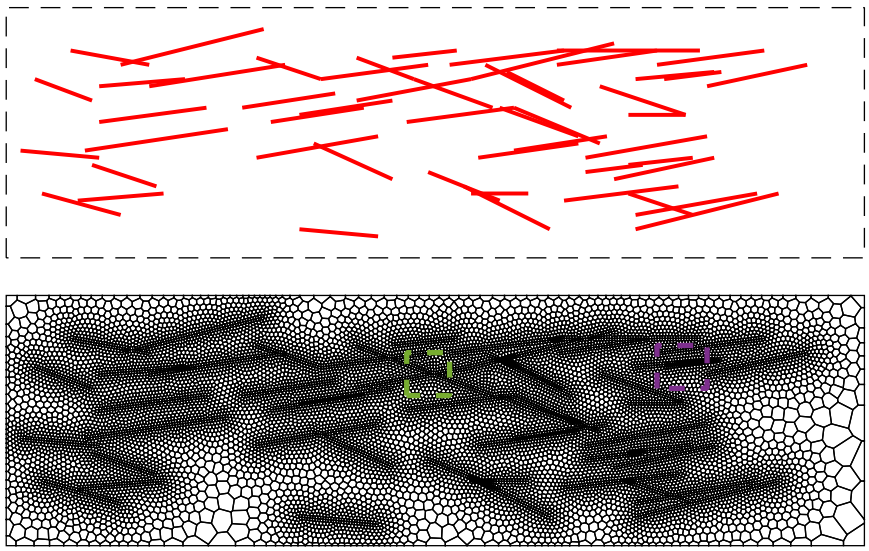


Figure 6.6: A fractured reservoir. The fractures are from the dataset `statistical_fractures` in the `hfm` module. Notice that the figures are rotated 90°. The fractures are shown in the top figure, while a grid created using our implementation is shown in the bottom figure. See Figure 6.7 for a zoomed view.

lation of the sites. This routine has some problems when the circumcircle of a Delaunay triangle is outside the boundary. We have therefore implemented our own PEBI-grid routine that creates the PEBI-grid directly from the intersection of bisectors:

```
G = clippedPebi2D(Pts,bdr);
```

This routine clips the Voronoi diagram defined by the sites `Pts` against the boundary `bdr` and returns a valid MRST grid.

Statistical Fractures

The module `hfm`, which is released in MRST 2016a, contains a dataset, `statistical_fractures`, of a set of fractures. We will create a grid of these fractures, where we treat the fractures as faults, that is, the fractures are traced by edges in the grid. We start by importing the dataset and transform the fractures into a cell array:

```
load('statistical_fractures.mat')
offset = 2;
f = mat2cell(fl, ones(size(fl,1),1), size(fl,2)).';
f = cellfun(@(c) reshape(c', 2,[])' + offset,f,'un',false);
```

The reservoir is 35 m by 120 m, and is shown in Figure 6.6. The reservoir contains 51 fractures that are aligned more or less in the same direction. This results in many sharp intersections.

Our algorithm does not have any method for handling fractures that are close by each other, but not intersecting. If two fractures are closer to each other than the fracture grid spacing, we can not guarantee conformity. To handle these cases, we decrease the fracture grid sizes for fractures in these areas, as shown in Figure 6.7b. Notice that the fracture cell sizes are only smaller where two fractures are close to each other. To decrease the fracture cell size in an area, we decrease the distance between the circles that are placed along these fractures. We define the density function as follows:

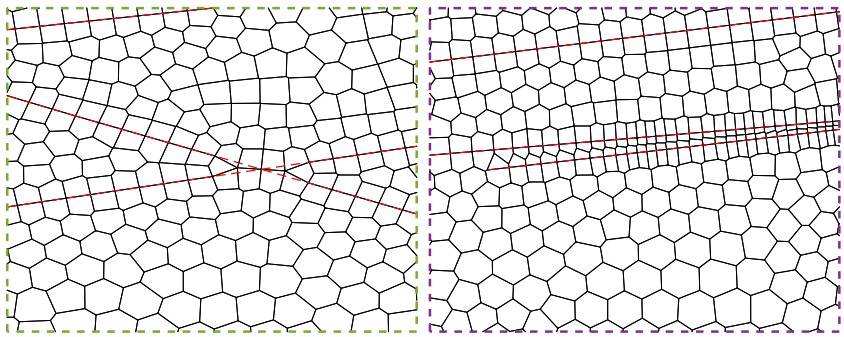
```
eps    = [3,5,1,1,1,2,2,1.5,1,1.5,1.5,1.5,1,1.5,1.5,2.5,1.5];
refPts = [17.4,79.5; 15.0,71.6; 25.9,98.7; 25.8,97.9; ...
          25.7,97.2; 25.7,96.4; 25.5,94.7; 25.3,92.8; ...
          9.1,87.0;  13.0,87.6; 13.1,88.5; 9.0 ,66.2; ...
          28.95,77;  26.0,70.0; 20.8,69.2; 24.9,24.8; 26.8,38.9];
amp    = [.5,.7,.3,.3,.4,.4,.4,.6,.3,.45,.45,.2,.3,.25,.55,.45,.6];
faultRho = @(p) min(ones(size(p,1),1), ...
   min(bsxfun(@times, amp,              ...
   exp(bsxfun(@rdivide, pdist2(p,refPts),eps))),[],2));
```

This density function has to be specified manually, which is very cumbersome for large datasets. Alternatively, we could define a very fine fracture grid size, but this would result in equivalently many more grid cells.

We create the grid by optimizing the reservoir sites:

```
G = pebiGrid(15,[35,120],'faultLines',1,'faultGridFactor',1/40,...
             'circleFactor',0.62,'faultRefinement',true,        ...
             'faultEps',5,'faultRho', faultRho,'mergeTol',0.12);
```

A option we have not yet seen is the `'mergeTol'` option. This sets the merge tolerance from Section 4.1; two conflict circles are merged if their relative distance is smaller than `'mergeTol'`. The default value of `'mergeTol'` is 0, but we have set it higher to treat some of the very sharp intersections.



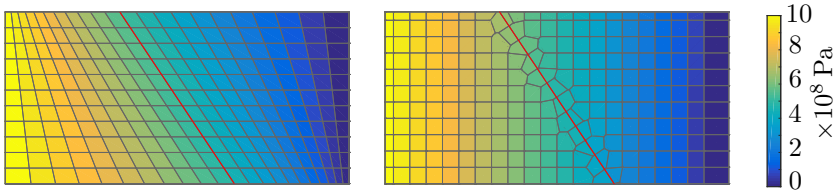
(a) (b)

Figure 6.7: Zoomed view of the grid shown in Figure 6.6. (a) Fractures intersecting at sharp angles. (b) Almost parallel, but not intersecting fractures.

The generated grid is shown in Figure 6.6. A close up view of the green and purple squares is shown in Figure 6.7.

6.3 Grid-Orientation Effects

Grid-orientation effects are a common example for how the grid can affect the numerical solution. These effects can cause the numerical scheme to converge to the wrong solution. We create a small reservoir sector of dimension $20\,\mathrm{m} \times 10\,\mathrm{m} \times 10\,\mathrm{m}$. We assume there is an inclined fault in the middle of the sector that the grid cells should conform to. We create two grids, one corner point grid and one unstructured grid using the method presented in Section 4.3. In the corner-point grid, cells at different heights will be stretched or compressed differently, which will introduce the grid orientation effects. We introduce a pressure gradient by setting the pressure difference from the left to the right boundary to $10^8\,\mathrm{Pa}$. This should result in a linear change in pressure in the x direction, and a constant pressure in the z and y directions. Figure 6.8 shows the calculated pressure in the reservoir using the Two Point Flux Approximation (TPFA) discretization. We clearly see that the pressure is not constant in the z direction for the corner-point grid. The unstructured grid gives a correct pressure solution. By using a consistent discretization, such as, MPFA, mimetic or mixed FEM, we do not get the same gird-orientation effects as for TPFA. For further reading, we refer the reader to [1, 54]



(a) (b)

Figure 6.8: A reservoir with a single fault. The figure shows a cross section of the reservoir in the $xz-$plane. The color shows the pressure in each cell. The red line is a fault the grids conform to. The pressure difference on the left and right boundary is 1×10^8 Pa and there are no-flow conditions on the top and bottom. The corner-point grid in (a) shows grid-orientation effects, whereas the unstructured grid in (b) does not.

6.4 Flow Simulation in a Fractured Reservoir

Fractures can have a significant impact on the flow in a reservoir. We will investigate two methods for representing fractures in a reservoir. The first method seperates the physical domain from the computational domain. A fracture is modeled as a line segment and we trace the fractures by edges in the grid. We call this for lower-dimensional fractures. Karimi-Fard et al [28] show how one can create a discretization of lower dimensional fractures. In the second representation, we explicitly create grid blocks of the fractures. We call this representation for volumetric fractures. Sun and Schechter [48] present several numerical examples of flow in reservoirs using this method. In our gridding module we can easily model both volumetric and lower-dimensional fractures. For volumetric fractures we create a grid by modeling the fractures as wells. Around the fractures we place a protection layer to obtain the correct aperture. For lower-dimensional fractures we model the fractures as faults. To solve for the pressure in reservoirs with volumetric fractures, we use the MRST routine `incompTPFA`. We use the explicit transport solver `explicitTransport` to solve the transport equations. For the lower-dimensional fractures we use the equivalent methods from the the `dfm` module; we use `icompTPFA_DFM` to solve for pressure and `explicitTransport_DFM` to solve the transport equations.

The first example we will look at is a single fracture that connects two wells. The aperture for each fracture cell block is chosen randomly between 1 cm and 5 cm. The dimension of the reservoir is 100 m $\times$ 100 m. Initially, the reservoir is filled with oil, but one well is injecting water. The difference in bottom hole pressure of the two wells is 10^8 Pa. The rock permeability is

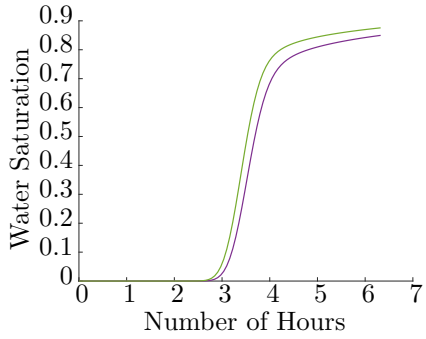
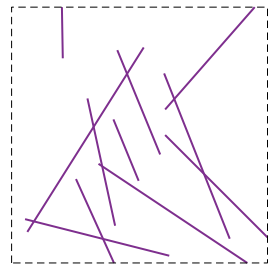
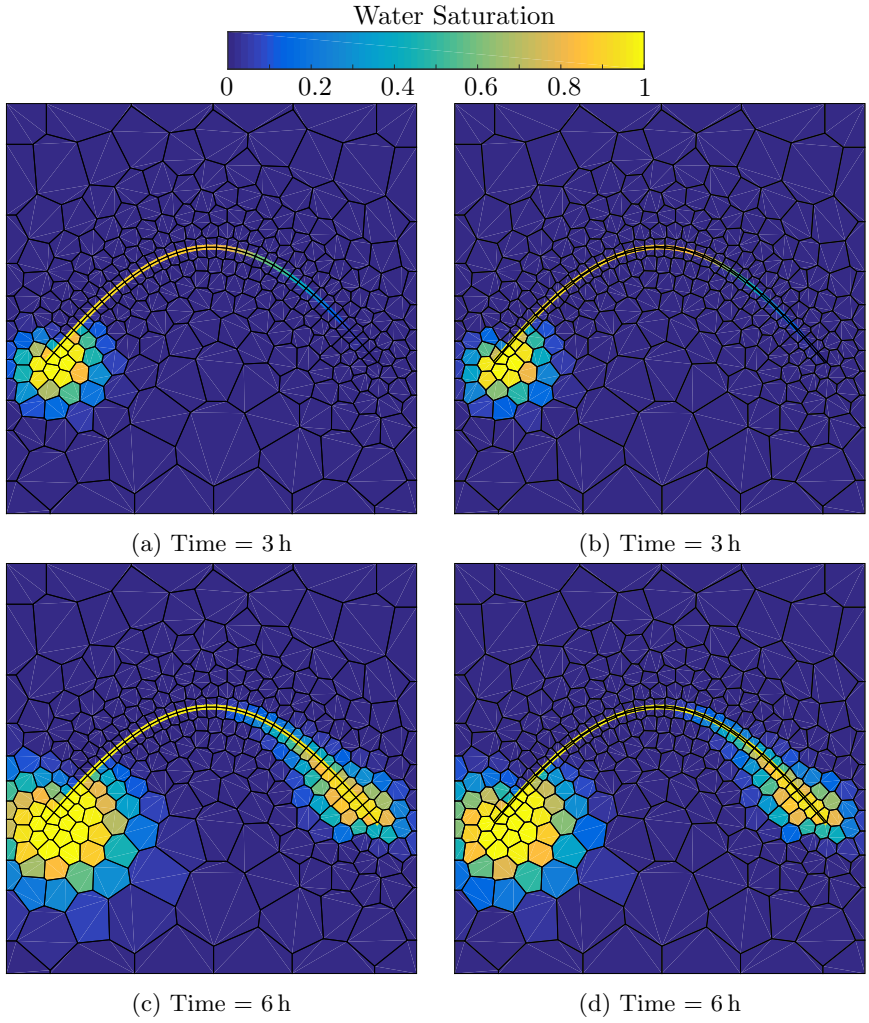


Figure 6.9: The water saturation in the producing well. The green line is water saturation for the volumetric fractures, while the purple line is the water saturation for the lower-dimensional fractures.

set to 1 md, while the porosity is set to 0.01. We model the permeability of the fracture from the parallel plate assumption [29] and set the permeability as $\frac{1}{12}a^2$, where a is the aperture. Figure 6.9 shows the saturation of the producer. Figure 6.10 shows a plot of the water saturation in the reservoir after 3 and 6 hours.



To compare the two methods further, we create a reservoir with several long fractures. The fractures are shown in shown in the figure on the right. We create two grids; one with volumetric fractures, and one with lower dimensional fractures. The aperture of the fractures is 1 mm. The reservoir sites of the reservoir are found by optimizing the dual Delaunay triangulation and are the same for both grids. The reservoir is 1000 m in the x- and y-directions. We set an input flux of water on the lower boundary ($y = 0$). The flux is $0.1\,\mathrm{m}^3/\mathrm{day}$ distributed evenly over the whole lower boundary. There are no flow over the left and right boundaries. The pressure on the upper boundary is set to zero. The permeability of the reservoir is set to 10 µd and the porosity to 0.001. The difference in saturation of the two grids after 500 days is shown in Figure 6.12. As a measure of the difference in saturation we create a vector S. Each element in S is the difference in saturation for a cell in the two grids. The difference for each time step is shown in Figure 6.11.



(a) Time = 3 h (b) Time = 3 h

(c) Time = 6 h (d) Time = 6 h

Figure 6.10: The water saturation in the reservoir with a single fracture. A well is injecting water at the left end of the fracture, while a well is producing at the right end. The left column shows the water saturation for the lower-dimensional fractures, while the right column shows the water saturation for the volumetric fractures. Notice that the saturation in the fracture is plotted wider than the aperture for visibility.

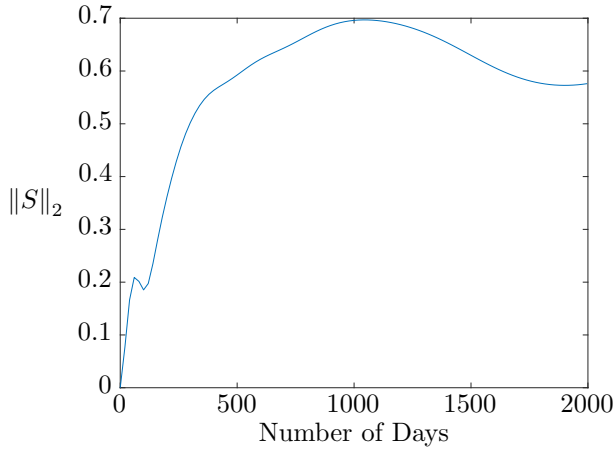
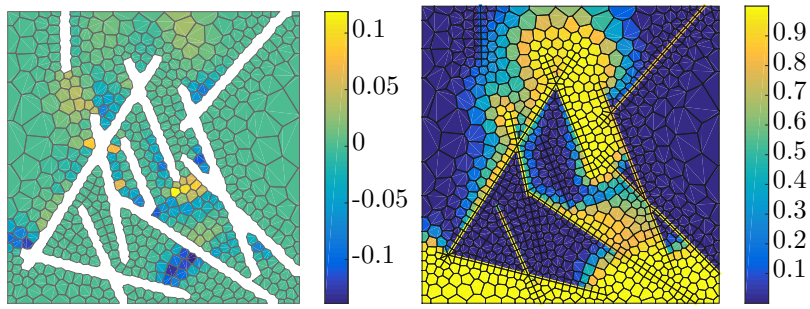


Figure 6.11: The relative error in water saturation. The error S is a vector where the elements are the difference in water saturation of the lower-dimensional fracture grid and the volumetric fracture grid for each cell.



(a) (b)

Figure 6.12: The water saturation in a fractured reservoir after 500 days. Water is injected at the lower boundary, while there are no flux over the left and right boundary. (a) The difference in water saturation between the volumetric fracture grid and the lower-dimensional fracture grid. (b) Water saturation in the reservoir for the edge centered grid. The saturation in the fractures are plotted wider than the aperture for visibility.

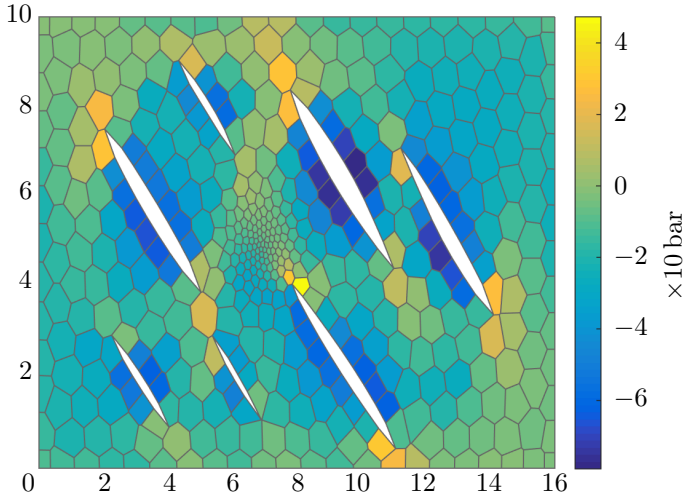


Figure 6.13: The maximum stress in the cells. A pressure of 10 bar is set inside the fractures, and there is a no displacement boundary condition.

6.5 UPR in the Literature

Our `UPR` module has already been used by several researchers. Halvor Møll Nilsen at Sintef ICT has used `UPR` to generate grids that he have used to solve mechanics. An example is shown in Figure 6.13. In this example, a pressure of 10 bar is set inside the fractures. Young's modulo is set to 10^9, and the Poisson's ratio for plane strain to 0.3. There is a no displacement condition on the boundary. The figure shows the maximum stress in the cells.

Bao et al. [2] use `UPR` to generate grids simulating polymer flooding. They use both `compositePebiGrid` and `pebiGrid` and compare the grids with corresponding coarse and fine Cartesian grids. Klemetsdal [30] generates unstructured PEBI-grids with `UPR`. He uses these grids to test his implementation of the virtual element method.

CHAPTER 7

Conclusion and Recommendations for Further Work

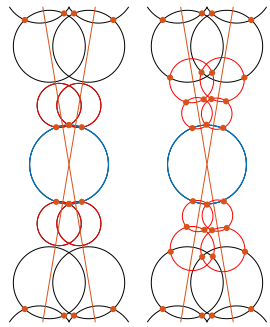
7.1 Summary and Conclusions

The goal of this thesis has been to implement a new module for creating unstructured grids in MRST. The module we have created is highly automated and requires minimal input and interaction from the user. It allows for unstructured PEBI-grids in both 2D and 3D. The grids generated by the module have the ability to conform to geological structures. Two different types of conformity are supported; faces following surfaces, and centroids following surfaces. The gridding routines in 2D are very versatile and can handle several hard cases, such as pinch-outs, intersection of multiple faults, and intersection of wells and faults. The module also supports grids in 3D that conform exactly to faults, however, conformity is not guaranteed when two faults intersect.

Matlab has support for using the Qhull algorithm for creating Voronoi diagrams in 3D. These diagrams extend to infinity, which is undesirable when we wish to use the corresponding grid for numerical simulations. Creation of clipped Voronoi diagrams are therefore studied in Chapter 3, and the method for generating clipped Voronoi diagrams is implemented in Matlab. We also presented two different approaches for creating an optimized Voronoi diagram. The first approach formulates an optimization problem which can be solved by quasi-Newton methods. At the optimum

point, the Voronoi sites coincide with the corresponding cell centroids. The second method optimized the dual Delaunay triangulation, and in this way created a Voronoi diagram with evenly shaped cells.

Chapter 4 presents a novel method to generate PEBI-grids that conform to faults. We discuss how to create grids conforming to wells and faults in 2D, 2.5D, and 3D. The 2D method is able represent intersections of faults exactly. The method presented, also handles intersection of multiple faults. When two faults intersect at very sharp angles, we have chosen to merge them in a small area around the intersection to create more uniform cells. Depending on the problem, this might not be the optimal method. Another approach is shown on the right. For the cases that in our algorithm will merge two circles, we could split each circle in two. By this method, we could represent the geometry exactly, however, it would result in cells with very sharp corners at the intersection. In 3D, we generalize the method used in 2D, which, to the best of our knowledge, has never been done before. The method honors faults if they do not intersect. At the intersection of two or more faults, only one of them is traced exactly by faces of the grid.



Chapter 5 shows the most important implementation details of our new MRST module. We also give examples of how it can be used efficiently. For the generated PEBI-grids, it is not uncommon that some cells have very small faces, especially when faults intersect. It would be interesting to implement a routine that removes these very small faces. In 2D, this could be done by merging the two vertices of the face (remember a face is a line segment) to a single vertex. The resulting grid would not be strictly Voronoi, but it might have better flow properties.

Chapter 6 shows numerical simulations. We first show the convergence of the L-BFGS algorithm for the optimization problem formulated in Section 3.4. We then show a few examples of how our implementation handles cases that are hard to grid. A comparison with gridding algorithms already found in the literature is done. Our module creates equivalent or better grids. We also demonstrate the advantage of an unstructured grid for reducing grid-orientation effects. Lastly, we model a fractured reservoir. We compare two different methods for modeling fractures. In the first method, the fractures are represented explicitly by cells in the grid. In the second, we create the grid as if the fractures are faults. The physical

domain and computational domain are separated, and the fault edges are given an imaginary volume. Our experiments shows that the two methods give equivalent results. The advantage of modeling the fractures explicitly by cell block, is that we can use any standard reservoir solver to model the fractures. For lower-dimensional fractures we often have to use specialized solvers.

7.2 Recommendations for Further Work

To get a truly versatile gridding module, one should implement a method for handling intersections of faults in 3D. It would be interesting to see if one could generalize the method for handling intersections in 2D. This would presumably add some new challenges; a fault site in 3D is created by intersecting three balls, so merging fault sites at the intersection would be a grater challenge.

Gridding in 3D is generally hard, but many times not necessary. 2.5D grids are often enough to create an accurate model of the reservoir. It would therefore be interesting to look closer at how we can construct general pillar grids by extruding unstructured 2D PEBI-grids.

A hard case to grid, which we have not studied, is when two faults are almost parallel, very close too each other, but not intersecting. This is common for fractures in a reservoir. Creating a very fine grid can solve this problem, but this might lead to unreasonable many cells in the grid.

We have presented two methods for optimizing the reservoir sites. In both cases, we held the fault and well sites fixed. A possible improvement would be to let the fault and well sites move; but only along their associated paths. This could improve the grids, especially when there are several intersections close by each other.

A grid should not only conform to features. It is maybe just as important that the grid conforms to flow in the reservoir. There have been some studies on creating grids that adapts to flow using a modified version of the CVD energy function[36]. Using the framework from our module, we believe this could be an achievable extension.

In our MRST module, we only have support for unstructured PEBI-grids. Other types of unstructured grids that conform to faults exist, and it would be exiting to expand the module with support of the recently developed cut-cell method.

Appendices

APPENDIX A

Computing the Gradient of a Voronoi Cell Cut by a Fault

A method for creating a Voronoi diagram conforming to faults is presented by Merland et al. [37, 38]. An objective function is presented, and several numerical experiments show that the method performs well. Unfortunately, details about how one can compute the gradient of the objective function are left out. In this section, we will present the objective function together with a method for computing its gradient. We will go through all steps in the computations, however, we have not implemented the results in our MRST module, nor tested the results numerically.

We define our objective function by adding a term to the CVD energy function penalizing cells that are cut in two by faults. Let $\{\boldsymbol{p}_i\}_{i=1}^n = P$ be a set of Voronoi sites in $\mathbb{R}^3$ and $\mathcal{V}$ the associated Voronoi diagram. Consider a fault cutting a cell v_{p_i}. We define $V_{p_i}^{\text{inn}}$ to be the volume of the cell that is on the same side as $\boldsymbol{p}_i$. Equivalently we define $V_{p_i}^{\text{out}}$ to be the volume that is opposite side of the fault, as shown in Figure A.1. The penalty function is the sum of all outer volumes. The complete objective function is

$$\tilde{F}(P) = F(P) + \beta F_V(P) = \sum_{i=1}^{n} \int_{V_{p_i} \cap \Omega} \rho(\boldsymbol{y}) ||\boldsymbol{y} - \boldsymbol{c}_i||^2 \, \mathrm{d}\boldsymbol{y} + \beta \sum_{i=1}^{n} V_{p_i}^{\text{out}},$$

where β is a constant that weights the importance of conformity. Merland et al. [38] report that β should be chosen such that $F \approx \beta F_V$. The gradient

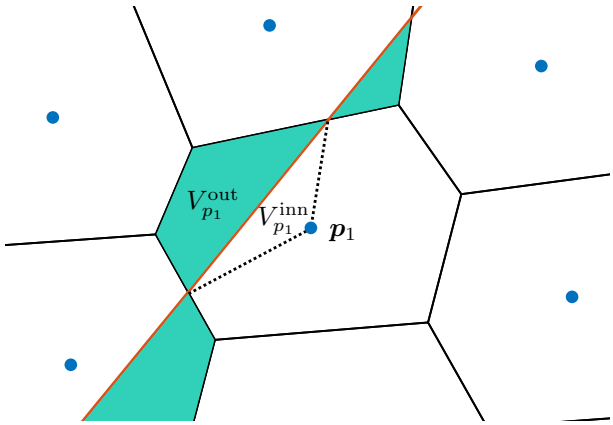


Figure A.1: Cell i is divided into two volumes, $V_{p_i}^{\text{out}}$ and $V_{p_i}^{\text{inn}}$ by a fault.

of the objective function is

$$\nabla \tilde{F}(P) = \nabla F(P) + \beta \nabla F_V(P).$$

The gradient of the CVD energy function $\nabla F(P)$ is [14, 26]:

$$\frac{\partial F}{\partial \boldsymbol{p}_i} = 2m_i(\boldsymbol{p}_i - \boldsymbol{c}_i),$$

We will now focus on finding the gradient of the penalty function $\nabla F_V(P) = \sum_{i=1}^{n} \nabla V_{p_i}^{\text{out}}$

Geometry

Suppose we have fault $\mathcal{F}$ cutting through our domain. The fault is described by a surface triangulation. The faces of the triangulation $\{t_i\}_{i=1\ldots m}$ are subsets of the planes $\boldsymbol{N}_{t_i}^{\top}\boldsymbol{x} + b_{t_i} = 0$. We compute the intersection of the Voronoi diagram and the fault by the method discussed in Section 3.1. The vertices from the intersection together with the Voronoi vertices on the opposite side of the fault defines the outer volume of a cell. After the intersection is found, the outer volume of a cell can be decomposed into tetrahedrons by joining $\boldsymbol{p}_i$ to three of it's vertices $\boldsymbol{C}_1$, $\boldsymbol{C}_2$ and $\boldsymbol{C}_3$, as shown in Figure A.2. The observant reader will notice that the union of these tetrahedrons is larger than the outer volume. We fix this by giving all tetrahedrons with three vertices on the surface $\mathcal{F}$ a negative volume. The sum of the volume of all tetrahedrons will then equal the outer volume. The three vertices $\boldsymbol{C}_1$, $\boldsymbol{C}_2$, $\boldsymbol{C}_3$ can be of four different types [31],

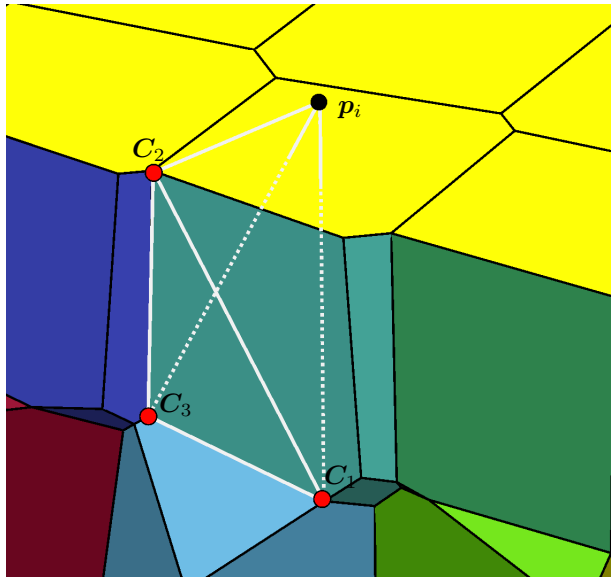


Figure A.2: Creating a tetrahedron of the outer Volume. White lines shows the integration tetrahedral. The yellow faces are the intersection of the Voronoi cells and the inner boundary. Only the outer Volumes of the Voronoi cells are shown.

A) a vertex from $\mathcal{F}$,

B) intersection of one bisection $[\boldsymbol{p}_i, \boldsymbol{p}_j]$ and two planes $(\boldsymbol{N}_1, b_1), (\boldsymbol{N}_2, b_2)$,

C) intersection of two bisections $[\boldsymbol{p}_i, \boldsymbol{p}_j]$, $[\boldsymbol{p}_i, \boldsymbol{p}_k]$ and one plane $(\boldsymbol{N}_1, b_1)$,

D) intersection of three bisections $[\boldsymbol{p}_i, \boldsymbol{p}_j$, $[\boldsymbol{p}_i, \boldsymbol{p}_k]$, $[\boldsymbol{p}_i, \boldsymbol{p}_l]$.

Let $V^T(\boldsymbol{p}_i, \boldsymbol{C}_1, \boldsymbol{C}_2, \boldsymbol{C}_3)$ be the volume of the integration tetrahedral. The outer volume $V_{p_i}^{\text{out}}$ and its gradient $\nabla V_{p_i}^{\text{out}}$ are found by summing over all integration tetrahedrons. The gradient of a tetrahedron is stated by Lévy and Liu [31], and Parigi and Piastra [42] give a detailed derivation.

Gradient of V^T

By the chain rule we obtain

$$\nabla_{\boldsymbol{p}} V^T(\boldsymbol{p}_i, \boldsymbol{C}_1, \boldsymbol{C}_2, \boldsymbol{C}_3) = \frac{\partial V}{\partial \boldsymbol{p}_i}\nabla_{\boldsymbol{p}}\boldsymbol{p}_i + \frac{\partial V}{\partial \boldsymbol{C}_1}\nabla_{\boldsymbol{p}}\boldsymbol{C}_1 + \frac{\partial V}{\partial \boldsymbol{C}_2}\nabla_{\boldsymbol{p}}\boldsymbol{C}_2 + \frac{\partial V}{\partial \boldsymbol{C}_3}\nabla_{\boldsymbol{p}}\boldsymbol{C}_3.$$

We here use the notation that $\nabla_{\boldsymbol{p}}$ is the gradient with respect to $\boldsymbol{p}_1, \ldots \boldsymbol{p}_n$. The gradient $\nabla_{\boldsymbol{p}}\boldsymbol{p}$ is a $3 \times 3n$ matrix with only zeros, except columns $3i-2$

to $3i$ which equals the identity matrix $I_{3\times 3}$. We will first derive $\frac{\partial V}{\partial \boldsymbol{C}_i}$, then this will be used to give an expression for $\frac{\partial V}{\partial \boldsymbol{p}_i}$. Lastly, we calculate $\nabla_{\boldsymbol{p}} \boldsymbol{C}_j$, which is the hardest part.

Define $\boldsymbol{U}_j = \boldsymbol{C}_j - \boldsymbol{p}_i$ for $j = 1, 2, 3$. This will center the $\boldsymbol{p}_i$ vertex of the tetrahedron at the origin. The vectors $\boldsymbol{U}_1$, $\boldsymbol{U}_2$, $\boldsymbol{U}_3$ equal the edges of the tetrahedron, and its volume V^T is given by

$$V^T = \frac{1}{6}\boldsymbol{U}_1 \cdot (\boldsymbol{U}_2 \times \boldsymbol{U}_3).$$

By using the triple product identity

$$\boldsymbol{U}_1 \cdot (\boldsymbol{U}_2 \times \boldsymbol{U}_3) = \boldsymbol{U}_2 \cdot (\boldsymbol{U}_3 \times \boldsymbol{U}_1) = \boldsymbol{U}_3 \cdot (\boldsymbol{U}_1 \times \boldsymbol{U}_2),$$

we calculate the partial derivatives

$$\frac{\partial V^T}{\partial \boldsymbol{C}_1} = \frac{1}{6}\boldsymbol{U}_2 \times \boldsymbol{U}_3, \quad \frac{\partial V^T}{\partial \boldsymbol{C}_2} = \frac{1}{6}\boldsymbol{U}_3 \times \boldsymbol{U}_1, \quad \frac{\partial V^T}{\partial \boldsymbol{C}_3} = \frac{1}{6}\boldsymbol{U}_1 \times \boldsymbol{U}_2.$$

The partial derivative $\frac{\partial V^T(\boldsymbol{p}_i, \boldsymbol{C}_1, \boldsymbol{C}_2, \boldsymbol{C}_3)}{\partial \boldsymbol{p}_i}$ is found by using the chain rule

$$\frac{\partial V^T(\boldsymbol{p}_i, \boldsymbol{C}_1, \boldsymbol{C}_2, \boldsymbol{C}_3)}{\partial \boldsymbol{p}_i} = \frac{\partial V^T(\boldsymbol{U}_1, \boldsymbol{U}_2, \boldsymbol{U}_3)}{\partial \boldsymbol{p}_i} = -\frac{\partial V^T}{\partial \boldsymbol{C}_1} - \frac{\partial V^T}{\partial \boldsymbol{C}_2} - \frac{\partial V^T}{\partial \boldsymbol{C}_3}.$$

Gradient of $\boldsymbol{C}$

To calculate the gradient of the vertex $\boldsymbol{C}$ we need to consider each of the four configuration cases separately. Depending on the configuration, we can express the vertex $\boldsymbol{C}$ as the intersection of three planes. This leads to a system of equations

$$A\boldsymbol{C} = \boldsymbol{B},$$

where the rows of A are the normal vectors of the planes and $\boldsymbol{B}$ the offset of each plane. First, we recall the matrix derivation rules [34]

$$\mathrm{d}(AB) = (\mathrm{d}A)B + A\,\mathrm{d}B, \quad \mathrm{d}(A^{-1}) = -A^{-1}(\mathrm{d}A)A^{-1}.$$

We use these to obtain an expression for the derivative of the vertex

$$\begin{aligned}\mathrm{d}\boldsymbol{C} = \mathrm{d}(A^{-1}\boldsymbol{B}) = (\mathrm{d}A^{-1})\boldsymbol{B} + A^{-1}\,\mathrm{d}\boldsymbol{B} &= -A^{-1}(\mathrm{d}A)A^{-1}\boldsymbol{B} + A^{-1}\,\mathrm{d}\boldsymbol{B} \\ &= A^{-1}(\mathrm{d}\boldsymbol{B} - (\mathrm{d}A)\boldsymbol{C}).\end{aligned}$$

Given a configuration we can compute the derivatives $\mathrm{d}\boldsymbol{B}$ and $\mathrm{d}A$, and plug this back into the equation. We will first state the results and then give the derivation. Columns of $\nabla_{\boldsymbol{p}}\boldsymbol{C}$ that are zero are understood implicitly and left out for readability. Please see the derivation for explanation.

Configuration A:

$$\nabla_{\boldsymbol{p}}\boldsymbol{C} = 0$$

Configuration B:

$$\nabla_{\boldsymbol{p}}\boldsymbol{C} = \begin{bmatrix} (\boldsymbol{p}_j - \boldsymbol{p}_i)^\top \\ \boldsymbol{N}_1^\top \\ \boldsymbol{N}_2^\top \end{bmatrix}^{-1} \begin{bmatrix} (\boldsymbol{C} - \boldsymbol{p}_i)^\top & (\boldsymbol{p}_j - \boldsymbol{C})^\top \\ \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}$$

Configuration C:

$$\nabla_{\boldsymbol{p}}\boldsymbol{C} = \begin{bmatrix} (\boldsymbol{p}_j - \boldsymbol{p}_i)^\top \\ (\boldsymbol{p}_k - \boldsymbol{p}_i)^\top \\ \boldsymbol{N}_2^\top \end{bmatrix}^{-1} \begin{bmatrix} (\boldsymbol{C} - \boldsymbol{p}_i)^\top & (\boldsymbol{p}_j - \boldsymbol{C})^\top & \mathbf{0} \\ (\boldsymbol{C} - \boldsymbol{p}_i)^\top & \mathbf{0} & (\boldsymbol{p}_k - \boldsymbol{C})^\top \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}$$

Configuration D:

$$\nabla_{\boldsymbol{p}}\boldsymbol{C} = \begin{bmatrix} (\boldsymbol{p}_j - \boldsymbol{p}_i)^\top \\ (\boldsymbol{p}_k - \boldsymbol{p}_i)^\top \\ (\boldsymbol{p}_l - \boldsymbol{p}_i)^\top \end{bmatrix}^{-1} \begin{bmatrix} (\boldsymbol{C} - \boldsymbol{p}_i)^\top & (\boldsymbol{p}_j - \boldsymbol{C})^\top & \mathbf{0} & \mathbf{0} \\ (\boldsymbol{C} - \boldsymbol{p}_i)^\top & \mathbf{0} & (\boldsymbol{p}_k - \boldsymbol{C})^\top & \mathbf{0} \\ (\boldsymbol{C} - \boldsymbol{p}_i)^\top & \mathbf{0} & \mathbf{0} & (\boldsymbol{p}_l.\boldsymbol{C})^\top \end{bmatrix}$$

Configuration A

In this configuration, $\boldsymbol{C}$ is a vertex of the fault. The fault is independent of the set of sites, i.e., the gradient is zero.

Configuration B

In this configuration, $\boldsymbol{C}$ is the intersection of one bisector $[\boldsymbol{p}_i, \boldsymbol{p}_j]$ and two planes $(\boldsymbol{N}_1, b_1)$, $(\boldsymbol{N}_2, b_2)$. This gives us the matrices

$$A = \begin{bmatrix} (\boldsymbol{p}_j - \boldsymbol{p}_i)^\top \\ \boldsymbol{N}_1^\top \\ \boldsymbol{N}_2^\top \end{bmatrix}, \quad \boldsymbol{B} = \begin{bmatrix} \frac{1}{2}(\boldsymbol{p}_j^\top \boldsymbol{p}_j - \boldsymbol{p}_i^\top \boldsymbol{p}_i) \\ -b_1 \\ -b_2 \end{bmatrix}. \tag{A.1}$$

The gradient of B is straight forward with only six columns different from zero. Columns $3i-2,\ 3i-1,\ 3i$ come from the derivative with respect to $\boldsymbol{p}_i$ and equal $[-\boldsymbol{p}_i, \mathbf{0}, \mathbf{0}]^\top$. Columns $3j-2,\ 3j-1,\ 3j$ come from the

derivative with respect to $\boldsymbol{p}_j$ and equal $[\boldsymbol{p}_j,\ \mathbf{0},\ \mathbf{0}]^\top$. The gradient of A is the derivative of a 3×3 matrix with respect to n vectors of size 3×1 . This gives us the $3 \times 3n \times 3$ tensor

$$\nabla_{\boldsymbol{p}} A = \begin{bmatrix} \frac{\partial(\boldsymbol{p}_j - \boldsymbol{p}_i)^\top}{\partial \boldsymbol{p}_1} & \dots & \frac{\partial(\boldsymbol{p}_j - \boldsymbol{p}_i)^\top}{\partial \boldsymbol{p}_n} \\ 0 & \dots & 0 \\ 0 & \dots & 0 \end{bmatrix}.$$

The derivative $\frac{\partial(\boldsymbol{p}_j - \boldsymbol{p}_i)^\top}{\partial \boldsymbol{p}_k}$ is an $1 \times 3 \times 3$ tensor which equal

$$\frac{\partial(\boldsymbol{p}_j - \boldsymbol{p}_i)^\top}{\partial \boldsymbol{p}_k} = \begin{cases} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, & k = j \\ \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}, & k = i \\ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, & k \neq j, i. \end{cases}$$

Multiplying $\nabla_{\boldsymbol{p}} A$ by the vector $\boldsymbol{C}$ gives us a $3 \times 3n$ matrix. The only columns different from zero are columns $3i-2,\ 3i-1,\ 3i$ which equal $[-\boldsymbol{C}, \mathbf{0}, \mathbf{0}]^\top$, and columns $3j-2,\ 3j-1,\ 3j$ which equal $[\boldsymbol{C}, \mathbf{0}, \mathbf{0}]^\top$. We will mention a small misprint in the original paper by Lévy and Liu [31]. They mistakenly add an extra element to $(\nabla_{\boldsymbol{p}} A)\boldsymbol{C}$ and $\nabla_{\boldsymbol{p}} \boldsymbol{B}$ and write that columns $3i-2$ to $3i$ of the two matrices are $[-\boldsymbol{C}, -\boldsymbol{C}, \mathbf{0}]^\top$ and $[-\boldsymbol{p}_i, -\boldsymbol{p}_i, \mathbf{0}]^\top$ respectively.

Configuration C

In configuration C, the vertex is given by the intersection of two bisectors $[\boldsymbol{p}_i, \boldsymbol{p}_j]$, $[\boldsymbol{p}_i, \boldsymbol{p}_k]$ and one plane $(\boldsymbol{N}_1, b_1)$. The A and $\boldsymbol{B}$ matrices are then

$$A = \begin{bmatrix} (\boldsymbol{p}_j - \boldsymbol{p}_i)^\top \\ (\boldsymbol{p}_k - \boldsymbol{p}_i)^\top \\ \boldsymbol{N}_2^\top \end{bmatrix}, \quad \boldsymbol{B} = \begin{bmatrix} \frac{1}{2}(\boldsymbol{p}_j^\top \boldsymbol{p}_j - \boldsymbol{p}_i^\top \boldsymbol{p}_i) \\ \frac{1}{2}(\boldsymbol{p}_k^\top \boldsymbol{p}_k - \boldsymbol{p}_i^\top \boldsymbol{p}_i) \\ -b_1 \end{bmatrix}. \tag{A.2}$$

The calculation of the gradient is equivalent to above, but there are another three columns different from zero. Columns $3k-2,\ 3k-1,\ 3k$ equal

$[0, \boldsymbol{p}_k, 0]^\top$ for $\nabla_{\boldsymbol{p}} \boldsymbol{B}$ and $[0, \boldsymbol{C}, 0]^\top$ for $(\nabla_{\boldsymbol{p}} A)\boldsymbol{C}$.

Configuration D

In configuration D, the vertex is given by the intersection of the three bisectors $[\boldsymbol{p}_i, \boldsymbol{p}_j]$, $[\boldsymbol{p}_i, \boldsymbol{p}_k]$, $[\boldsymbol{p}_i, \boldsymbol{p}_l]$. This gives the matrices

$$A = \begin{bmatrix} (\boldsymbol{p}_j - \boldsymbol{p}_i)^\top \\ (\boldsymbol{p}_k - \boldsymbol{p}_i)^\top \\ (\boldsymbol{p}_l - \boldsymbol{p}_i)^\top \end{bmatrix}, \quad \boldsymbol{B} = \begin{bmatrix} \frac{1}{2}(\boldsymbol{p}_j^\top \boldsymbol{p}_j - \boldsymbol{p}_i^\top \boldsymbol{p}_i) \\ \frac{1}{2}(\boldsymbol{p}_k^\top \boldsymbol{p}_k - \boldsymbol{p}_i^\top \boldsymbol{p}_i) \\ \frac{1}{2}(\boldsymbol{p}_l^\top \boldsymbol{p}_l - \boldsymbol{p}_i^\top \boldsymbol{p}_i) \end{bmatrix}. \tag{A.3}$$

The calculation of the gradient is equivalent to above, but with three more columns different from zero. Column $3l-2,\ 3l-1,\ 3l$ equal $[0, 0, \boldsymbol{p}_l]^\top$ for $\nabla_{\boldsymbol{p}} \boldsymbol{B}$ and $[0, 0, \boldsymbol{C}]^\top$ for matrix $(\nabla_{\boldsymbol{p}} A)C$.

Bibliography

[1] I. Aavatsmark. Interpretation of a two-point flux stencil for skew parallelogram grids. *Computational Geosciences*, 11(3):199–206, 2007. ISSN 1573-1499. URL http://dx.doi.org/10.1007/s10596-007-9042-1.

[2] K. Bao, K.-A. Lie, O. Møyner, and M. Liu. Fully Implicit Simulation of Polymer Flooding with MRST. European Conference on the Mathematics of Oil Recovery, September 2016.

[3] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The Quickhull Algorithm for Convex Hulls. *ACM TRANSACTIONS ON MATHEMATICAL SOFTWARE*, 22(4):469–483, 1996.

[4] R. L. Berge. Unstructured Grids Adapting to Geological Features. Specialization project, NTNU, January 2016.

[5] R. L. Berge. Unstructured PEBI-grids for Reservoirs (UPR). Git Repository, June 2016. URL https://github.com/92runarlb/UPR.git.

[6] L. Branets, S. S. Ghai, S. L. Lyons, and X.-H. Wu. Efficient and Accurate Reservoir Modeling Using Adaptive Gridding with Global Scale Up. In *Proceedings of the SPE Reservoir Simulation Symposium*, The Woodlands, Texas, Jan. 2009. doi: 10.2118/118946-MS.

[7] M. Brewer, D. Camilleri, S. Ward, and T. Wong. Generation of hybrid grids for simulation of complex, unstructured reservoirs by a simulator

with mpfa. *Society of Petroleum Engineers*, 2015. URL http://dx.doi.org/10.2118/173191-MS.

[8] P. Cignoni, C. Montani, and R. Scopigno. DeWall: A fast divide and conquer Delaunay triangulation algorithm in Ed. *Computer-Aided Design*, 30(5):333–341, 1998. ISSN 0010-4485. doi: 10.1016/S0010-4485(97)00082-1. URL http://www.sciencedirect.com/science/article/pii/S0010448597000821.

[9] G. Courrioux, S. Nullans, A. Guillen, J. D. Boissonnat, P. Repusseau, X. Renaud, and M. Thibaut. 3D volumetric modelling of Cadomian terranes Northern Brittany, France): an automatic method using Voronoi diagrams. *Tectonophysics*, 331:181–196, 2001.

[10] W. C. Davidon. Variable Metric Method for Minimization. *SIAM Journal on Optimization*, 1(1):1–17, 1991. URL http://dx.doi.org/10.1137/0801001.

[11] B. Delaunay. Sur la sphére vide. *Bull. Acad. Sci. USSR, Classe Sci. Mat. Nat.*, 7:793–800, 1934.

[12] X. Y. Ding and L. S. K. Fung. An Unstructured Gridding Method for Simulating Faulted Reservoirs Populated with Complex Wells. In *Proceedings of the SPE Reservoir Simulation Symposium*, Houston, Texas, USA, February 2015. doi: 10.2118/173243-MS.

[13] Y. Ding and P. Lemonnier. Use of Corner Point Geometry in Reservoir Simulation. International Meeting on Petroleum Engineering, November 1995. ISSN 978-1-55563-438-4. Beijing, China.

[14] Q. Du, V. Faber, and M. Gunzburger. Centroidal Voronoi Tessellations: Applications and Algorithms. *SIAM Review*, 41(4):637–676, 1999. URL http://dx.doi.org/10.1137/S0036144599352836.

[15] J.-F. Dufourd and Y. Bertot. *Formal Study of Plane Delaunay Triangulation*, volume 6172 of *Lecture Notes in Computer Science*, pages 211–226. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-14051-8. URL http://dx.doi.org/10.1007/978-3-642-14052-5_16.

[16] L.-K. Fung, A. Hiebert, and L. Nghiem. Reservoir Simulation With a Control-Volume Finite-Element Method. *Society of Petroleum Engineers*, 7(3):349–356, Aug. 1992. doi: 10.2118/21224-PA.

[17] L. S. K. Fung, X. Y. Ding, and A. H. Dogru. Unconstrained Voronoi Grids for Densely Spaced Complex Wells in Full-Field Reservoir Simulation. *Society of Petroleum Engineers*, 19(5):803–815, Oct. 2014. doi: 10.2118/163648-PA.

[18] GISTEMP. Giss surface temperature analysis (gistemp), 06 2016. URL http://data.giss.nasa.gov/gistemp/.

[19] E. J. Gringarten, G. B. Arpat, M. A. Haouesse, A. Dutranois, L. Deny, S. Jayr, A.-L. Tertois, J.-L. Mallet, A. Bernal, and L. X. Nghiem. New Grids for Robust Reservoir Modeling. SPE Annual Technical Conference and Exhibition, 21-24 September, Denver, Colorado, USA, 2008. ISSN 978-1-55563-147-5. URL http://dx.doi.org/10.2118/116649-MS.

[20] E. J. Gringarten, M. A. Haouesse, G. B. Arpat, and L. X. Nghiem. Advantages of Using Vertical Stair Step Faults in Reservoir Grids for Flow Simulation. SPE Reservoir Simulation Symposium, 2-4 February, The Woodlands, Texas, 2009. ISSN 978-1-55563-209-0. URL http://dx.doi.org/10.2118/119188-MS.

[21] D. Guerillot and P. Swaby. An Interactive 3D Mesh Builder for Fluid Flow Reservoir Simulation. *Society of Petroleum Engineers*, 5(6):5–10, Nov. 1993. doi: 10.2118/26227-PA.

[22] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(1-6): 381–413, 1992. ISSN 0178-4617. URL http://dx.doi.org/10.1007/BF01758770.

[23] J. Hansen, R. Ruedy, and K. Lo. Global surface temperature change. *Reviews of Geophysics*, 2010. doi: 10.1029/2010RG000345.

[24] Z. E. Heinemann, C. W. Brand, M. Munka, and Y. M. Chen. Modeling Reservoir Geometry With Irregular Grids. *Society of Petroleum Engineers*, 6(2):225–232, May 1991. doi: 10.2118/18412-PA.

[25] R. Holm, R. Kaufmann, B.-O. Heimsund, E. Øian, and M. S. Espedal. Meshing of domains with complex internal geometries. *Numerical Linear Algebra with Applications*, 13(9):717–731, 2006. ISSN 1099-1506. doi: 10.1002/nla.505. URL http://dx.doi.org/10.1002/nla.505.

[26] M. Iri, K. Murota, and T. Ohya. *System Modelling and Optimization: Proceedings of the 11th IFIP Conference Copenhagen, Denmark, July*

25–29, 1983, chapter A fast Voronoi-diagram algorithm with applications to geographical optimization problems, pages 273–288. Springer Berlin Heidelberg, Berlin, Heidelberg, 1984. ISBN 978-3-540-38828-9. URL http://dx.doi.org/10.1007/BFb0008901.

[27] C. Jamin, S. Pion, and M. Teillaud. 3D Triangulations. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.7 edition, 2015. URL http://doc.cgal.org/4.7/Manual/packages.html#PkgTriangulation3Summary.

[28] M. Karimi-Fard, L. J. Durlofsky, and K. Aziz. An Efficient Discrete-Fracture Model Applicable for General-Purpose Reservoir Simulators. *Society of Petroleum Engineers*, 9(02):227 – 236, June 2004. URL http://dx.doi.org/10.2118/88812-PA.

[29] M. Kaviany. Laminar flow through a porous channel bounded by isothermal parallel plates. *International Journal of Heat and Mass Transfer*, 28(4):851 – 858, 1985. ISSN 0017-9310. URL http://dx.doi.org/10.1016/0017-9310(85)90234-0.

[30] Øystein. S. Klemetsdal. The Virtual Element Method as a Common Framework for Finite Element and Finite Difference Methods. Master's thesis, Norwegian University of Science and Technology, June 1016.

[31] B. Lévy and Y. Liu. Lp Centroidal Voronoi Tessellation and Its Applications. *ACM Trans. Graph.*, 29(4):119:1–119:11, July 2010. ISSN 0730-0301. URL http://doi.acm.org/10.1145/1778765.1778856.

[32] K.-A. Lie. *An Introduction to Reservoir Simulation Using MATLAB: User guide for the Matlab Reservoir Simulation Toolbox (MRST)*. SINTEF ICT, Dec 2015. URL http://www.sintef.no/MRST.

[33] Y. Liu, W. Wang, B. Lévy, F. Sun, D.-M. Yan, L. Lu, and C. Yang. On Centroidal Voronoi Tessellation&Mdash;Energy Smoothness and Fast Computation. *ACM Trans. Graph.*, 28(4):101:1–101:17, Sept. 2009. ISSN 0730-0301. URL http://doi.acm.org/10.1145/1559755.1559758.

[34] J. Magnus and H. Neudecker. *Matrix Differential Calculus with Applications in Statistics and Econometrics*, chapter 8, page 168 and 171. Wiley Series in Probability and Statistics - Applied Probability and Statistics Section. John Wiley & Sons, 3rd edition, 1988.

[35] B. Mallison, C. Sword, T. Viard, W. Milliken, and A. Cheng. Unstructured Cut-Cell Grids for Modeling Complex Reservoirs. *SPE-163642-PA*, 2014. URL http://dx.doi.org/10.2118/163642-PA.

[36] R. Merland, C. Guillaume, L. Bruno, and C.-D. Pauline. Building Centroidal Voronoi Tessellations For Flow Simulation In Reservoirs Using Flow Information. In *Proceedings of the SPE Reservoir Simulation Symposium*, The Woodlands, Texas, USA, February 2011. doi: 10.2118/141018-MS.

[37] R. Merland, B. Lévy, and G. Caumon. Building PEBI Grids Conforming To 3D Geological Features using Centroidal Voronoi Tessellation. In *IAMG*, page 12, Salzburg, 2011.

[38] R. Merland, G. Caumon, B. Lévy, and P. Collon-Drouaillet. Voronoi grids conforming to 3D structural features. *Computational Geosciences*, 18(3-4):373–383, 2014. ISSN 1420-0597. URL http://dx.doi.org/10.1007/s10596-014-9408-0.

[39] H. Mustapha. G23fm: a tool for meshing complex geological media. *Computational Geosciences*, 15(3):385–397, 2011. ISSN 1573-1499. doi: 10.1007/s10596-010-9210-6. URL http://dx.doi.org/10.1007/s10596-010-9210-6.

[40] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer-Verlag New York, 2 edition, 2006. doi: 10.1007/978-0-387-40065-5.

[41] C. Palagi and K. Aziz. Use of Voronoi Grid in Reservoir Simulation. *Society of Petroleum Engineers*, 2(2):69–77, Apr. 1994. doi: 10.2118/22889-PA.

[42] G. Parigi and M. Piastra. Gradient of the Objective Function for an Anisotropic Centroidal Voronoi Tessellation (CVT) - A revised, detailed derivation. *CoRR*, abs/1408.5622, 2014. URL http://arxiv.org/abs/1408.5622.

[43] P.-O. Persson and G. Strang. A Simple Mesh Generator in MATLAB. *SIAM Review*, 46(2):329–345, 2004. URL http://dx.doi.org/10.1137/S0036144503429121.

[44] D. Ponting. Corner Point Geometry in Reservoir Simulation. 1st European Conf. On Math. of Oil Rec., July 1989. Axford, England.

[45] T. Sandve, I. Berre, and J. Nordbotten. An efficient multi-point flux approximation method for Discrete Fracture–Matrix simulations. *Journal of Computational Physics*, 231(9):3784–3800, 2012. ISSN 0021-9991. doi: 10.1016/j.jcp.2012.01.023.

[46] T. H. Sandve. *Multiscale Simulation of Flow and Heat Transport in Fractured Geothermal Reservoirs*. PhD thesis, University of Bergen, 2012.

[47] J. R. Shewchuk, S.-W. Cheng, and T. K. Dey. *Delaunay Mesh Generation*. Computer and Information Science. Chapman and Hall/CRC, 2012. doi: 10.1201/b12987-3.

[48] J. Sun and D. Schechter. Optimization-Based Unstructured Meshing Algorithms for Simulation of Hydraulically and Naturally Fractured Reservoirs With Variable Distribution of Fracture Aperture, Spacing, Length, and Strike. *SPE-170703-PA*, 18(04):463 – 480, November 2015. URL http://dx.doi.org/10.2118/170703-PA.

[49] I. E. Sutherland and G. W. Hodgman. Reentrant Polygon Clipping. *Commun. ACM*, 17(1):32–42, Jan. 1974. ISSN 0001-0782. URL http://doi.acm.org/10.1145/360767.360802.

[50] S. M. Toor, M. G. Edwards, A. H. Dogru, and T. M. Shaalan. Boundary Aligned Grid Generation in Three Dimensions and CVD-MPFA Discretization. In *Proceedings of the SPE Reservoir Simulation Symposium*, Houston, Texas, USA, feb 2015. doi: 10.2118/173313-MS.

[51] S. Verma and K. Aziz. A Control Volume Scheme for Flexible Grids in Reservoir Simulation. SPE Reservoir Simulation Symposium, June 1997. ISSN 978-1-55563-403-2.

[52] G. Voronoi. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Deuxième mémoire. Recherches sur les parallélloèdres primitifs. *Journal für die reine und angewandte Mathematik*, 134:198–287, 1908. URL http://eudml.org/doc/149291.

[53] P. Wolfe. Convergence conditions for ascent methods. *SIAM Review*, 11(2):226–235, 1969. URL http://dx.doi.org/10.1137/1011036.

[54] X.-H. Wu and R. Parashkevov. Effect of Grid Deviation on Flow Solutions. *Society of Petroleum Engineers*, 14(01):67–77, March 2009. URL http://dx.doi.org/10.2118/92868-PA. SPE-92868-PA.

[55] D.-M. Yan, B. Lévy, Y. Liu, F. Sun, and W. Wang. Isotropic Remeshing with Fast and Exact Computation of Restricted Voronoi Diagram. In *Proceedings of the Symposium on Geometry Processing*, SGP '09, pages 1445–1454, Aire-la-Ville, Switzerland, Switzerland, 2009. Eurographics Association. URL http://dl.acm.org/citation.cfm?id=1735603.1735629.

[56] D.-M. Yan, W. Wang, B. Lévy, and Y. Liu. *Advances in Geometric Modeling and Processing: 6th International Conference, GMP 2010, Castro Urdiales, Spain, June 16-18, 2010. Proceedings*, chapter Efficient Computation of 3D Clipped Voronoi Diagram, pages 269–282. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-13411-1. URL http://dx.doi.org/10.1007/978-3-642-13411-1_18.