1

Unstructured PEBI Grids Conforming to Lower-Dimensional Objects

RUNAR L. BERGE, ØYSTEIN S. KLEMETSDAL, AND KNUT-ANDREAS LIE

Abstract

The `upr` module in the MATLAB Reservoir Simulation Toolbox (MRST) can construct unstructured Voronoi grids that conform to polygonal boundaries and geometric constraints in arbitrary dimensions prescribed inside the reservoir volume. The resulting volumetric tessellations are usually realized as locally orthogonal, perpendicular bisector (PEBI) grids, in which cell faces can be aligned to accurately preserve objects of codimension one (curves in 2D and surfaces in 3D) and/or cell centroids can be set to follow curves in 2D or 3D. This enables you to accurately model faults, let grid cells follow horizontal and multilateral well paths, or create lower-dimensional or volumetric representations of fracture networks. The module offers methods for improving grid quality, like configurable policies for treating intersecting geometric objects and handling conflicts among constraints, methods for locating and removing conflicting generating points, as well as force-based and energy-minimization approaches for optimizing the grid cells. You can use `upr` to create a consistent hierarchy of grids that represent the reservoir volume, the constraining geometric objects (surfaces and curves), as well as their intersections. The hierarchy is built such that the cell faces of a given (sub)grid conform to the cells of all bounding subgrids of one dimension lower.

1.1 Introduction

The basic geometric description of a reservoir or aquifer will typically consist of multiple surfaces representing the top and base of the reservoir and the main bounding faults, as well as surfaces that represent internal structures such as depositional environments, lithological contrasts, minor faults, and fractures that restrict or guide the fluid flow. It is important to respect these geological boundaries in the computational grid to achieve accurate simulations, and the number of surfaces

https://doi.org/10.1017/9781009019781.005 Published online by Cambridge University Press

to be respected tends to increase (steeply) as more details are added to the reservoir characterization. Until recently, identifying and tracing geological surfaces in processed seismic data has mainly been a manual process. If emerging automated interpretation approaches based on machine learning become more widespread, one should expect a significant growth in both the number and the complexity of the surfaces users will desire to incorporate in simulation models.

The MATLAB Reservoir Simulation Toolbox's (MRST) grid structure is very flexible and allows for completely unstructured topologies and general polyhedral cell geometries. (You can find a detailed discussion in chapter 3 of the MRST textbook [11].) The core module of MRST includes several functions for creating a wide variety of grids, from simple rectilinear meshes, via corner-point grids and unstructured simplex grids, to hybrid and multiblock grids. The grid structure puts few restrictions on the types of grids you can represent, but constructing very complex grids that adapt to outer and inner constraints on the reservoir geometry can be a tedious and complicated process using the basic functionality from the core module.

The main motivation for the `upr` module was to develop a family of relatively simple yet flexible methods for generating grids that adapt automatically to internal geometric structures that delineate structural, stratigraphic, sedimentological, or diagenetic heterogeneities. From a geological perspective, these structures are of (very) different natures. However, for grid-generation purposes, we divide the corresponding constraints into two groups: constraints for which the cell centroids should align with the geometric object and constraints for which the faces of the cells should align with the geometric object. Alignment of cell centroids is typically desirable to trace out the paths of deviated or horizontal wells. Wells are usually modeled by analytical or semi-analytical inflow performance relationships (see [11, subsection 11.7]), which in their basic form assume that the well is perforated at the center of the cell. One can also use the same functionality to trace out fractures that should be represented as volumetric objects. Boundary alignment is desirable to trace out faults and various forms of internal layering and zonation within the reservoir, as well as fractures that are to be represented as lower-dimensional objects in discrete fracture–matrix (DFM) models.

In the `upr` module, we use so-called clipped Voronoi diagrams [19] to create unstructured polyhedral and polygonal grids that allow two different types of conformity requirements: cell centroids tracing prescribed lines in 2D or 3D or cell faces tracing surfaces in 3D and lines in 2D. In the literature, the names "Voronoi mesh," "Voronoi diagram," and "perpendicular bisector (PEBI) grid" are all used to denote the same types of grids. The `upr` module uses PEBI as name convention, because this is most common in the petroleum industry. Table 1.1 lists the entry-level functions for generating grids using `upr`. The list is by no means exhaustive

Table 1.1 *Short overview of the entry-level functions in the* `upr` *module you can use to generate different types of PEBI grids. In the description, the word "sites" refers to generating points that control the Voronoi diagrams or the underlying Delaunay tessellations.*

Function name	Description
`pebiGrid2D`	Generate a 2D PEBI grid that conforms to internal constraints
`compositePebiGrid2D`	Similar to pebiGrid2D but with a structured background grid
`clippedPebi2D`	From a given set of sites, create a 2D clipped PEBI grid
`CPG2D`	From a given set of sites, optimize the site positions
`compositePebiGrid3D`	Generate a 3D PEBI grid that conforms to internal constraints
`mirroredPebi3D`	From a given set of sites, create a 3D PEBI grid
`CPG3D`	From a given set of sites, optimize the site positions

but offers an overview of the most important functions and an introductory point to `upr`. See also the overview in Figure 1.26 at the end of this chapter. The main purpose of this chapter is to give a basic introduction to these functions, briefly explain some of the underlying theory, and give several code-centric examples of how the functionality in the module can be used to generate complex grids that conform to geological objects and well paths. For a more comprehensive overview of alternative methods and previous research, the reader can consult [2, 6, 8, 13–15, 18].

We emphasize that the `upr` module is a research tool for constrained gridding and not a robust industry-grade geomodeling tool. You can use it to generate reservoir models with grid topologies and cell shapes that are representative of what one may encounter in complex geological descriptions of real-life problems. However, functions in the module use geometric algorithms and tolerances that primarily have been tested and adjusted for grids of unit size and close to unit aspect ratios. To create grids with more realistic dimensions and aspect ratios, we generally advise you to scale and translate inner and outer constraints to the unit domain in the first quadrant (or moderate multiples thereof) when creating the grid and then rescale and translate the grid back to the desired size and position afterward.

1.2 Basic Introduction to PEBI Grids

PEBI grids are closely related to Delaunay triangulations; in fact, the two are the dual of each other, as we will see. Both Delaunay and PEBI grids are uniquely defined (up to any degeneracy) by a set of generating points, or *sites* for short.

For Delaunay, the sites, denoted by $\{\vec{s}_i\}_{i=1,\ldots,n} = \mathcal{S}$, correspond to the vertices of the grid, whereas the sites are associated with cells for the PEBI grid.

In this section, we first introduce the Delaunay triangulation, describe the PEBI grid, and review some of its important properties. We also explain how you can construct simple unstructured grids using MRST and features from the `upr` module. This section hence gives the basic tools needed to construct conforming PEBI grids. You can find the complete source code for all examples in this section in the script `uprBookSection2.m` in the `examples/book-ii` directory of the `upr` module.

1.2.1 Delaunay Triangulation

There exist many different methods for creating the Delaunay triangulation given a set of sites; see, e.g., the textbook by Shewchuk et al. [17]. Discussing these methods is outside the scope of this chapter, but to make the discussion as self-contained as possible, we will nonetheless introduce some important properties that are useful when constructing the dual PEBI grid. Let us start by giving a precise definition of the Delaunay triangulation.

Theorem 1.1. *Let $\mathcal{S}$ be a set of points. The Delaunay triangulation, denoted by $\mathcal{T}$, is a tessellation of the convex hull of $\mathcal{S}$ into simplices, such that the interior of the circumsphere of each simplex contains no sites from $\mathcal{S}$.*

This theorem does not present an obvious way to construct the Delaunay triangulation but gives a straightforward condition you can use to check whether a triangulation is Delaunay: Draw the circumsphere around each element in the triangulation and check that the spheres do not contain any sites from $\mathcal{S}$. The empty circumsphere principle is shown in Figure 1.1 for a Delaunay triangulation of six sites.

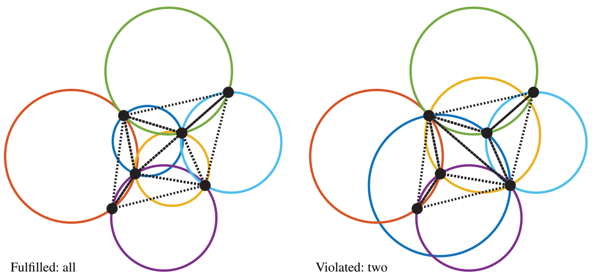


Figure 1.1 The empty circumsphere principle for Delaunay triangulations in 2D. The circumcircles of each of the six triangles shown in different colors should not contain any vertices in their interior. (Not fulfilled for the dark blue/yellow circles in the right plot.)

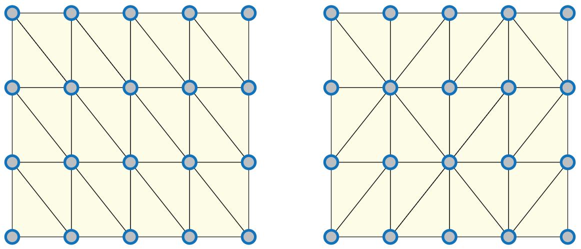
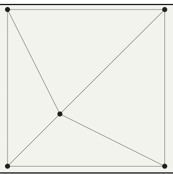


Figure 1.2 Two valid Delaunay triangulations of a Cartesian point set created by `delaunay` (left) and `delaunayn` (right). The four vertices on each quadrangle can be placed on the same circle, and both ways of splitting the quadrangle are therefore equivalent.

There is a lot of triangulation software available, and instead of implementing its own Delaunay triangulation, MRST offers an interface to transform a general triangulation into a grid object `G`. The following code uses the built-in MATLAB function `delaunay` to create a Delaunay grid from five generation points (or sites):

```
% Define sites
sites = [0, 0; 1, 0; 1, 1; 0, 1; 1/3, 1/3];
% Create Delaunay triangulation
t = delaunay(sites);
% Convert to MRST grid
G = triangleGrid(sites, t);
```



The Delaunay triangulation for a set of sites is unique up to any degenerate points. We say that $d+2$ sites from $\mathcal{S} \subset \mathbb{R}^d$ are degenerate if there exists a sphere that intersects all $d+2$ sites. In 2D, this happens, e.g., when the four sites form a quadrangle (Figure 1.2), because no matter which diagonal we pick as an edge in the triangulation, we have a valid Delaunay triangulation. Most triangulation algorithms are able to handle these degenerate cases, some algorithms will pick a diagonal at random, whereas others will always pick the same. The `delaunay` function in MATLAB produces a structured triangulation, whereas `delaunayn` and versions of `delaunay` prior to R2009b use Qhull [1] and produce the unstructured variation.

1.2.2 PEBI Grids

We denote the PEBI grid of a set of sites by $\mathcal{P}$. These sites uniquely define the PEBI grid, and for each site $\vec{s}_i \in \{\vec{s}_j\}_{j=1,\dots,n} = \mathcal{S}$, we associate a cell v_{s_i} that is defined by

$$v_{s_i} = \{\vec{x} : \vec{x} \in \mathbb{R}^d, \|\vec{x} - \vec{s}_i\| < \|\vec{x} - \vec{s}_k\|, \forall k \in \{1,\dots,n\} \setminus \{i\}\}. \tag{1.1}$$

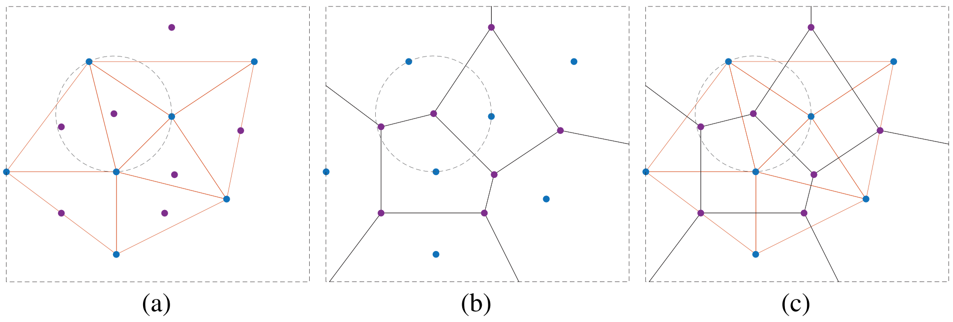


Figure 1.3 The duality of a Delaunay triangulation and a PEBI grid in 2D. Delaunay vertices correspond to PEBI cells (blue points). Delaunay edges are perpendicular to the corresponding PEBI edges. The circumcenter of a Delaunay triangle corresponds to a PEBI vertex (purple points).

An intuitive explanation of this is that a cell v_{s_i} is defined by all points in $\mathbb{R}^d$ that are closer to $\vec{s}_i$ than any other sites; see Figure 1.3 for an example of a PEBI grid.

A face between the two cells v_{s_i} and v_{s_j}, denoted by v_{s_i,s_j}, can now be defined as the intersection between the closure of the two cells $v_{s_i,s_j} = \bar{v}_{s_i} \cap \bar{v}_{s_j}$. For a PEBI grid, the face can alternatively be expressed as

$$v_{s_i,s_j} = \{\vec{x} : \vec{x} \in \mathbb{R}^d,\ \|\vec{x} - \vec{s}_i\| = \|\vec{x} - \vec{s}_j\| < \|\vec{x} - \vec{s}_k\|,\ \forall k \in \{1, \dots, n\} \setminus \{i, j\}\}.$$

This means that the face v_{s_i,s_j} between the two cells v_{s_i}, v_{s_j} consists of all points that are closer to the two sites $\vec{s}_i$ and $\vec{s}_j$ than any other site.

In general, we define a k-face of the grid as the k-dimensional intersection between cells. In 3D, a 2-face is an interface between two neighboring cells, a 1-face is an edge that defines the intersection between at least two cell interfaces, and 0-faces are nodes/vertices in the grid that represent intersections among cell edges. In addition, each cell is said to be a 3-face. For a PEBI grid in $\mathbb{R}^d$, the $(d - i + 1)$-face is defined by a set of sites $\{\vec{s}_1, \dots, \vec{s}_i\} \subset \mathcal{S}$ as

$$v_{s_1,\dots,s_i} = \{\vec{x} : \vec{x} \in \mathbb{R}^d,\ \|\vec{x} - \vec{s}_1\| = \dots = \|\vec{x} - \vec{s}_i\| < \|\vec{x} - \vec{s}_k\|, \\ k = i + 1, \dots, n\}. \quad (1.2)$$

Thus, the sites $\vec{s}_1, \dots, \vec{s}_i$ (with $i = 4$ in 3D and $i = 3$ in 2D) define a vertex $v_{s_1,\dots,s_i}$ if and only if the interior of the ball that intersects the sites $\vec{s}_1, \dots, \vec{s}_i$ does not contain any other sites from $\mathcal{S}$. The circle intersecting three sites in Figure 1.3 demonstrates this.

The duality between the Delaunay triangulation and the PEBI grid gives us some important properties and relations we can exploit to construct conforming

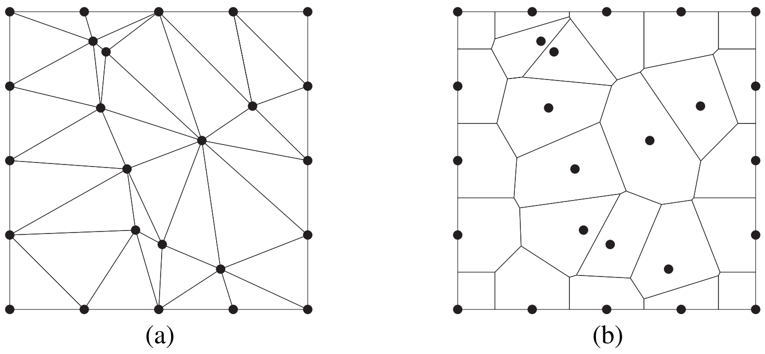


Figure 1.4 A set of sites that are perturbed randomly: (a) The Delaunay triangulation of the sites; (b) The PEBI grid of the sites.

PEBI grids. It will also give us a better understanding of why the conforming algorithms presented later in this chapter work. The duality of the two grids is stated as follows.

Theorem 1.2. *Let S be a generic point set (maximum $d+1$ points can be intersected by one sphere) in $\mathbb{R}^d$. Let $\mathcal{P}$ and $\mathcal{T}$ be the associated PEBI grid and Delaunay triangulation, respectively. Define a subset $P = \{\vec{s}_1, \ldots, \vec{s}_j\} \subset S$. Then, the convex hull of P is a k-face of $\mathcal{T}$ if and only if $v_{s_1,\ldots,s_j}$ is a $(d-k)$-face of $\mathcal{P}$.*

Most important, this means that each node in $\mathcal{T}$ is associated with a cell in $\mathcal{P}$ and each cell in $\mathcal{T}$ is associated with a node in $\mathcal{V}$. In fact, the center of the circumsphere around a cell from the Delaunay triangulation will be a vertex in the PEBI grid. Further, if there is an edge in $\mathcal{T}$ between two sites, then the two sites share a face in $\mathcal{P}$; see Figure 1.3 for an illustration of the duality.

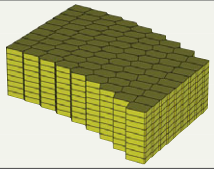
The function `pebi` creates PEBI grids directly from a Delaunay triangulation in 2D by connecting the perpendicular bisectors of all edges with the circumcenters of each cell. This method is relatively fast but is only implemented in 2D. Also be aware that it will fail along the domain boundary if the circumcenter of any triangle lies outside the convex hull of the sites.

As an example, we create and perturb a set of sites and generate the Delaunay triangulation and PEBI grid shown in Figure 1.4:

```
n = 5;
[X,Y] = meshgrid(linspace(0,1,n));
isIn  = false(n,n);  isIn(2:end-1,2:end-1) = true;
sites = [X(:),Y(:)];
sites(isIn(:),:) = sites(isIn(:), :) + 0.1*randn((n-2)^2, 2);
Gt    = triangleGrid(sites);
Gp    = pebi(Gt);
```

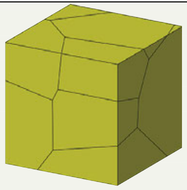
MATLAB also offers the voronoin function, based on Qhull [1], for computing Voronoi vertices and cells from a set of generating points. The upr module has a routine that transforms the resulting set of vertices and cells into a valid MRST grid, illustrated by the following code example for a set of points in 3D:

```
keep     = false(11,11,11);   % Flag used later to remove
keep(2:10,2:10,1:11)=true;    % boundary cells
[X,Y,Z]  = meshgrid(linspace(0,1,11));
sites    = [X(:)+.5*Y(:).^2 Y(:) Z(:)];
[V,C]    = voronoin(sites);
G        = voronoi2mrstGrid3D(V,C(keep));
```



Here, V holds the vertices and C maps from each cell to its vertices. Note that voronoin creates an unbounded Voronoi diagram in which the boundary cells extend to infinity, which is not suitable in a practical grid. Such cells are disregarded by voronoi2mrstGrid3D, but here we clip away the outer cell layer explicitly using the Boolean array keep. The upr module also includes another function mirroredPebi3D that creates the PEBI grid of a convex domain by placing mirror sites outside the boundary. This function is a wrapper around the voronoin function and will clip the PEBI grid by the convex hull of the specified boundary. We discuss clipped 2D PEBI grids further in the next subsection.

```
pts = rand(10, 3);
% Define the unit cube as boundary
bnd = [0 0 1 1 0 0 1 1; ...
       0 1 1 0 0 1 1 0; ...
       0 0 0 0 1 1 1 1]';
% Create clipped Pebi grid
G = mirroredPebi3D(pts, bnd);
```



1.2.3 *Clipping PEBI Grids*

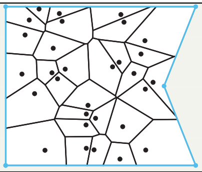
Having PEBI cells that extend to infinity is not desirable when the grid is to be used for numerical approximations of a finite domain. To resolve this, we restrict our domain to a bounded subset $\Omega \subset \mathbb{R}^d$:

$$v_{s_i} = \{\vec{x} : \vec{x} \in \Omega, \|\vec{x} - \vec{s}_i\| < \|\vec{x} - \vec{s}_k\|, \forall k \in \{1, \dots, n\} \setminus \{i\}\}.$$

MRST's pebi function assumes that the domain is bounded by the *convex hull* of the sites. To generate a grid that would fit a more general domain, either we would have to carefully place mirror sites along the outside of the boundary that can later be clipped away, as illustrated for a simple case in the previous subsection, or we could use a fictitious-domain approach, as discussed in section 3.1 of the MRST textbook. Both approaches are somewhat cumbersome to implement, and special care must be taken to ensure that the remaining cells adapt to the correct boundary; we will discuss how to do this for *internal* boundaries in Section 1.4.

A more straightforward clipping approach is possible, in particular if the domain is bounded by polygonal curves or piecewise polynomial surfaces. The `upr` module implements the algorithm presented by [19], which clips each 2D PEBI cell against a polygon boundary. (In 3D, you should use `mirroredPebi3D`.) The boundary is specified by the vertices of the polygon and should be ordered counterclockwise:

```
% Define boundary as a polygon
bnd = [0, 0; 1.2, 0; 1, 0.5; 1.2, 1; 0, 1];
% Set random sites
sites = rand(30, 2);
% Generate Voronoi grid
G = clippedPebi2D(sites, bnd);
```



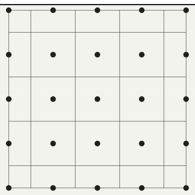
1.3 Three Approaches for Optimizing PEBI Grids

By now, you have probably realized that grid quality depends strongly on the site positions; placing sites somewhat haphazardly tends to give ill-shaped cells. The `upr` module supplies three different methods for generating optimized PEBI grids that can be used to fill the reservoir volume away from geometric constraints: The first method places site points on a uniform mesh that covers the specified bounding box, the second optimizes the dual Delaunay triangulation using force-based smoothing, and the third minimizes the so-called centroidal PEBI grid (CPG) energy function. (Complete source code for all examples is available in `uprBookSection3.m`.)

1.3.1 Background Cartesian Grids

MRST has two options for generating Cartesian grids (see section 3.1 of the MRST textbook): `cartGrid([n,m])` gives uniform grids with a prescribed number of cells in each direction, and `tensorGrid(x,y)` generates a rectilinear grid with vertices given by `x'*y`. Specifying these vertices as sites to generate a PEBI grid will result in the Cartesian *dual*, in which each cell is shifted a distance $(\Delta x/2, \Delta y/2)$:

```
% UPR equivalent of cartGrid
[dx, dy]      = deal(0.25);
[xmax, ymax] = deal(1);
G = compositePebiGrid2D([dx, dy], [xmax, ymax]);
% Equivalent of tensorGrid
[X, Y] = meshgrid(linspace(0,1,5));
G      = pebi(triangleGrid([X(:), Y(:)]));
```



To avoid having half-size cells next to the boundary, you can use `clippedPebi` with site positions shifted `-[dx,dy]/2` to the centroids of the Cartesian grid.

We will return to `compositePebiGrid2D` in Section 1.4 and explain in more detail the methods this function implements for adapting the grid cells locally so that cell faces follow prescribed (internal) curve constraints.

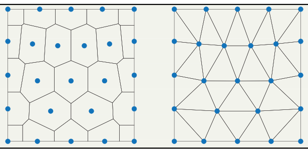
1.3.2 Delaunay Optimization

Cartesian grids are simple to create, ensure that standard two-point discretizations are consistent for isotropic permeabilities, and give nice cells away from constraints. On the other hand, if there are many constraints in the domain, you may want to use a completely unstructured grid. Placing sites for such grids manually can be a time-consuming task, so we instead seek some optimization process that can do this for us. The `upr` module implements two different optimization procedures. The first is based on optimizing the Delaunay triangulation using the DistMesh[1] software [16], discussed in subsection 3.2.4 of the MRST textbook.

DistMesh optimizes the Delaunay triangulation through a force-based smoothing procedure in which one first associates the vertices of the triangulation with joints and the edges by springs and then solves for force equilibrium. After the triangulation is optimized, we can construct the dual PEBI grid by using the vertices as sites. The `upr` module contains a slightly modified version of DistMesh (which, e.g., removes the time-consuming plotting) and supplies a wrapper function around the software that can be accessed by the `pebiGrid2D` function:

[1] DistMesh is distributed under the GNU GPL licence from persson.berkeley.edu/distmesh/.

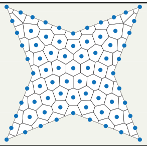
```
% Optimized Delaunay triangulation
h          = 0.25;
domain     = [1, 1];
[G, sites] = pebiGrid2D(h, domain);
Gt         = triangleGrid(sites);
```



The function also returns the optimized sites. This is useful for a variety of reasons; e.g., to recreate the dual Delaunay triangulation. However, if this is your only interest, you would be better off modifying the `pebiGrid2D` function to return the Delaunay triangulation before the PEBI grid is made or call DistMesh directly. It may be worth knowing that `compositePebiGrid2D` also returns the sites.

You can also adapt the grid to polygon boundaries, specified by a set of vertices ordered counterclockwise and supplied by the optional keyword `'polyBdr'`:

```
%% Star shaped boundary
bndr = [0, 0; 0.5, 0.2; 1, 0; 0.8, 0.5;
        1, 1; 0.5, 0.8; 0, 1; 0.2, 0.5];
% Create grid with h = 0.1
G = pebiGrid2D(0.1, domain, 'polyBdr', bndr);
```



1.3.3 *Minimized Centroidal Energy Function*

Instead of optimizing the Delaunay triangulation, we could optimize the PEBI grid directly. The optimization algorithm presented in this subsection tries to place the sites so that they correspond to the cell centroids, which is attractive for finite-volume schemes based on two-point flux approximations. The CPG energy function is defined as [7, 10]

$$F(\vec{s}) = \sum_{i=1}^{n} \int_{v_{s_i}} \|\vec{y} - \vec{x}_i\|^2 \, \mathrm{d}\vec{y}, \tag{1.3}$$

where $\vec{s} = [\vec{s}_1^{\top}, \ldots, \vec{s}_n^{\top}]^{\top}$ contains the sites and $\vec{x}_i$ denotes the mass centroid of the PEBI cell v_{s_i}. It can be shown that the CPG-energy function is minimum if and only if the cell centroids correspond to the sites [7]. Many different optimization methods have been used to find the minimum. For a long time, fixed-point methods were used, because the function was believed to lack regularity when the topology of the grid changed. However, Liu et al. [12] showed that the CPG-energy function is almost always two times differentiable. This allows for using

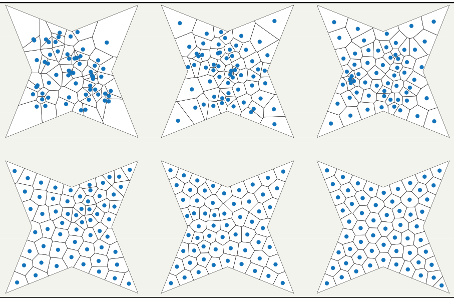
Newton's method in optimization to find the minimum of (1.3), but because constructing the full Hessian is memory demanding, quasi-Newton methods are usually preferred [12].

The upr module contains two methods, CPG2D and CPG3D, that use the limited-memory BFGS algorithm (implemented in upr as lbfgs) to find the minimum of the CPG-energy function:

```
% Domain boundary
bndr = [0, 0; 0.5, 0.2;
        1, 0; 0.8, 0.5;
        1, 1; 0.5, 0.8;
        0, 1; 0.2, 0.5];

% Initial sites
sites = 0.2 + 0.6 * rand(60,2);

% Call optimization routine
G = CPG2D(sites, bndr);
```



Here, the inset figures show how the grid develops from the initial grid constructed directly from the random sites (upper left), via iterations 3, 5, 10, and 20, to the final converged grid obtained after 40 iterations shown in the lower-right figure. Comparing the optimized CPG2D grid with the grid in Subsection 1.3.2, where the Delaunay triangulation is optimized, we see that the two grids are very similar. The main difference is that the Delaunay optimization will place sites on the boundary, whereas minimizing the CPG-energy tends to give boundary cells with volumes similar to the internal cells.

1.4 Internal Face Constraints

One of the main features of upr is its ability to include lower-dimensional constraints on the cell faces. By a lower-dimensional constraint we mean a surface in 3D or a line in 2D that should be traced by the faces of the neighboring grid cells. It is usually of interest to allow the constraining surfaces and lines to intersect; e.g., to represent crossing fault surfaces or fractures, well paths with branches, well paths crossing faults or fractures, and so on.

Creating a PEBI grid conforming to lower-dimensional objects can be very challenging. Special care must be taken around intersections of constraints that interact with each other. Because the constraints we are working with are lower-dimensional, the intersection of two constraints will have codimension two. This means that for a 3D domain, the intersections of 2D surfaces will be lines. Further, when lines intersect, which happens at the point where three or more 2D surfaces intersect, we get a 0D point intersection. The `upr` module supports two different ways of creating a grid conforming to surfaces. The first method assumes that the user has already tessellated the prescribed constraints by simplices. This method is the most robust and enables more control of the grid size and local grid refinement. A disadvantage is that tessellating a surface can be a challenge in itself, and unfortunately the `upr` module does not give much help with this. You can to some extent use the DistMesh software [16], but to tessellate very complex surfaces, you would be better off using a more advanced and robust third-party software like `Gmsh` [9] (gmsh.info).

For the second method, you only have to supply each surface as a polygon, from which `upr` calculates all intersections and builds a hierarchy of PEBI grids for all dimensions (3D, 2D, 1D, 0D). This is of interest not only to improve gridding but also for mixed-dimensional models for flow in fractured media. The resulting grids are constructed in ascending order based on the dimension. First, we create the grids of the 0D points, then the 1D lines, the 2D surfaces, and finally the 3D domain. The procedure can be stopped at any point if, e.g., only the 2D grids are of interest.

1.4.1 First Method: Simplex Conformity

The method assumes that each lower-dimensional object is tessellated by simplices. Points and line segments are simplices in 0D and 1D, respectively, and the first case to consider is therefore how to generate 2D grids conforming to 1D curves consisting of piecewise linear segments. We first go manually through the necessary steps using low-level library routines from the `upr` module to illustrate the basic principles, before we present the corresponding high-level routines that automate the process. (Complete source code: `uprBookSection41.m`.)

Doing it manually: Let $\{\vec{p}_i\}_{i=1,\dots,n}$ be the vertices of a 1D tessellation of a curve. Our goal is to place the 2D sites such that each cell of the 1D grid becomes a face in the 2D PEBI grid. Assume that the cells are ordered such that cell c_i has vertices $\vec{p}_i$ and $\vec{p}_{i+1}$. As an example, we consider a piecewise linear curve specified by four vertices that define three line segments (1D cells):

```
p = [0, 0.4; 0.2, 0.5; 0.4, 0.5; 0.6, 0.6];
d = sqrt(sum(diff(p).^2,2)); % distance between vertices
```

We start by drawing a circle around each vertex; see Figure 1.5. Requiring that the two circles associated with each 1D cell intersect gives us an upper/lower bound on the radii, $|R(\vec{p}_i) - R(\vec{p}_{i+1})| \le d_i \le R(\vec{p}_i) + R(\vec{p}_{i+1})$, where d_i is the distance

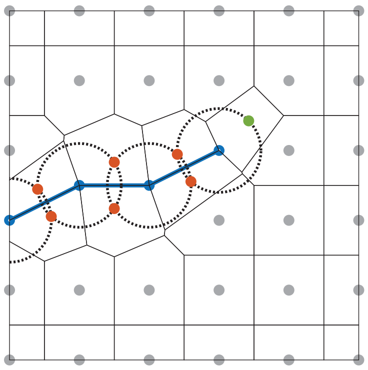


Figure 1.5 A 2D PEBI grid conforming to a piecewise linear curve. The blue points are the 1D vertices, the red points are sites added at the intersection of circles, and the green point is an extra site added to the rightmost circle. The gray dots are distributed to create a Cartesian background grid.

between the two vertices. We then place a set of sites where the circles intersect; shown as red dots in Figure 1.5.

```
R  = .6*min(d);                                     % Circle radius
dn = sqrt(R^2 - (d/2).^2);                          % Normal offset
t  = bsxfun(@rdivide, diff(p), d);                  % Tangent vector
n  = [-t(:, 2), t(:, 1)];                           % Normal vector
center = p(1:end-1, :) + bsxfun(@times, d/2, t);    % Segment centers
left_sites  = center + bsxfun(@times, dn, n);       % Sites to the left
right_sites = center - bsxfun(@times, dn, n);       % Sites to the right
```

For each interior vertex there are now exactly four sites located the same distance from the vertex. For the rightmost vertex we need to add an extra site, called a tip site, to have at least three sites located the same distance from the vertex. The leftmost vertex is on the boundary and hence we do not have to add a tip site here. Tip sites can be placed anywhere on the circumcircles of end vertices as long as they do not lie inside any other circle:

```
tip_sites = p(end, :) + R / sqrt(2);
```

Equation (1.2) tells us that the sites we have created so far are sufficient to create a 2D PEBI grid conforming to the 1D grid. The vertices of the 1D grid are vertices in the 2D grid, and the cells of the 1D grid are faces in the 2D grid. However, to populate the remaining domain with cells, we should add a few more sites, called background sites. Equation (1.2) also tells us that we can add such background sites to the domain in any way we would like, as long as the circles remain empty.

In practice, it is often most convenient to first distribute the background sites without thinking about the circles and then subsequently remove any sites inside a circle using; e.g., the `removeConflictPoints` function:

```
[X, Y]   = meshgrid(0:.2:1, 0:.2:1);
bg_sites = removeConflictPoints([X(:), Y(:)], p, R);
```

We then collect all of the different sites in a vector and construct the grid

```
bnd = [0, 0; 1, 0; 1, 1; 0, 1];
G   = clippedPebi2D([left_sites; right_sites; tip_sites; bg_sites], bnd);
```

MRST implementation: The `compositePebiGrid2D` and `pebiGrid2D` functions implement a generalized version of the procedure just outlined. You can invoke this functionality by supplying the 1D line constraints as a cell array using the optional `'faceConstraints'` keyword. You can see examples of how to use these function in the following example as well as in Section 1.6. The functions also handle intersecting lines and will adapt the prescribed grid size locally to avoid conflicting sites at any intersection. The main difference between them lies in how the functions distribute the background sites, as discussed in Subsections 1.3.1 and 1.3.2.

Example 1.3. To illustrate adaption of cell faces, we consider a case with four constraints; the first three are straight lines, and the last consists of two line segments. The first two constraints intersect at a sharp angle, whereas the last two intersect at an almost straight angle (see Figure 1.6).

```
  lines = {[0.2, 0.2; 0.7, 0.05], ...
           [0.2, 0.05; 0.7, 0.2], ...
           [0.1, 0.4; 0.6, 0.6], ...
           [0.1, 0.7; 0.45, 0.7; 0.55, 0.3]};
```

We use `compositePebiGrid2D` to generate the grid with structured topology everywhere, except near the constraints:

```
G = compositePebiGrid2D([0.05 .05], [1, 1], 'faceConstraints', lines);
% For PEBI, uncomment the next line
% G = pebiGrid2D(.05, [1 1],'faceConstraints', lines);
```

This makes it easier to see the adaptive parts, in contrast to `pebiGrid2D`, which gives unstructured topology everywhere (the resulting grid is not shown for brevity).

To avoid introducing arbitrarily small cells near the very sharp intersection between the first two constraints, the algorithm merged some sites locally around the intersection. If this is undesired, you need to prescribe a smaller grid size globally or locally around the constraint (as we will discuss in the next subsection). Notice also

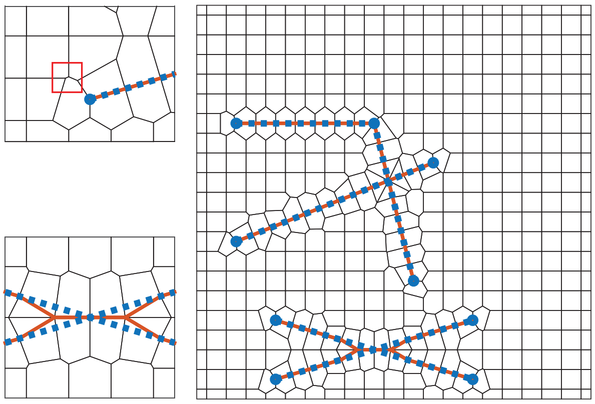


Figure 1.6 Grid conforming to four line constraints (in blue). Conforming faces are colored red. The lower-left plot zooms around two constraints intersecting at a sharp angle. Here, the corresponding cell faces are merged to avoid introducing small grid cells. The upper-left plot shows that the unstructured cells may contain very small faces, which is an undesired artifact. These can be removed by `removeShortEdges`.

the presence of very small faces in some of the unstructured cells (the ratio between the longest and shortest among the 1 105 faces is 969). This is difficult to avoid when constructing grids directly from a set of sites. In fact, it is even more pronounced for `pebiGrid2D`, because the background sites are generated by optimizing the Delaunay triangulation and not the Voronoi diagrams. With the same discretization parameter, `pebiGrid2D` gives 1 716 faces and a length ratio of 4 746. On the other hand, aspect ratios of this magnitude are not uncommon in real geomodels.

1.4.2 Configuring the Simplex-Conformity Methods

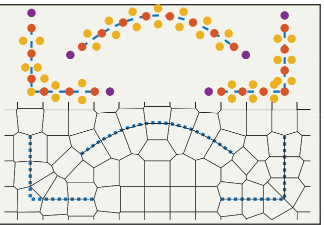
Now that the basic concepts of the simplex-conformity methods have been introduced, let us also look at some of the different features that are implemented in the two corresponding library functions. (You find complete source code for the examples in this subsection in `uprBookSection42.m`.)

Tessellating constraints: This is an important part of creating adapted grids and therefore has a dedicated library function that is used internally by both `pebiGrid2D` and `compositePebiGrid2D`:

```
F = surfaceSites2D(lines, h)
```

Here, `lines` is a cell array describing the constraints as arrays and `h` is the desired discretization parameter. The output parameter `F` is a structure that contains the sites on opposite sides of each constraint, the circle centers (i.e., the vertices of the 1D tessellation), as well as the tip sites, all shown in Figure 1.5. The default method subdivides each line segment into 1D cells of approximate length `h` or, to be precise, a segment of length `L` is divided into `ceil(L/h)` 1D cells. This works well as long as each segment is longer than `h`. However, the function can also interpolate points along the curve, which is useful, e.g., if a constraint consists of many segments that are significantly shorter than `h`. The following example illustrates this:

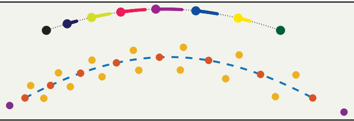
```
% Interpolating constraints
x     = linspace(0.2,0.8);
y     = 0.4+0.3*sin(pi*x);
lines = {[x(:), y(:)], ...
         [.0,.65; .0,.4; .25,.4],...
         [.75,.4; 1.0, .4; 1.0, .65]};
F = surfaceSites2D(lines,0.1, ...
        'interpolateFC', [true,true,false]);
```



The curved constraint is represented by 100 points and is tessellated correctly by the interpolation method. The left L-shaped constraint, on the other hand, is cut at the corner and will not be represented exactly in a resulting grid. Fortunately, supplying the `'interpolateFC'` keyword with an array of Boolean variables with one entry per constraining line enables you to pick a method individually for each constraint.

The interpolation function also accepts a function handle that enables you to control the discretization parameter in space:

```
% Local control of cell length
F = surfaceSites2D(lines(1),0.1, ...
        'interpolateFC', true,...
        'distFun', @(x) .05+.125*x(:,1));
```



This functionality is implemented using a force-balanced smoothing similar to DistMesh along each constraint. The upper figure shows how the eight vertices move during the force balancing; small dots are intermediate positions and larger dots are the final positions. In the lower figure, the final positions are shown as red dots, and orange dots represent sites placed at the intersections of the corresponding circles. The parameter `'circleFactor'` sets the ratio between the radius and distance between the circles (i.e., between neighboring vertices in the 1D tessellation). The parameter takes values in the interval [0.5,1] and has a default

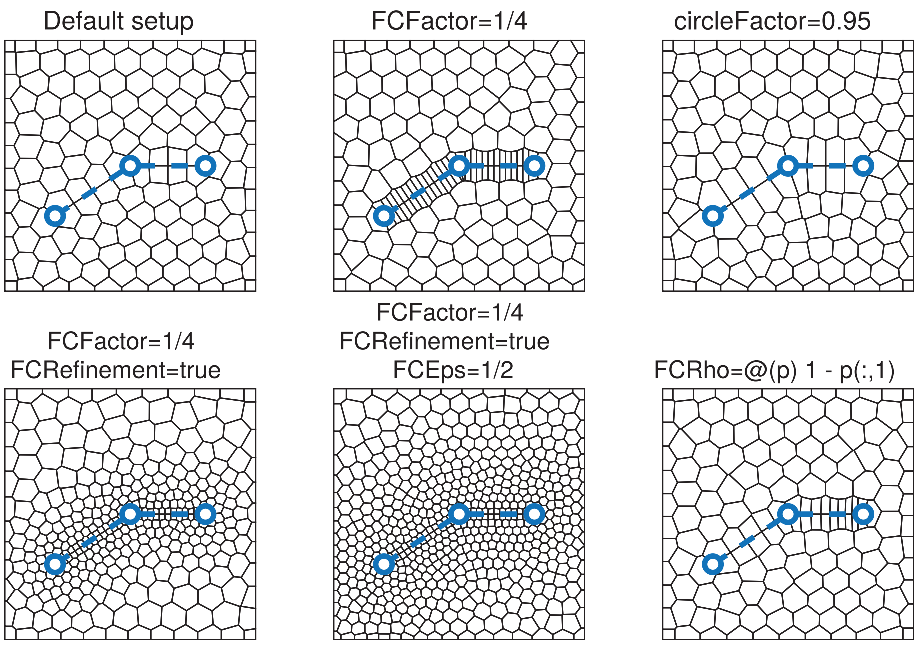


Figure 1.7 The various options you can use to control how `pebiGrid2D` adapts to face constraints.

value of 0.6. Setting a smaller value will place the sites closer to the constraint curve, whereas a larger value will place them farther from the constraint.

Controlling volumetric adaption: All parameters just described can be passed to `pebiGrid2D` and `compositePebiGrid2D` to control tessellation and placement of sites along constraints. The `pebiGrid2D` routine offers some additional parameters you can use to control how the grid adapts to the face constraints (see Figure 1.7 and the example `showOptionValuesPebiGrid.m` in the `upr` example folder):

- `'FCFactor'` sets the ratio of the distance between the vertices along the tessellated constraints to the distance between reservoir sites. With a value of 0.5, the dimension of the cells next to the constraint will be approximately half the dimension of the background cells. The default value is 1.
- `'FCRho'` sets the function assigned to `'distFun'` to control how vertices are distributed along the constraints. The default value is `@(x) ones(size (x,1),1)`.
- `'FCRefinement'` set to `true` (defaults to `false`) will ask DistMesh to gradually refine the background grid toward the face constrains. Technically, this is

done by passing a scaled distance function $1.2\exp(\min_y \|x - y\|/\varepsilon)$ to DistMesh. Here, y are the face sites, which are enforced as fixed points, and the scaling ε can be set by the parameter `'FCEps'`. If this parameter is not set, `upr` will make a guess based on the domain size.

You can also use the `'FCFactor'` parameter to specify how densely face constraints are tessellated in the composite grid compared to the grid size of the background Cartesian mesh.

1.4.3 Second Method: PEBI Conformity

This subsection shows how a hierarchy of PEBI grids of different dimensions can be constructed in such a way that the grids of one dimension conform to all grids of one dimension lower, which in turn enables us to construct a 3D PEBI grid that conforms to 2D intersecting surfaces. To introduce and explain the necessary methodology, we will use a conceptual setup with perpendicular surfaces:

```
surf = {ellipticSurface3D([3,3, 3], 1.5, 1.5, 0, 0, pi/2), ...
            [2,2,3.3; 5,2,3.3; 5,4,3.3; 2,4,3.3], ...
            [3,1, 1;  3,5,1;   3,5,5;   3,1,5]};
```

Here we have used a utility function `ellipticSurface3D` from the `upr` module to create a polygon approximating an elliptic surface, which is the typical shape seen for hydraulic fractures. The two planes can be thought of as natural fractures. The example is primarily designed to highlight the basic methodology and is not necessarily representative of real physical systems. (Source code: `uprBookSection43.m`.)

To construct a conforming grid in 3D, the user must supply the constraining 2D surfaces as a set of convex (flat) polygons. The first step to create the conforming grid is to calculate the intersection of all of the polygonal surfaces. This is done by the following function:

```
intersections = surfaceIntersections3D(surf);
```

Figure 1.8 shows the surfaces and the intersections. After the intersections are calculated, we construct the grids of each dimension. We start by creating the 1D grids of the intersection lines:

```
ds    = 0.25;              % Mesh size
gamma = ds./[1, 4, 8];     % Offset of lower-dimensional sites
grids1D = lineGrid3D(intersections, gamma(1));
```

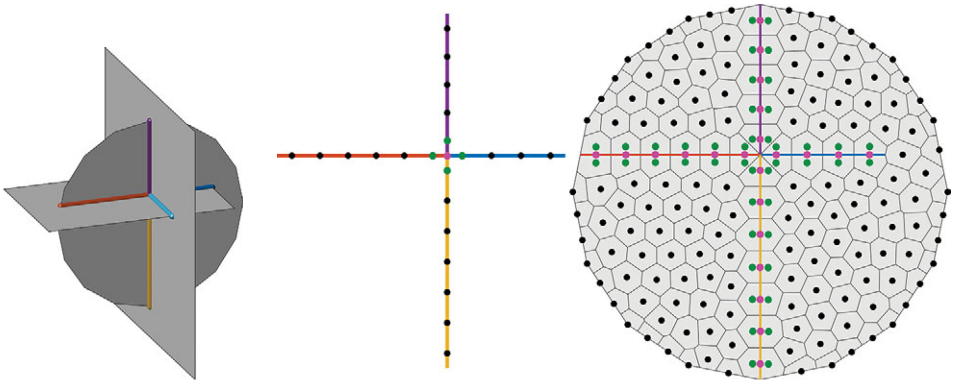


Figure 1.8 The three surfaces shown in the left plot define six intersection lines (colored lines). To create a conforming 2D grid of the circle (right plot), the corresponding 1D intersection lines are first gridded (center plot). The pink points represent sites of the lower-dimensional grids, the green points represent sites that are added to enforce conformity to the lower-dimensional grids, and the black points are sites that are added to create the background grid.

The 1D intersection lines returned from `surfaceIntersections3D` are split so that they form a set of nonintersection lines (except possibly at the end points). Then, `lineGrid3D` places two sets of sites. The first set contains sites that are placed a distance `gamma(1)` from the 0D intersection. It is important that the closest site to the 0D intersection is exactly `gamma(1)` for all lines that intersect at this point; otherwise, the algorithm will fail when creating the 2D grid, because the sites placed to conform to one line will interfere with the sites placed to conform to the other line. The second set of sites contains the remaining background sites and could be placed freely. However, the `surfaceIntersections3D` function tries to distribute them equidistantly. Figure 1.8 shows the 1D sites of the ellipsoidal surface.

The next step is to create the grids of the 2D surfaces using `surfaceGrid3D`:

```
grids2D = surfaceGrid3D(surf, grids1D, intersections, ds, gamma(2));
```

In the same way as for the 1D grids, the gridding is done in two steps. The first step is to create two duplicates of the 1D sites and then move the duplicates a distance `gamma(2)` in the two in-plane normal directions of the 1D line. In the second step, the remaining background sites are generated and the function `surfaceGrid3D` uses the optimized Delaunay triangulation to place these sites. The sites of the ellipsoidal surface are shown in Figure 1.8. Notice that the 2D grid conforms not only to the 1D grids but also to the 0D intersection.

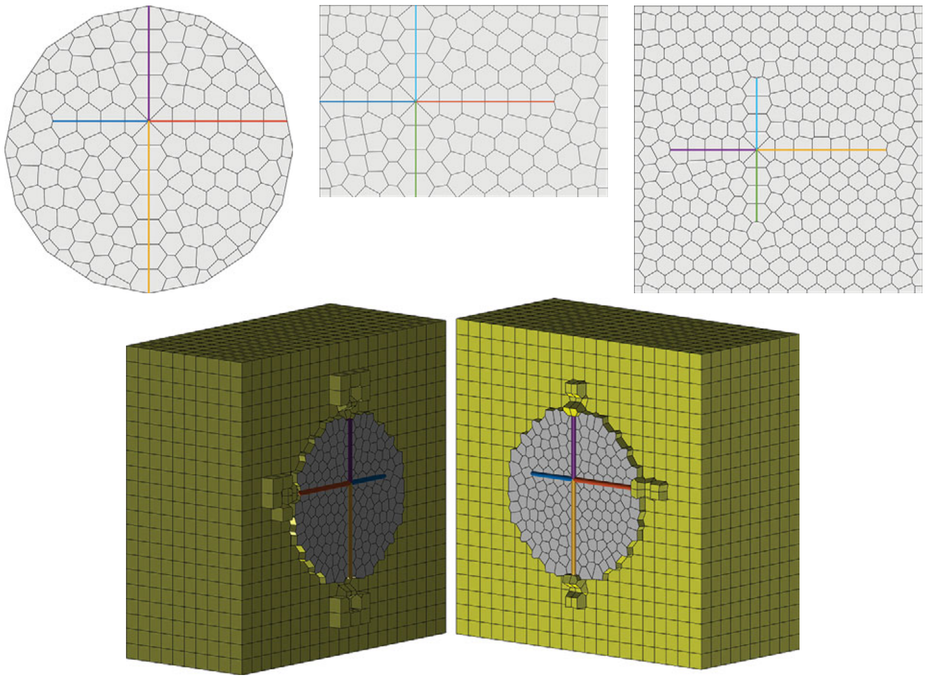


Figure 1.9 The top row shows 2D grids of the three constraining surfaces from Figure 1.8, with 1D grids outlined in different colors. The bottom row shows the 3D grid, which is opened up along the circular polygon.

The final step is to create the 3D grid conforming to the 2D surfaces, which is done by calling the `volumeGrid3D` function:

```
G3 = volumeGrid3D([6, 6, 6], surf, grids2D, ds, gamma(3));
```

This function works similar to the `surfaceGrid3D` function in that we first make two duplicates of the lower-dimensional sites and then move them a distance `gamma(3)` in the two normal directions of the 2D surface. The background sites are generated as a uniform Cartesian mesh. Figure 1.9 shows the full hierarchy of PEBI grids, from the 0D intersection of the three fracture planes, via the 1D grids representing pairwise intersection of fracture planes, each of which is represented by a 2D grid, and to the full 3D volumetric tessellation.

This procedure generates a grid of each dimension and gives much nicer cells conforming to the 2D surfaces than the simplex algorithm discussed in Subsection 1.4.1. The disadvantage is that the hierarchical construction is not as robust with respect to intersecting surfaces. If the surfaces intersect at sharp angles, they very easily interfere with each other and ruin the conformity of the mesh.

1.5 Adapting Cell Centroids

The upr module also supports cell-centroid conformity to piecewise linear curves (in 2D or 3D). Tracing such curves by cell centroids is of interest, e.g., for well paths. In this section we discuss how to create 2D grids conforming to cell constraints using the simplex-conformity method; Subsection 1.6.5 shows a 3D example.

In upr, each cell constraint is represented by the vertices of a piecewise linear curve, and we can use the function lineSites2D to distribute a set of points evenly along the line segments. As an example, consider a single constraint consisting of three line segments:

```
% Constrained lines
cellConstraints = {[0, 0.4; 0.2, 0.5; 0.6, 0.5; 0.8, 0.6]};
[CCSites, cGs]  = lineSites2D(cellConstraints, 0.12);
```

This function works much in the same way as surfaceSites2D discussed in Subsection 1.4.2 in the sense that it takes a cell array of piecewise linear paths and a desired grid size and distributes a set of sites along these paths. The function returns the constrained sites as well as the distances between them.

To create the complete grid, we also have to distribute the background sites; here, we place them equidistantly. As for the surface constraints in Section 1.4, background sites should not be closer to the cell constraint than the cell width, and hence we use the function removeConflictPoints to remove sites that are too close. This will guarantee that two consecutive constrained cells have a face connecting them.

```
[X, Y]  = meshgrid(linspace(0,1,10));
bgSites = [X(:), Y(:)];
bgSites = removeConflictPoints(bgSites, CCSites, cGs);
sites   = [CCSites; bgSites];
bnd     = [0, 0; 1, 0; 1, 1; 0, 1];
G       = clippedPebi2D(sites, bnd);
```

Figure 1.10 shows the resulting grid. Note that the procedure just outlined cannot guarantee that centroids of the final grid cells coincide with the cell sites, because conformity depends on how the background sites are placed. However, if background sites are placed by one of the optimization algorithms presented in Section 1.3, the cell centroids are usually very close to the prescribed path.

Adapting cell centroids to paths may lead to rather irregular constrained cells for a Cartesian background grid. To improve the conformity and make the constrained cells as rectangular as possible, we can add a set of protection sites around each cell

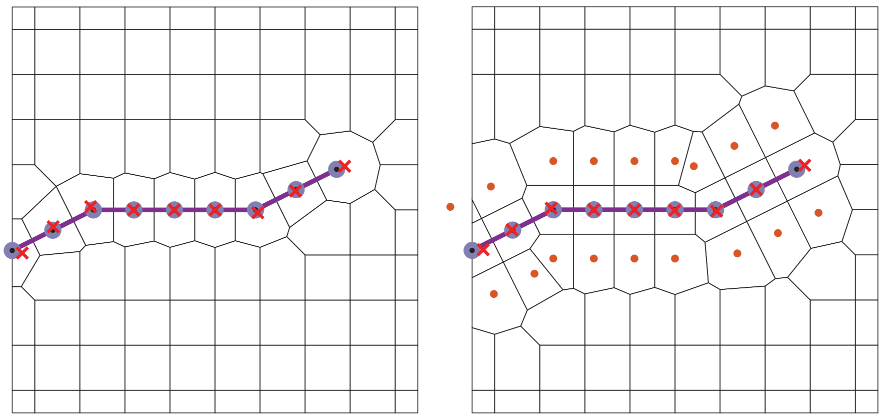


Figure 1.10 PEBI grids conforming to a cell constraint shown as a purple line. The grid to the left does not have protection sites, whereas the right figure shows the same grid with protection sites (orange dots). The constrained sites, shown as gray dots, do not fully coincide with the cell centroids shown as red crosses.

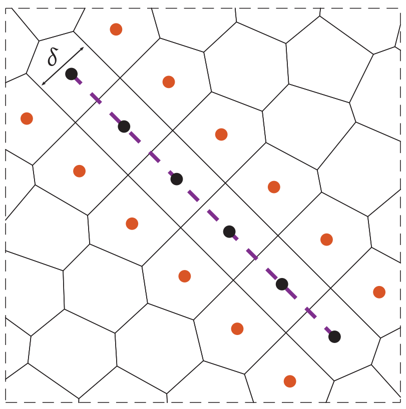


Figure 1.11 Construction of protection sites along a path. The constrained sites are shown as black dots and the protection sites as red dots. The width of the constrained cells is given by the distance, δ, which equals the distance from the constrained sites to the protection sites.

constraint. This is done by tracing the constrained path and making two duplicates, called protection sites, of each constrained site. These protection sites are shifted a length δ in opposite directions normal to the path; see Figure 1.11. This not only ensures that cell centroids trace the path but also enables us to explicitly control the width of the constrained cells, which, after the protection sites are added, will be δ. Whether this grid is better for flow simulation depends on the spatial discretization.

Protection sites are added by setting 'protLayer' to true and possibly supplying the keyword 'protD' to define the offset δ:

```
distance = @(x) 0.12 * ones(size(x, 1), 1);
[CCSites, cGs, protSites, pGs] = ...
   lineSites2D(cellConstraints, 0.12, 'protLayer',true,'protD', {distance});
```

The function returns the protection sites protSites as well as their associated grid size pGs, which can be used to eliminate conflicting background sites to create the grid shown to the right in Figure 1.10:

```
bgSites = removeConflictPoints(bgSites, protSites, pGs);
sites   = [CCSites; protSites; bgSites];
G       = clippedPebi2D(sites, bnd);
```

As for the face constraints, the simplest way to create grids conforming to cell constraints is by using the wrapper function pebiGrid2D for an unstructured background grid and compositePebiGrid2D for a structured background grid. These functions enable adaption to both cell constraints and face constraints and will handle intersections as well.

As an example, we add two cell constraints to the case from Figure 1.6. The first constraint is horizontal and follows a prescribed curve, whereas the second is vertical and given as a single point. The grid resolution is set to increase toward both cell constraints so that the constrained cells are four times smaller than the background cells (see Figure 1.12):

```
cellConstraints = {[0.1, 0.6; 0.2, 0.6; 0.3, 0.5; 0.4, 0.3], [0.8, 0.8]};
G = pebiGrid2D(0.06, [1, 1], 'faceConstraints', lines, 'cellConstraints',...
               cellConstraints, 'CCRefinement',true, 'CCFactor', 0.2);
```

Controlling the adaption: Through the examples discussed in this section, we have already outlined several parameters to customize adaption to cell constraints, including 'CCRefinement', 'CCFactor', 'protLayer', and 'protD'. Likewise, parameters similar to those outlined in Subsection 1.4.2 for controlling the tessellation of face constraints apply to cell constraints: 'interpolateCC' lets you select between sampling and interpolation of cell constraints, whereas 'CCRho' and 'CCEps' determine how the background cells adapt to the constraint paths for pebiGrid2D. If you prescribe density functions for controlling the cell size based on the distance to both cell and face constraints, the algorithm uses the minimum value of the two; i.e., the distance to the nearest

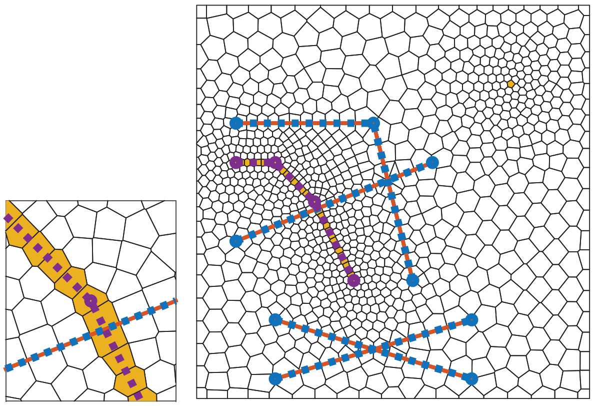


Figure 1.12 A PEBI grid conforming to four face constraints (blue), one curved cell constraint (purple), and one point cell constraint. The conforming faces are colored red, and the conforming cells are colored orange. The left plot shows a zoom of the grid around the intersection of the cell constraint and the face constraint.

constraint. For more examples of how these parameters are used, see the example `showOptionValuesPebiGrid.m` in the `upr` example folder.

For composite grids, it is also possible to add one or more levels of local refinement, as shown in Figure 1.13. These refinements are controlled by two parameters: `'mlqMaxLevel'` sets the number of refinement levels and `'mlqLevelSteps'` specifies the outer radius for each level. Each new level represents a 2×2 refinement of the reservoir sites, and to ensure that the cell constraints are tessellated densely enough, it is important that you adjust `'CCFactor'` to either match or be smaller than the distance between reservoir sites at the finest refinement level.

1.6 Worked Examples

In this section, we will go through a number of examples to demonstrate how the functionality outlined so far can be used to create adapted grids in a more realistic setting. Because the examples are more comprehensive, it is not natural to discuss the necessary MATLAB code of all examples at the same level of detail we have done in the previous sections. However, we emphasize that you can find all of the necessary details in the accompanying example scripts under `book-ii` in the example folder of the `upr` module.

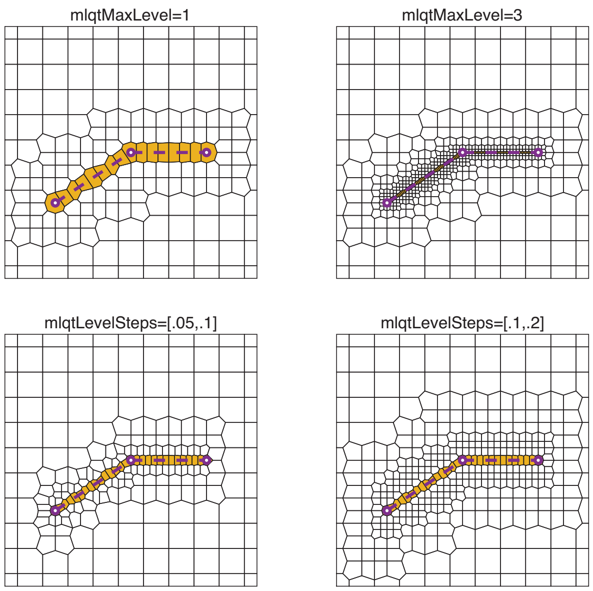


Figure 1.13 Local refinement for `compositePebiGrid2D` is controlled by two parameters that specify the number of levels and the outer radius for each level. Here, we have set `'CCFactor'` to $1/2^{\ell}$, where ℓ is the number of refinement levels, so that the tessellation of the constraint curve matches the resolution of the innermost reservoir sites.

1.6.1 Complex Fault Network in 2D

Our first example is taken[2] from Branets et al. [4] and describes a planar projection of a hydrocarbon reservoir with a complex fault network. Such fault networks are very challenging to represent accurately using corner-point grids and is one of the key motivations for using PEBI grids. (Complete source code is found in `complexFaultNetwork.m`.)

We start by loading the data set into MATLAB:

```
load(fullfile(mrstPath('upr'), 'datasets', 'gridBranets.mat'))
```

This gives us two data objects: a 41×2 array `bdr` with the points in the polygon that describe the reservoir boundary and a 1×21 cell array `fault` that contains the line segments of the individual fault lines. To create a grid of this domain we can use the wrapper function `pebiGrid2D` with the arguments:

[2] We do not have access to the exact data used by Branets et al. [4] but used Inkscape to draw an accurate reproduction and then extracted the coordinates of the fault lines and the reservoir perimeter from the resulting `svg` file.

```
h  = max(bdr(:))/50;   % assuming minimum point is at the origin
Gd = pebiGrid2D(h, [], 'faceConstraints',fault,'polyBdr',bdr,'interpolateFC',true)
```

To use the optimization algorithm in Subsection 1.3.3, prepossessing of the faults is needed. First, the fault curves are split at their intersections

```
  [fault, fCut] = splitAtInt2D(fault, {});
```

Altogether, this results in 75 noncrossing fault curves that can be tessellated and used to generate fault sites:

```
  F = surfaceSites2D(fault, h,'fCut',fCut,'interpolateFC',true);
  % Remove tip sites outside domain
  innside = inpolygon(F.t.pts(:,1), F.t.pts(:,2), bdr(:,1), bdr(:,2));
  F.t.pts = F.t.pts(innside, :);
```

The tip sites of the `F` struct also include tip sites for constraints reaching the boundary, and the last two lines remove these because they lie outside the domain boundary. The next step is to generate suitable reservoir sites. To this end, we first generate 1 500 points within the bounding box of the domain and then remove all points that are outside the domain or conflict with the fault sites:

```
  pInit = bsxfun(@times, rand(1500,2), max(bdr)-min(bdr));  % generate and scale
  pInit = bsxfun(@plus, pInit, min(bdr));                    % translate
  keep  = inpolygon(pInit(:,1),pInit(:,2),bdr(:,1), bdr(:,2)); % true inside bdr
  pInit = pInit(keep,:);
  pInit = removeConflictPoints(pInit,F.f.pts,F.f.Gs);
```

We can now create the PEBI grid as a centroidal Voronoi diagram by minimizing the CPG energy function in Equation (1.3) using the L-BFGS algorithm. We do this by calling the corresponding wrapping function:

```
  G = CPG2D(pInit, bdr,'fixedPts', F.f.pts,'maxIt',10);
```

This iteration can consume considerable time to fully converge, but you usually get quite good results after a few tens of iterations. Here, we have used only 10 iterations, taking approximately 90 seconds on a (mid-range) laptop. As a comparison, the `pebiGrid2D` algorithm (which uses the Delaunay optimization) takes approximately 10 seconds to generate the grid. Figure 1.14 shows the input data, the generating points, and the grids generated by the two different methods. Using `pebiGrid2D` gives 66% more cells when supplying the same resolution, because this resolution now also governs the tessellation that places reservoir sites. For larger values of `h`, the routine fails to provide conformity.

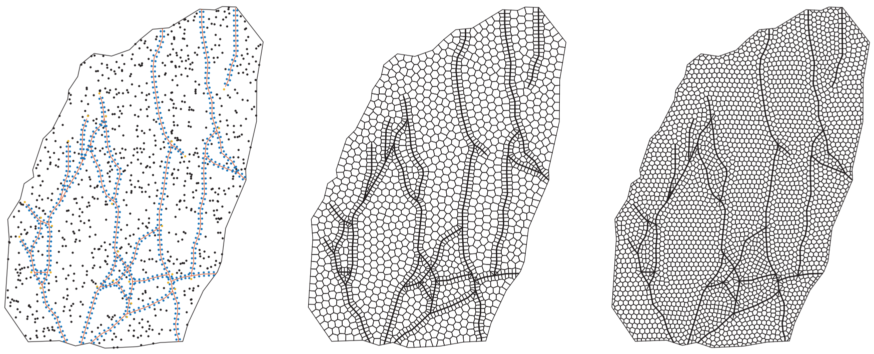


Figure 1.14 PEBI grids for a planar projection of a hydrocarbon reservoir with a complex fault network from [4]. Initial sites (left), optimized centroidal PEBI grid with 1 955 cells created by `CPG2D` (middle), and grid with 3 260 cells generated by `pebiGrid2D` (right). (Source code: `reservoirWithComplexFaultNetwork.m`.)

1.6.2 Statistical Fracture Distribution

In the next example, we consider an example of a fractured medium and use the `upr` module to generate a PEBI grid with fractures represented as lower-dimensional objects. (Such a grid is suitable for simulations with DFM models.) Our data set comes from the hierarchical fracture module (`hfm`) in MRST and consists of 51 fracture lines that have been statistically generated to mimic fracture patterns observed in carbonate outcrops. (Complete source code is found in `statisticalFractures.m`.)

From Figure 1.15 we see that there are several fractures that are very close to each other without intersecting. This essentially means that we should refine the grid locally in these areas. The `upr` module does not have any functionality for doing this automatically, so some manual work is necessary. That is, we mark a number of points that need particular focus and introduce local grid refinement by setting up a grid density function whose values decay exponentially as we approach any of the points of interest

$$h(x) = \min\big(1, \min_i \big(a_i \exp(\|x - y_i\|/\varepsilon_i)\big). \tag{1.4}$$

Here, a_i and ε_i are scaling parameters that are specific to each focus point y_i. The lower plot in Figure 1.15 shows the resulting density function. Once this function has been properly set up, we generate the grid shown in Figure 1.16 with

```
G = pebiGrid2D(15, [35,120], 'faceConstraints', lines, 'FCFactor',1/50,...
         'circleFactor', 0.55, 'FCRefinement', true, 'FCEps', 5,...
         'FCRho', FCRho, 'useMrstPebi', true, 'linearize', true);
```

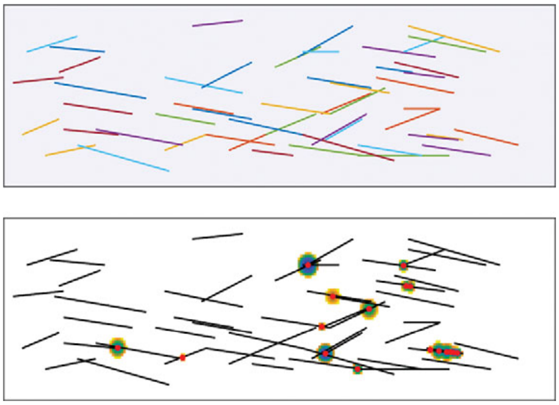


Figure 1.15 The statistical fracture case. The upper plot shows the 51 fracture lines. The lower plot shows focus points marked as a red dot and the grid density function, with white color denoting a unit value.

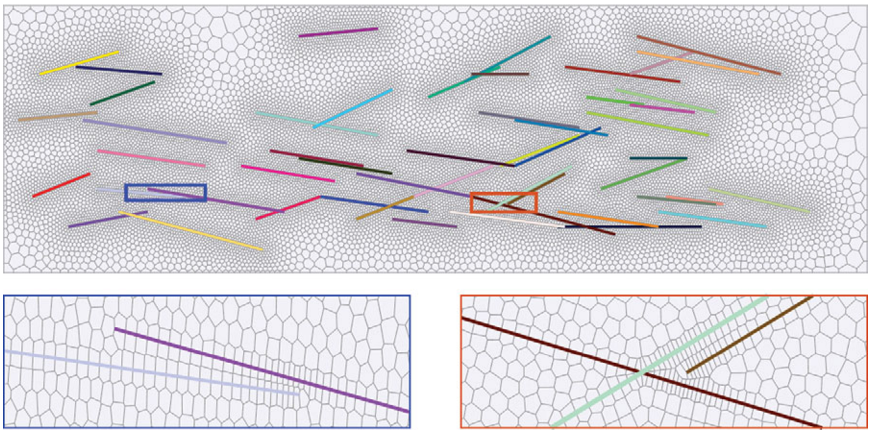


Figure 1.16 The grid generated for the statistical fracture case. The lower plots show enlarged views of the regions around two focus areas where fractures are very close without intersecting.

Here, you should notice the last parameter `'linearize'`. For a grid with many sites from face and/or cell constraints, the distance function DistMesh used to determine the local grid resolution becomes expensive. In each iteration, this function is evaluated at the midpoint of every edge of the Delaunay triangulation. By turning on the linearization flag, we instead use a new method we have implemented that evaluates the distance function in the triangle vertices and *interpolates* linearly to the midpoint on the triangle edges. This significantly reduces the overall computational time (by approximately 60% in this particular example).

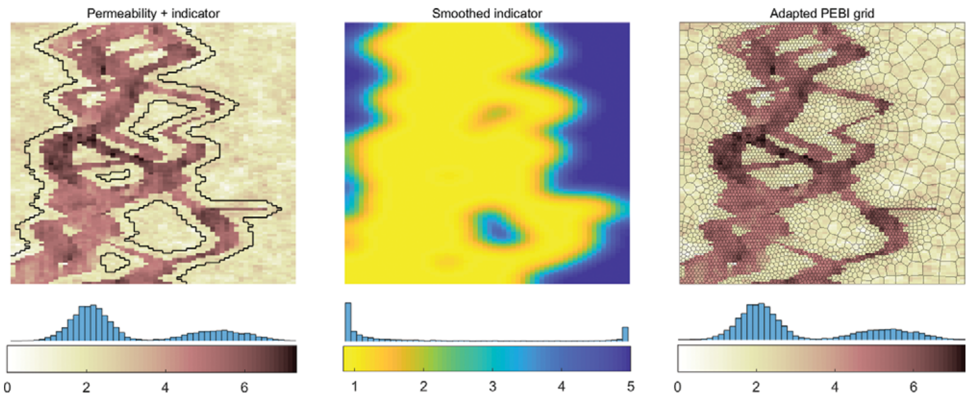


Figure 1.17 Grid adapted to the permeability of the fluvial Upper Ness formation from SPE10 using grid density functions in DistMesh to determine reservoir sites. The left figure shows the logarithm of the permeability field; the black line shows the result of growing the binary indicator three cells away from the initial interface between high and low permeability. The middle plot shows the grid density function used in DistMesh. The right plot shows the resulting PEBI grid.

1.6.3 Adapting to Permeability (SPE10)

Our next example is inspired by Branets et al. [3] and illustrates how you can use methods from DistMesh and the `upr` module to derive a reduced grid that adapts to permeability in high-contrast systems. As an example of such a system, we consider a waterflooding scenario posed on one half of a horizontal layer from the fluvial Upper Ness formation in Model 2 of the SPE10 benchmark [5] with injection along the south boundary and production along the north boundary. (Full source code is found in `spe10LayerGrid.m`.)

We start by defining a Boolean indicator `ind` by segmenting the logarithm of the permeability into two bins that approximately represent the low-permeability shales and coal and the high-permeability channel sand and then using the adjacency matrix (see [11, subsection 14.3.3]) to grow this indicator a few cells wider:

```
lperm = log10(rock.perm(:,1));
ind   = lperm > min(lperm) + 4;
N     = getNeighbourship(G);
A     = getConnectivityMatrix(N);
for i=1:3, sum(A*ind,2) > 0; end
```

The reason we grow the indicator is to ensure that we have small cells on both sides of a strong contrast; see the left plot in Figure 1.17. We then define the grid density function by first smoothing the indicator and then using a scattered interpolator from MATLAB to interpolate at any point:

```
S  = 2*A + speye(G.cells.num);
S  = S./sum(S,2);
for i = 1:20, ind = S*ind; end
x  = G.cells.centroids;
fh = scatteredInterpolant(x(:,1), x(:,2), 1./(ind + 0.2));
fh = @(x) fh(x(:,1), x(:,2));
```

Finally, we use DistMesh to create reservoir sites that are sent to `clippedPebi2D`. This gives the grid shown to the right in Figure 1.17, whose 3 993 cells represent a 40% reduction in cell count compared with the original Cartesian grid.

As an alternative, we can try to create a grid that conforms to the outline of the high-permeability channels. For this to work well, the outline cannot be too irregular. Instead of growing a binary segmentation outward, we first replace each cell value by the maximum over the cell and its four face neighbors and then perform a few smoothing steps:

```
nc  = G.cells.num;
I   = sparse(N(:,1), N(:,2), max(lperm(N),[],2), nc, nc);
ind = full(max(max(I,[],2), max(I,[],1)'));

for i = 1:3, ind = S*ind; end  % Smoothen out the indicator
```

This local mean filter reduces the heterogeneity of the permeability field so that a subsequent contouring algorithm will give a smoother outline and avoid small-scale clutter from salt-and-pepper effects inside the high-permeability channels:

```
xc = reshape(G.cells.centroids(:,1),G.cartDims);
yc = reshape(G.cells.centroids(:,2),G.cartDims);
c  = contourc(xc(:,1), yc(1,:)', reshape(ind, G.cartDims)',1);
```

We can then extract the individual lines generated by the contouring algorithm, split any circular lines, and use the resulting lines as constraints in the grid generation:

```
G2 = pebiGrid2D(dx, L, ...
      'faceConstraints', permLines  , ... % Lines
      'interpolateFC'  , true       , ... % Interpolate faults
      'FCRefinement'   , true       , ... % Refine reservoir sites
      'FCFactor'       , 0.09       , ... % Relative fault cell size
      'FCEps'          , 0.07*max(L), ... % Size of ref transition region
      'linearize'      , true);
```

Figure 1.18 shows the smoothed contour and the resulting grid. At a first inspection, the grid seems to be okay and have the features we desire in a reduced model: high grid resolution near contrasts between high and low permeability and somewhat

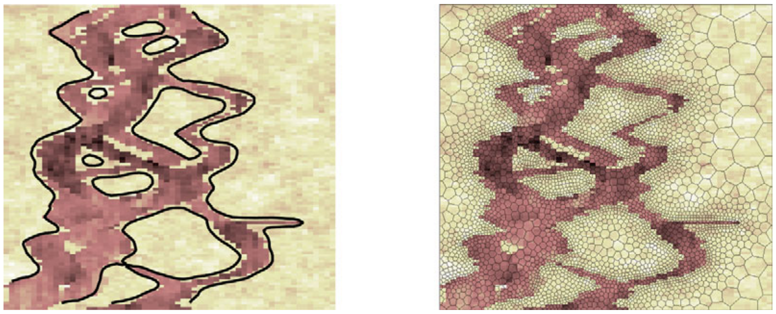


Figure 1.18 Adapting a PEBI grid to the smoothed outline of the high-permeability channels of the SPE10 example; left plot shows contours, right plot shows the resulting grid.

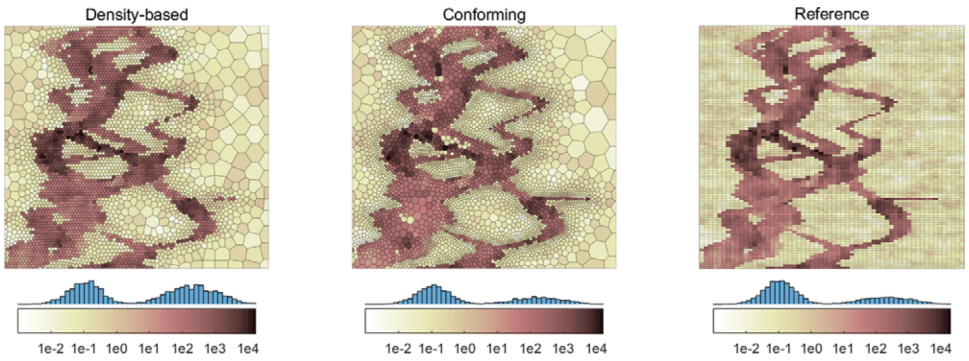


Figure 1.19 Reduced PEBI grids with resampled permeability compared with the original model.

lower resolution in regions with relatively homogeneous permeability. With 5 012 cells, the grid represents a 24% reduction in the number of grid cells compared with the original Cartesian grid.

Figure 1.19 shows resampled permeability on the two reduced models. It is clear that the conforming grid preserves the original permeability field much better than the density-based grid, not only in the histogram of the permeability values but also in how the two grids resolve thin high-permeability channels. These have lost their continuity in the first grid but are represented well by the second grid.

We end this example by comparing the result of a flow simulation. To this end, we use the fluid model from the original SPE10 benchmark and inject water from the south boundary, driven by a pressure difference of 99 bar from the south to the north boundary. With prescribed pressures only, we are not guaranteed to get the same total flow rate through the model, and we also observe in Figure 1.20

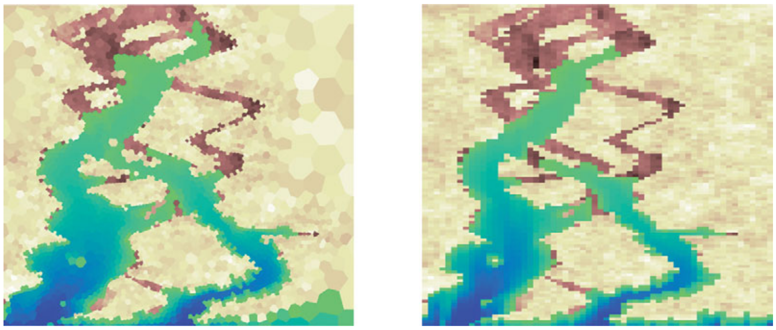


Figure 1.20 Comparison of the water saturation predicted on the contouring-based PEBI grid (left) and the original Cartesian grid (right) after time step number 12.

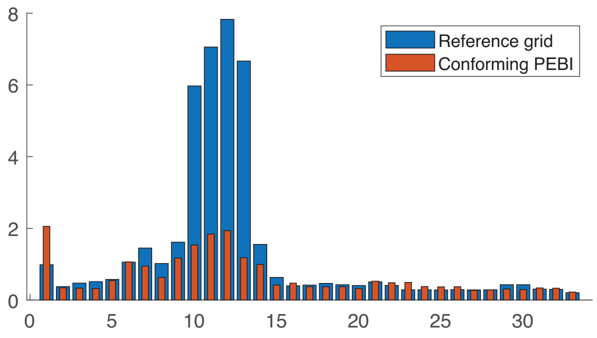


Figure 1.21 Runtime in seconds per time step for a waterflood simulation posed on the SPE10 case. For the reference grid, the nonlinear solver failed to converge and had to cut the time step in steps 10 to 13; hence the longer runtime.

that the tip of the saturation front penetrates further into the reservoir for the PEBI grid than in the simulation on the Cartesian grid. However, the overall match between the two simulations is very good. (The solution changes very little for a *refined* PEBI grid with three times as many cells, whereas PEBI grids with fewer cells appear to predict higher overall flow rate and more penetration.) The computational time is reduced by a factor 50% on the PEBI grid, which, unlike the original Cartesian grid, does not suffer from time step chops; see Figure 1.21.

1.6.4 Conforming to Triangulated Surfaces in 3D

In our first 3D example, adapted from [2], we generate a PEBI grid conforming to two curved faults represented as triangulated surfaces. To this end, we will

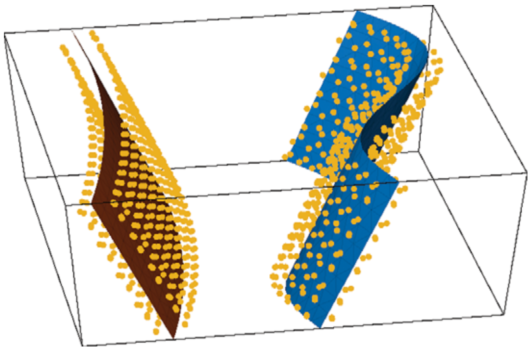


Figure 1.22 Two fault surfaces with corresponding fault sites placed at the intersection of the circumspheres of the vertices in the triangulations describing each fault.

essentially follow the same procedure as in 2D. Our starting point is an array `bdr` with the eight vertices that define the bounding box of the domain as well as two triangulations `t1` and `t2` of the curved surfaces given on standard MATLAB format; i.e., as structures that contain the points and their connectivity list. (Creating these structures constitutes a major part of the script `twoCurvedFaults.m`.)

To create fault sites, we must first draw spheres around each vertex in the triangulations of the two surfaces. All cells in each triangulation have three vertices, and we add a site where these three spheres intersect:

```
R = @(p) 1/20 * ones(size(p, 1), 1); % Radius of spheres
F = surfaceSites3D({t1, t2}, {R, R});
```

Compared with its 2D equivalent, this function is less sophisticated and does not contain any functionality for interpolating or subdividing the cells on the constraining surface. If this is desired, you must specify the input surfaces accordingly. Figure 1.22 shows how the fault sites are distributed around each of the two fault surfaces. We can then add equidistant background sites covering the interior domain and remove any sites inside the spheres to give the grid shown in Figure 1.23.

```
% Get centroids of a uniform mesh covering the bounding box
h  = 1 / 20;
xr = min(bdr(:,1))+h/2: h : max(bdr(:,1)-h/2;
:
[X,Y,Z] = ndgrid(xr, yr, zr);

% Create reservoir sites and construct grid
rSites  = surfaceSufCond3D([X(:),Y(:),Z(:)], F.c.CC, F.c.R);
G       = mirroredPebi3D([F.f.pts; rSites], bdr);
```

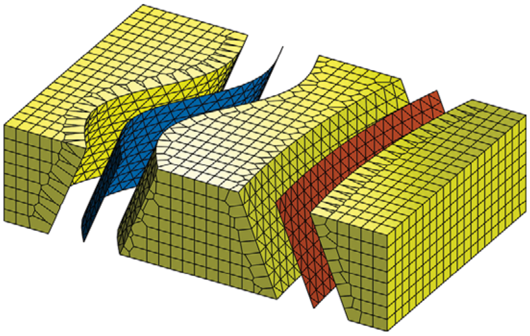


Figure 1.23 The 3D PEBI grid (yellow) conforms to two triangulated surfaces (red and blue). The grid is shifted away from the surfaces in the illustration to visualize the conforming faces.

Along the fault, the volumetric grid cells have triangular faces to match the prescribed surface constraint, which complicates the topology. On the other hand, unstructured connections can be found at most two cells away from each fault.

1.6.5 Representing a Multilateral Well Path

The procedure is similar to create a grid that conforms to well paths in 3D. We define each branch of the multilateral well shown in Figure 1.24 as a piecewise linear curve, distribute the sites along the resulting line segments, and finally create the background grid. In this example, we consider a multilateral well with four branches (full source code: `mlWellPebi3D.m`). Here, the branches are assumed to be in a single vertical plane to simplify visualization of the resulting grid, but the procedure we present is more general and can easily be changed to handle horizontal well paths. Assuming that the main well path and its branches are given as a cell array `wpath`, we can use the function `lineSites3D` to distribute the sites along these well paths:

```
wGc = @(p) 1/25 / 2 * ones(size(p, 1), 1);
Wc  = lineSites3D(wpath, repmat({wGc}, 1, numel(wpath)));
```

We then distribute a set of equidistant background sites, `bgSites`, in the domain, equivalent to the procedure in the faulted 3D reservoir example in Subsection 1.6.4. However, instead of calling `surfaceSufCond3d` to remove conflicting background sites, we first call an equivalent function for cell-centroid constraints:

```
bgSites = lineSufCond3D(bgSites, Wc);
sites   = [Wc.pts; bgSites];
G       = mirroredPebi3D(sites, bdr);
```

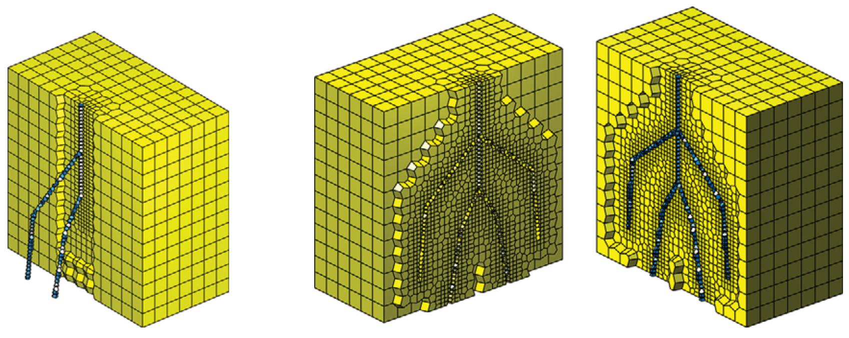


Figure 1.24 Grid conforming to a multilateral well with four branches. The well cells are colored blue and the background cells are colored yellow. (a) A side view of the well with half of the 3D cells cropped away. (b) The grid is opened up to unveil the well. (Source code: `mlWellPebi3D.m`.)

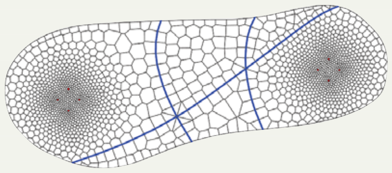
The procedure outlined thus far assumes that individual well paths do not cross each other. To also handle crossing well paths, we simply split two corresponding well curves at each intersection point to obtain a set of noncrossing well segment paths. Along each well segment curve we then place a set of sites. Once again, we cannot guarantee that the centroids of well cells coincide exactly with the prescribed well sites, but by using one of the optimization algorithms for placing the background sites presented in Section 1.3, the centroids and sites will usually be very close.

1.6.6 A More Realistic 3D Case

In the last example, we construct a synthetic but representative 3D grid that has many of the features seen in real models; this particular model is used to simulate geothermal heat storage in Chapter 12. Rock formations of interest typically span several kilometers in the lateral direction but only a few hundred meters in the vertical direction. As discussed in section 3.3 of the MRST textbook, it is quite common to use so-called stratigraphic grids, by first forming a lateral grid that adapts to a two-dimensional description of the reservoir perimeter and major fault lines and then extruding the result in the vertical direction to match with the stratigraphic layering. (Full source code is found in `makeModelIMKGT.m`.)

Our starting point herein is a structure `rp` that contains three sets of points that describe the areal outline of the rock formation, three intersecting faults that each cut through the whole reservoirs, and the positions of two groups of four vertical wells each. Because most of the important dynamics in geothermal heat storage takes place in the near-well zone, we use areal refinement around all wells:

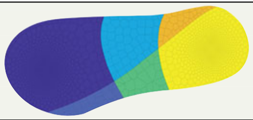
```
% Construct 2D PEBI grid from points rp, with refinement around the wells
[n,L] = deal(25, max(rp.outline));
G2D = pebiGrid2D(max(L)/n, L, ...
      'polybdr'          , rp.outline   , ...
      'faceConstraints'  , rp.faultLines, ...
      'FCFactor'         , 0.8          , ...
      'cellConstraints'  , rp.wellLines , ...
      'CCRefinement'     , true         , ...
      'CCFactor'         , 0.1          , ...
      'interpolateFC'    , true         , ...
      'CCEps'            , 0.08*max(L)  );
G2D = removeShortEdges(G2D, 1);
```



As pointed out in Subsection 1.4.1, PEBI grids constructed by simplex conformity often have a number of short edges. Removing any edges of length shorter than 1.0 reduces the number of edges from 5 345 to 5 271 and the ratio between the longest and the shortest edge from 2 400 to 42. (The grid and numbers will vary slightly between each realization because the initial sites in DistMesh are placed randomly.)

The three faults divide our model into six natural compartments. We classify the cells belonging to each of those compartments using functionality from the `coarsegrid` module (see [11, subsection 14.1.2]):

```
% Classify into compartments
p = ones(G2D.cells.num,1);
p = processPartition(G2D, p, ...
        find(G2D.faces.tag));
```



We then construct a 2.5D grid by extruding the 2D grid in the vertical direction, with a prescribed thickness for each individual of the 23 horizontal grid layers:

```
G0 = makeLayeredGrid(G2D, layerThickness);
```

When creating a conforming (areal) grid, the `pebiGrid2D` function will mark all faces and cells that adapt to constraints using the `faces.tag` and `cells.tag` in the resulting grid structure. The extrusion process does not automatically propagate these tags to the resulting 2.5D grid. So, we do this manually and at the same time generate layer, compartment, and well indicators in each cell:

```
G0.cells.tag  = repmat(G2D.cells.tag, nLayers, 1);    % extrude cell tags
G0.faces.tag  = false(G0.faces.num,1);                % initialize 3D face tag
hf = abs(G0.faces.normals(:,3))<.01;                  % non-vertical faces
G0.faces.tag(hf) = repmat(G2D.faces.tag, nLayers, 1); % extrude 2D face tags
wellNo     = repmat(wellNo2D, nLayers, 1);            % extrude well indicator
layerID    = reshape(repmat(1:nLayers, G2D.cells.num, 1), [], 1);
compartID  = repmat(p,nLayers,1);                     % extrude 2D partition
```

Here, the face tags from the 2D grid have simply been copied to all faces whose normal does not have a significant vertical component.

In the next step, we mimic erosion and geological activity by removing some of the layers in each of the six compartments:

```
bnds = [0 8; 1 7; 1 7; 5 3: 2 6; 5 3];
flag = false(G0.cells.num,1);
for i=1:6
    flag = flag | ((compartID==i) & ...  % add next compartment
        (layerID<=bnds(i,1) | layerID>=(nLayers-bnds(i,2)))); % crop top/bottom
end
[G, indexMap] = removeCells(G0, flag);
```

Here, `indexMap` maps from the indices in the old and new grids, which we use to extract the correct subset of the layer, compartment, and well indicators.

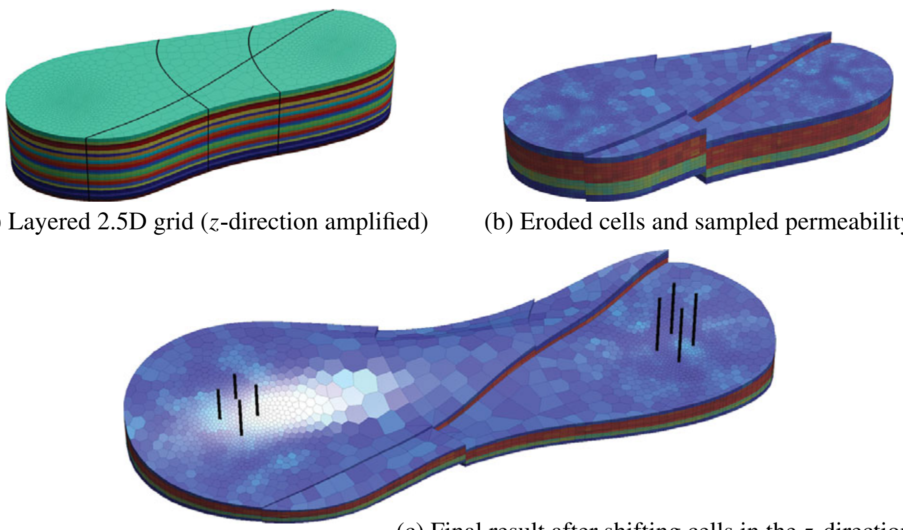
In the second last step, we populate the grid with a layered, lognormal, isotropic permeability field. To model displacement in geological layers across the individual faults, we sample the permeability inside each compartment from the same cube, scaled to fit the bounding box of the compartment:

```
permMean = [10, 912, 790, 90, 10];    % Mean permeability in each layer
N        = [90, 30, nLayers];         % Resolution of cube
ind      = [1,5,13,15,20,nLayers+1]; % Layer indices
K        = reshape(logNormLayers(N, permMean, 'indices', ind), N);
perm     = nan(G.cells.num,1);
for i = 1:6, idx = parts==i;
    perm(idx) = sampleFromBox(G, K, find(idx))*milli*darcy;
end
```

This produces a plausible layering structure but gives different spatial correlation within the individual compartment and does not preserve lateral correlations across faults. However, this crude approach should be sufficient for our purpose of illustration. As a last touch, we shift the vertical coordinates using the built-in MATLAB function `peaks` to mimic the folding/bending often seen in rock formations. Figure 1.25 shows some of the different steps during the construction.

1.7 Concluding Remarks

State-of-the-art methods and scientific development in reservoir modeling often require grids to explicitly include geometric constraints such as bounding faults, depositional environments, lithological contrasts, minor faults, and fractures. Further, to reduce numerical errors and computational time, local grid refinement toward wells and other regions of interest is often necessary. These requirements



(a) Layered 2.5D grid (z-direction amplified) (b) Eroded cells and sampled permeability

(c) Final result after shifting cells in the z-direction

Figure 1.25 Construction of a plausible geomodel with three intersecting faults, layered stratigraphy, and areal near-well refinement. The three plots show key steps following the 2.5D extrusion from a 2D areal description of the reservoir perimeter, the major faults, and the well positions.

put large demands on the grid generation. The `upr` module is developed as a tool to make the gridding easier and extends the basic gridding capabilities included in MRST core. Whereas the main focus of `upr` is on creating PEBI grids that conform to lower-dimensional structures, either as surfaces or as well traces, the module can also be convenient if you wish to grid more complex domains than the unit square. Specifically, the module provides an interface to the DistMesh software that enables you to easily define polygonal boundaries and local refinement toward wells/faults and also implements a linearization that speeds up the necessary distance computations.

The functionality in the `upr` module can be accessed at several different layers. The top layer, given by the functions `pebiGrid2D` and `compositePebiGrid2D`, avoids most of the technicalities of the gridding. These functions are quite flexible and can create almost all of the 2D grids shown in this chapter. The functions have a long list of parameters that can be specified; we have covered the most important herein, but to get the full power of the method you should investigate the options yourself. A good place to start are the tutorials in the example folder of the module. See also the flowchart in Figure 1.26 illustrating the main workflows in `upr`.

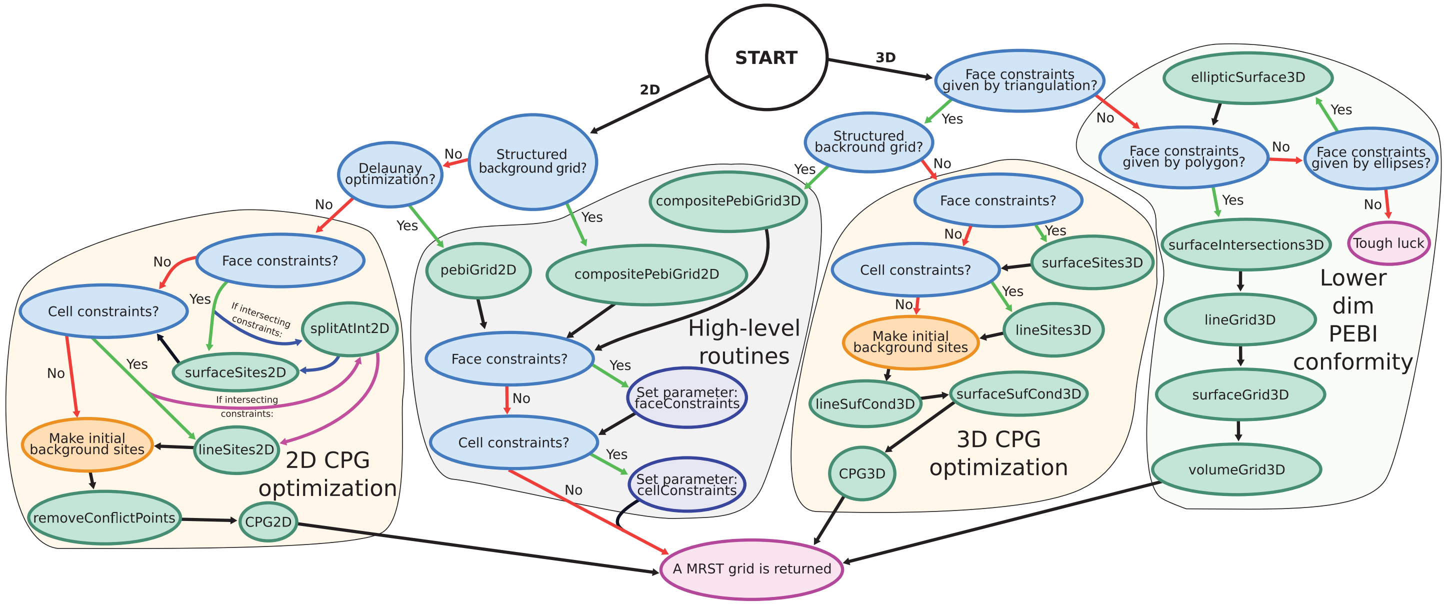


Figure 1.26 Flowchart outlining different workflows you can use to create PEBI grids using the `upr` module.

As mentioned in the Introduction, some of the geometric algorithms in `upr` rely on tolerances that have mainly been tested for grids placed within a bounding box with minimum point at the origin and dimensions of order one. This does not mean that the underlying principles should not work on smaller or larger spatial scales, but this may require adjustment of input parameters and internal tolerances that are not fully exposed to the user. If you encounter strange error messages when trying to create grids with dimensions (and spatial coordinates) that are more typical for oil reservoirs described in map coordinates, the cause will in many cases be absolute tolerances that are too strict and do not capture the increased numerical rounding error for larger domains. If possible, you should therefore strive to scale your constraints and the bounding box to unit size when creating the grid and then rescale the grid to the desired size after it has been created.

We have already pointed out that gridding the statistically distributed fracture network in Subsection 1.6.2 required significant manual effort to find and refine the grid size in challenging areas. Automating and improving the choice of grid sizes to handle such cases would be a very beneficial extension to `upr`. The module can already adapt the grid size at the intersection of constraints, but this is not always sufficient. Specifically, when constraints are close, but not intersecting, the grid size should be decreased based on the distance between the line constraints to avoid conflict between the sites of the different constraints.

Finally, we mention that it is still an open research question how to best formulate efficient and robust algorithms for creating constrained Voronoi grids in 3D. In particular, the 3D methods in `upr` are not as sophisticated as the 2D methods. Grids conforming to simple surfaces can be constructed but, as we have seen, the user must do more of the work. Also, the computational time to convert the grid returned from `voronoin` to an MRST grid structure is significant. In its present state, `upr` mainly targets small- to medium-scale problems and its primary function is to help researchers move from simple grids posed on the unit square to more complex grids that contain many of the conceptual difficulties one expects to see in high-end, high-resolution geomodels.

References

[1] 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. doi: 10.1145/235815.235821.

[2] R. L. Berge, Ø. S. Klemetsdal, and K.-A. Lie. Unstructured Voronoi grids conforming to lower dimensional objects. *Computational Geosciences*, 23(1):169–188, 2019. doi: 10.1007/s10596-018-9790-0.

[3] L. V. Branets, S. S. Ghai, S. L. Lyons, and X.-H. Wu. Challenges and technologies in reservoir modeling. *Communications in Computational Physics*, 6(1):1, 2009.

[4] L. Branets, S. S. Ghai, S. L. Lyons, and X.-H. Wu. Efficient and accurate reservoir modeling using adaptive gridding with global scale up. Paper presented at SPE Reservoir Simulation Symposium, Houston, TX, 2009. doi: 10.2118/118946-MS.

[5] M. A. Christie and M. J. Blunt. Tenth SPE comparative solution project: A comparison of upscaling techniques. *SPE Reservoir Evaluation and Engineering*, 4:308–317, 2001. doi: 10.2118/72469-PA. URL `www.spe.org/web/csp/datasets/set02.htm`.

[6] X. Y. Ding and L. S. K. Fung. An unstructured gridding method for simulating faulted reservoirs populated with complex wells. Paper presented at: SPE Reservoir Simulation Symposium, Houston, TX, 2015. doi: 10.2118/173243-MS.

[7] Q. Du, V. Faber, and M. Gunzburger. Centroidal Voronoi tessellations: Applications and algorithms. *SIAM Review*, 41(4):637–676, 1999. doi: 10.1137/S0036144599352836.

[8] D. D. Filippov, I. Y. Kudryashov, D. Y. Maksimov, D. A. Mitrushkin, B. V. Vasekin, and A. P. Roshchektaev. Reservoir modeling of complex structure reservoirs on dynamic adaptive 3D Pebi-grid. Paper presented at: SPE Russian Petroleum Technology Conference, Moscow, Russia, October 16–18, 2017. doi: 10.2118/187799-MS.

[9] C. Geuzaine and J.-F. Remacle. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009. doi: 10.1002/nme.2579.

[10] M. Iri, K. Murota, and T. Ohya. A fast Voronoi-diagram algorithm with applications to geographical optimization problems. In P. Thoft-Christensen, ed., *System Modelling and Optimization*, pp. 273–288. Springer, Berlin. doi: 10.1007/BFb0008901.

[11] K.-A. Lie. *An Introduction to Reservoir Simulation Using MATLAB/GNU Octave: User Guide for the MATLAB Reservoir Simulation Toolbox (MRST)*. Cambridge University Press, Cambridge, UK, 2019. doi: 10.1017/9781108591416.

[12] Y. Liu, W. Wang, B. Lévy, F. Sun, D.-M. Yan, L. Lu, and C. Yang. On centroidal Voronoi tessellation – energy smoothness and fast computation. *ACM Transactions on Graphics*, 28(4):101:1–101:17, 2009. doi: 10.1145/1559755.1559758.

[13] S. Manzoor, M. G. Edwards, and A. H. Dogru. Three-dimensional geological boundary aligned unstructured grid generation, and CVD-MPFA flow computation. Paper presented at: SPE Reservoir Simulation Conference, Galveston, TX, April 10–11, 2019. doi: 10.2118/193874-MS.

[14] S. Manzoor, M. G. Edwards, A. H. Dogru, and T. M. Al-Shaalan. Interior boundary-aligned unstructured grid generation and cell-centered versus vertex-centered CVD-MPFA performance. *Computational Geosciences*, 22(1):195–230, 2018. doi: 10.1007/s10596-017-9686-4.

[15] 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. doi: 10.1007/s10596-014-9408-0.

[16] P.-O. Persson and G. Strang. A simple mesh generator in MATLAB. *SIAM Review*, 46(2):329–345, 2004. doi: 10.1137/S0036144503429121.

[17] J. R. Shewchuk, S.-W. Cheng, and T. K. Dey. *Delaunay Mesh Generation*. Chapman and Hall/CRC, New York, 2012. doi: 10.1201/b12987.

[18] 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 Reservoir Evaluation & Engineering*, 18(4):463–480, 2015. doi: 10.2118/170703-PA.

[19] D.-M. Yan, W. Wang, B. Lévy, and Y. Liu. Efficient computation of clipped Voronoi diagram for mesh generation. *Computer-Aided Design*, 45(4):843–852, 2013. doi: 10.1016/j.cad.2011.09.004.