3

Grids in Subsurface Modeling

The basic geological description of a petroleum reservoir or an aquifer system will typically consist of two sets of surfaces. Geological horizons are lateral surfaces describing the bedding planes that delimit the rock strata, whereas faults are vertical or inclined surfaces along which the strata may have been displaced by geological processes. In this chapter, we discuss how to turn the basic geological description into a discrete model that can be used to formulate various computational methods, e.g., for solving the equations that describe fluid flow.

A *grid* is a tessellation of a planar or volumetric object by a set of contiguous simple shapes referred to as *cells*. Grids can be described and distinguished by their *geometry*, which gives the shape of the cells that form the grid, and their *topology* that tells how the individual cells are connected. In 2D, a cell is in general a closed polygon for which the geometry is defined by a set of *vertices* and a set of *edges* that connect pairs of vertices and define the interface between two neighboring cells. In 3D, a cell is a closed polyhedron for which the geometry is defined by a set of vertices, a set of edges that connect pairs of vertices, and a set of *faces* (surfaces delimited by a subset of the edges) that define the interface between two different cells; see Figure 3.1. Herein, we assume that all cells are nonoverlapping, so that each point in the planar/volumetric object represented by the grid is either inside a single cell, lies on an interface or edge, or is a vertex. Two cells that share a common face are said to be connected. Likewise, one can also define connections based on edges and vertices. The topology of a grid is defined by the total set of connections, which is sometimes also called the *connectivity* of the grid.

When implementing grids in modeling software, one always has the choice between generality and efficiency. To represent an arbitrary grid, it is necessary to explicitly store the geometry of each cell in terms of vertices, edges, and faces, as well as storing the connectivity among cells, faces, edges, and vertices. However, as we will see later, huge simplifications can be made for particular classes of grids by exploiting regularity in the geometry and structures in the topology. Consider, for instance, a planar grid consisting of rectangular cells of equal size. Here, the topology can be represented by two indices, and one only needs to specify a reference point and the two side lengths of the rectangle to describe the geometry. This way, one ensures minimal memory usage and optimal efficiency when accessing the grid. On the other hand, exploiting the simplified description

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

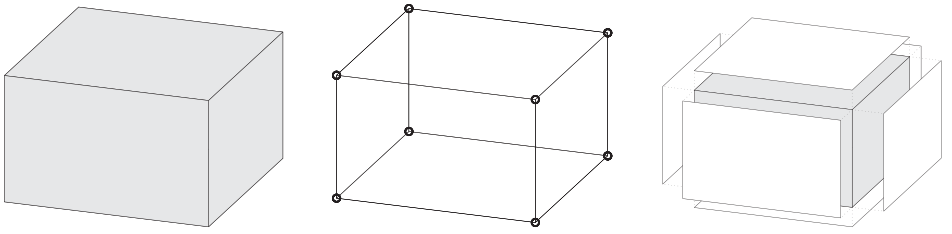


Figure 3.1 Illustration of a single cell, vertices and edges, and cell faces.

explicitly in your flow or transport solver inevitably means that the solver must be reimplemented if you later decide to use another grid format.

The most important goal for our development of MRST is to provide a toolbox that both allows *and* enables the use of various grid types. To avoid having a large number of different, and potentially incompatible, grid representations, we have therefore chosen to store *all grid types* using a general unstructured format in which cells, faces, vertices, and connections between cells and faces are explicitly represented. This means that we, for the sake of generality, have sacrificed some of the efficiency one can obtain by exploiting special structures in a particular grid type, and instead have focused on obtaining a flexible grid description that is not overly inefficient. Moreover, our grid structure can be extended by other properties that are required by various discretization schemes for flow and transport simulations. A particular discretization may need the *volume* or the *centroid* (grid-point, midpoint, or generating point) of each cell. Likewise, for cell faces one may need to know the face areas, the face normals, and the face centroids. Although these properties can be computed from the geometry (and topology) of the grid, it is often useful to precompute and include them explicitly in the grid representation.

The first third of this chapter is devoted to standard grid formats that are available in MRST. We introduce examples of structured grids, including regular Cartesian, rectilinear, and curvilinear grids, and briefly discuss unstructured grids, including Delaunay triangulations and Voronoi grids. The purpose of our discussion is to demonstrate the basic grid functionality in MRST and show some key principles you can use to implement new structured and unstructured grid formats. In the second part of the chapter, we discuss industry-standard formats for stratigraphic grids that are based on extrusion of 2D shapes (corner-point, prismatic, and 2.5D PEBI grids). Although these grids have an inherent logical structure, representation of faults, erosion, pinch-outs, and so on lead to cells that can have quite irregular shapes and an (almost) arbitrary number of faces. In the last part of the chapter, we discuss how the grids introduced in the first two parts of the chapter can be partitioned to form flexible coarse descriptions that preserve the geometry of the underlying fine grids. The ability to represent a wide range of grids, structured or unstructured on the fine and/or coarse scale, is a strength of MRST compared to the majority of research codes arising from academic institutions.

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

A number of videos that complement the material presented in this chapter can be found in the second MRST Jolt [189]. This Jolt introduces different types of grids, discusses how such grids can be represented, and outlines functionality you can use to generate your own grids.

3.1 Structured Grids

As we saw in this chapter's introduction, a grid is a tessellation of a planar or volumetric object by a set of simple shapes. In a *structured* grid, only one basic shape is allowed and this basic shape is laid out in a regular repeating pattern so that the topology of the grid is constant in space. The most typical structured grids are based on quadrilaterals in 2D and hexahedrons in 3D, but in principle it is also possible to construct grids with a fixed topology using certain other shapes. Structured grids can be generalized to so-called multiblock grids (or hybrid grids), in which each block consists of basic shapes that are laid out in a regular repeating pattern.

Regular Cartesian Grids

The simplest form of a structured grid consists of unit squares in 2D and unit cubes in 3D, so that all vertices in the grid are integer points. More generally, a regular Cartesian grid can be defined as consisting of congruent rectangles in 2D and rectilinear parallelepipeds in 3D, etc. Hence, the vertices have coordinates $(i_1\Delta x_1, i_2\Delta x_2, \dots)$ and the cells can be referenced using the multi-index $(i_1, i_2, \dots)$. Herein, we only consider finite Cartesian grids that consist of a finite number $n_2 \times n_2 \times \cdots \times n_k$ of cells that cover a bounded domain $[0, L_1] \times [0, L_2] \times \cdots \times [0, L_k]$.

Regular Cartesian grids can be represented very compactly by storing n_i and L_i for each dimension. In MRST, however, Cartesian grids are represented as if they were fully unstructured using a general grid structure that will be described in more detail in Section 3.4. Cartesian grids therefore have special constructors,

```
G = cartGrid([nx, ny], [Lx Ly]);
G = cartGrid([nx, ny, nz], [Lx Ly Lz]);
```

that set up the data structures representing the basic geometry and topology of the grid. The second argument is optional.

Rectilinear Grids

A rectilinear grid (also called a tensor grid) consists of rectilinear shapes (rectangles or parallelepipeds) that are not necessarily congruent to each other. In other words, whereas a regular Cartesian grid has a uniform spacing between its vertices, the grid spacing can vary along the coordinate directions in a rectilinear grid. The cells can still be referenced using a multi-index $(i_1, i_2, \dots)$, but the mapping from indices to vertex coordinates is nonuniform.

In MRST, one can construct a rectilinear grid by specifying vectors with the grid vertices along the coordinate directions:

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

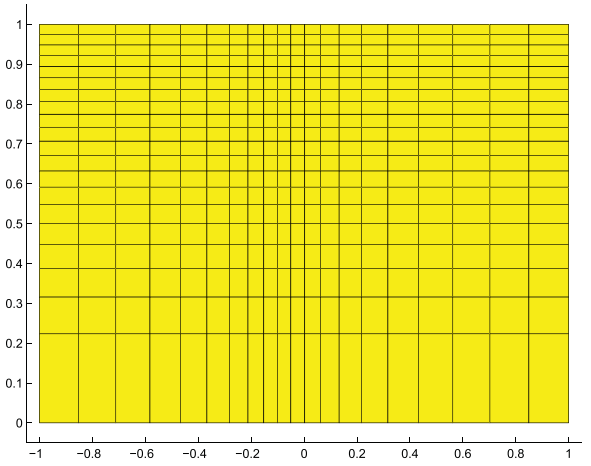


Figure 3.2 Example of a rectilinear grid.

```
G = tensorGrid(x, y);
G = tensorGrid(x, y, z);
```

This syntax is the same as for the MATLAB functions `meshgrid` and `ndgrid`.

As an example of a rectilinear grid, we construct a 2D grid that covers the domain $[-1, 1] \times [0, 1]$ and is graded toward $x = 0$ and $y = 1$ as shown in Figure 3.2.

```
dx = 1-0.5*cos((-1:0.1:1)*pi);
x = -1.15+0.1*cumsum(dx);
y = 0:0.05:1;
G = tensorGrid(x, sqrt(y));
plotGrid(G); axis([-1.05 1.05 -0.05 1.05]);
```

Curvilinear Grids

A curvilinear grid is a grid with the same topological structure as a regular Cartesian grid, but in which the cells are quadrilaterals rather than rectangles in 2D and cuboids rather than parallelepipeds in 3D. The grid is given by the coordinates of the vertices, but there exists a mapping that will transform the curvilinear grid to a uniform Cartesian grid so that each cell can still be referenced using a multi-index $(i_1, i_2, \dots)$.

For the time being, MRST has no constructor for curvilinear grids. Instead, you can create such grids by first instantiating a regular Cartesian grid or a rectilinear grid and then manipulating the vertices. This method is quite simple as long as there is a one-to-one mapping between the curvilinear grid in physical space and the logically Cartesian grid in reference space. The method will *not* work if the mapping is not one-to-one so that vertices with different indices coincide in physical space. In this case, the user should create an

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

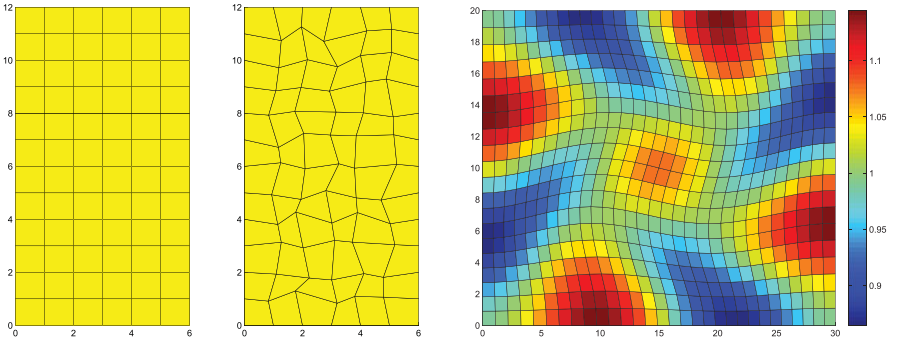


Figure 3.3 The middle plot shows a rough grid created by perturbing all internal nodes of the regular 6×12 Cartesian grid in the left plot. The right plot shows a curvilinear grid created using the function `twister` that uses a combination of sin functions to perturb a rectilinear grid. The color is determined by the cell volumes.

ECLIPSE input file [270, 33] with keywords `COORD[XYZ]` (see Section 3.3.1) and use the function `buildCoordGrid` to create the grid.

To illustrate the discussion, we show two examples of how to create curvilinear grids. In the first example, we create a rough grid by perturbing all internal nodes of a regular Cartesian grid (see Figure 3.3):

```
nx = 6; ny=12;
G = cartGrid([nx, ny]);
subplot(1,2,1); plotGrid(G);
c = G.nodes.coords;
I = any(c==0,2) | any(c(:,1)==nx,2) | any(c(:,2)==ny,2);
G.nodes.coords(~I,:) = c(~I,:) + 0.6*rand(sum(~I),2)-0.3;
subplot(1,2,2); plotGrid(G);
```

In the second example, we use the routine `twister` to perturb the internal vertices. The function maps the grid back to the unit square, perturbs the vertices according to the mapping

$$(x_i, y_i) \mapsto \big(x_i + f(x_i, y_i), y_i - f(x_i, y_i)\big), \quad f(x, y) = 0.03 \sin(\pi x) \sin\big(3\pi(y - \tfrac{1}{2})\big),$$

and then maps the grid back to its original domain. The right plot of Figure 3.3 shows the resulting grid. To illuminate the effect of the mapping, we have colored the cells according to their volume, which has been computed using the function `computeGeometry`, which we will come back to later in the chapter.

```
G = cartGrid([30, 20]);
G.nodes.coords = twister(G.nodes.coords);
G = computeGeometry(G);
plotCellData(G, G.cells.volumes, 'EdgeColor', 'k'), colorbar
```

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

Fictitious Domains

One obvious drawback with Cartesian and rectilinear grids, as just defined, is that they can only represent rectangular domains in 2D and cubic domains in 3D. Curvilinear grids, on the other hand, can represent more general shapes by introducing an appropriate mapping, and can be used in combination with rectangular/cubic grids in multiblock grids for efficient representation of realistic reservoir geometries. However, finding a mapping that conforms to a given boundary is often difficult, in particular for complex geologies, and using a mapping in the interior of the domain will inadvertently lead to cells with rough geometries that deviate far from being rectilinear. Such cells may in turn introduce problems if the grid is to be used in a subsequent numerical discretization, as we will see later.

As an alternative, complex geometries can be easily modeled using structured grids by a so-called fictitious domain method. In this method, the complex domain is embedded into a larger "fictitious" domain of simple shape (a rectangle or cube) using, e.g., a Boolean indicator value in each cell to tell whether the cell is part of the domain or not. The observant reader will notice that we already have encountered the use of this technique for the SAIGUP dataset (Figure 2.16) and the Johansen dataset in Chapter 2. In some cases, one can also adapt the structured grid by moving the nearest vertices to the domain boundary.

MRST has support for fictitious domain methods through the function `removeCells`, which we will demonstrate in the next example, where we create a regular Cartesian grid that fills the volume of an ellipsoid:

```
x = linspace(-2,2,21);
G = tensorGrid(x,x,x);
subplot(1,2,1); plotGrid(G);view(3); axis equal

subplot(1,2,2); plotGrid(G,'FaceColor','none');
G = computeGeometry(G);
c = G.cells.centroids;
r = c(:,1).^2 + 0.25*c(:,2).^2+0.25*c(:,3).^2;
G = removeCells(G, r>1);
plotGrid(G); view(-70,70); axis equal;
```

Worth observing here is the use of `computeGeometry` to compute cell centroids that are not part of the basic geometry representation in MRST. Figure 3.4 shows plots of the grid before and after removing the inactive parts. Because of the fully unstructured representation used in MRST, calling `computeGeometry` actually removes the inactive cells from the grid structure, but from the outside, the structure behaves as if we had used a fictitious domain method.

You can find more examples of how you can make structured grids and populate them with petrophysical properties in the fourth video of the second MRST Jolt [189].

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

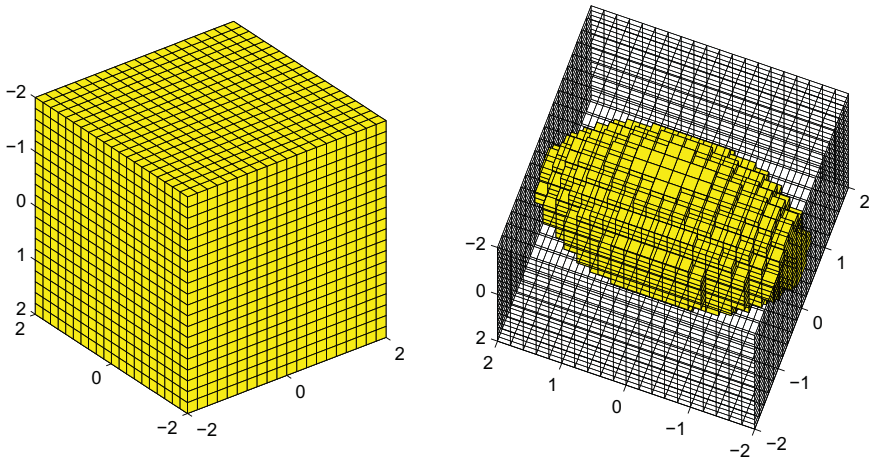
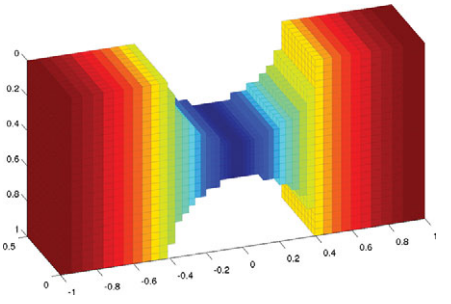


Figure 3.4 Example of a regular Cartesian grid representing a domain in the form of an ellipsoid. The underlying logical Cartesian grid is shown in the left plot and as a wireframe in the right plot. The active part of the model is shown in yellow color in the right plot.

Computer exercises

3.1.1 Make the grid shown:



Hint: the grid spacing in the x-direction is given by $\Delta x(1 - \frac{1}{2}\cos(\pi x))$ and the colors signify cell volumes.

3.1.2 Metaballs are commonly used in computer graphics to generate organic-looking objects. Each metaball is defined as a smooth function that has finite support. One example is

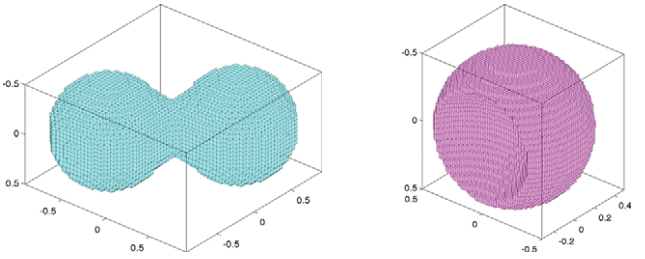
$$m(\vec{x}, r) = \left[1 - \min\left(\frac{|\vec{x}|^2}{r^2}, 1\right)\right]^4.$$

Metaballs can be used to define objects implicitly, e.g., as all the points $\vec{x}$ that satisfy

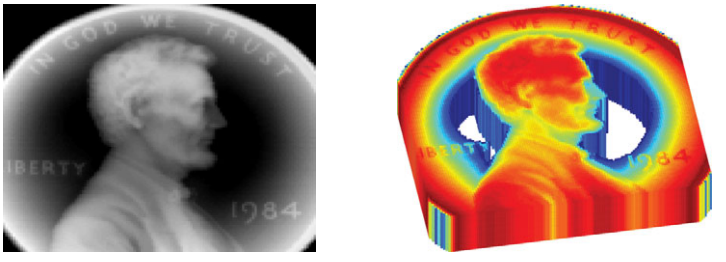
$$\sum_i m(\vec{x} - \vec{x}_i, r_i) \leq C, \qquad C \in \mathbb{R}^+$$

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

Use this approach and try to make grids similar to the ones shown here:



3.1.3 A simple way to make test models with funny geometries is to use the method of fictitious domains and let an image define the domain of interest. In the figures, the image was taken from `penny`, which is one of the standard data sets that are distributed with MATLAB, and then used to define the geometry of the grid and assign permeability values



Pick your own favorite image or make one in a drawing program and use `imread` to load the image into MATLAB as a 3D array, which you can use to define your geometry and petrophysical values. If you do not have an image at hand, you can use `penny` or `spine`. For `penny`, in particular, you may have to experiment a bit with the threshold used to define your domain to ensure that all cells are connected, i.e., that the grid you obtain consists of only one piece.

3.2 Unstructured Grids

An unstructured grid consists of a set of simple shapes that are laid out in an irregular pattern so that any number of cells can meet at a single vertex. The topology of the grid will therefore change throughout space. An unstructured grid can generally consist of a combination of polyhedral cells with varying numbers of faces, as we will see in this section. However, the most common forms of unstructured grids are based on triangles in 2D and tetrahedrons in 3D. These grids are very flexible and are relatively easy to adapt to complex domains and structures, or refine to provide increased local resolution.

Unlike structured grids, unstructured grids cannot generally be efficiently referenced using a structured multi-index. Instead, one must describe a list of connectivities that specifies the way a given set of vertices make up individual cells and cell faces, and how these cells are connected to each other via faces, edges, and vertices.

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

To understand the properties and construction of unstructured grids, we start by a brief discussion of two concepts from computational geometry: Delaunay tessellation and Voronoi diagrams. Both these concepts are supported by standard functionality in MATLAB.

3.2.1 Delaunay Tessellation

A tessellation of a set of *generating points* $\mathcal{P} = \{x_i\}_{i=1}^n$ is defined as a set of simplices that completely fills the convex hull of $\mathcal{P}$. The *convex hull* H of $\mathcal{P}$ is the convex minimal set that contains $\mathcal{P}$ and can be described constructively as the set of convex combinations of a finite subset of points from $\mathcal{P}$,

$$H(\mathcal{P}) = \left\{ \sum_{i=1}^{\ell} \lambda_i x_i \;\middle|\; x_i \in \mathcal{P}, \lambda_i \in \mathbb{R}, \lambda_i \geq 0, \sum_{i=1}^{\ell} \lambda_i = 1, 1 \leq \ell \leq n \right\}.$$

Delaunay tessellation is by far the most common method of generating a tessellation based on a set of generating points. In 2D, the Delaunay tessellation consists of a set of triangles defined so that three points form the corners of a Delaunay triangle only when the *circumcircle* that passes through them contains no other generating points; see Figure 3.5. The definition using circumcircles can readily be generalized to higher dimensions using simplices and hyperspheres.

The center of the circumcircle is called the *circumcenter* of the triangle. We will come back to this quantity when discussing Voronoi diagrams in the next subsection. When four (or more) points lie on the same circle, the Delaunay triangulation is not unique. As an example, consider four points defining a rectangle. Using either of the two diagonals will give two triangles satisfying the Delaunay condition.

The Delaunay triangulation can alternatively be defined using the so-called *max-min angle criterion*, which states that the Delaunay triangulation is the one that maximizes the minimum angle of all angles in a triangulation; see Figure 3.6. Likewise, the Delaunay triangulation minimizes the largest circumcircle and minimizes the largest min-containment circle, which is the smallest circle that contains a given triangle. Additionally, the closest two generating points are connected by an edge of a Delaunay triangulation. This is called

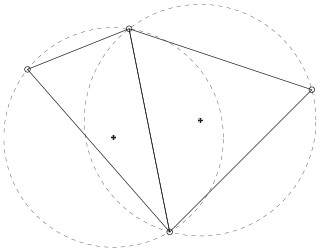


Figure 3.5 Two triangles and their circumcircles.

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

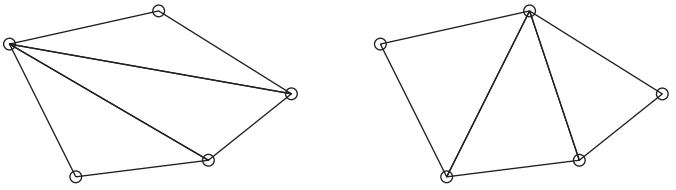


Figure 3.6 Example of two triangulations of the same five points; the triangulation to the right satisfies the min-max criterion.

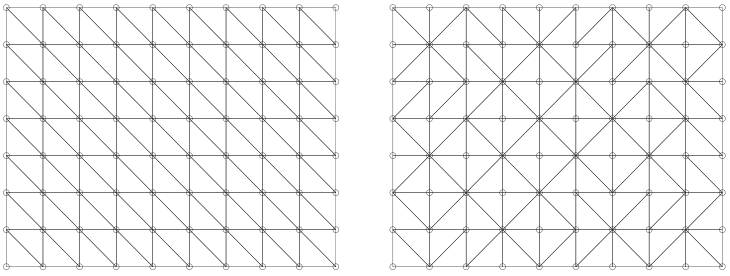


Figure 3.7 Two different Delaunay tessellations of a rectangular point mesh.

the *closest-pair property*, and the two neighboring points are referred to as natural neighbors. This way, the Delaunay triangulation can be seen as the natural tessellation of a set of generating points.

Delaunay tessellation is a popular research topic and there exists a large body of literature on theoretical aspects and computer algorithms. Likewise, there are a large number of software implementations available on the Internet. For this reason, MRST does not have any routines for generating tessellations based on simplexes. Instead, we have provided simple routines for mapping a set of points and edges, as generated by MATLAB's Delaunay triangulation routines, to the internal data structure used to represent grids in MRST. How these routines work will be illustrated in terms of a few simple examples.

In the first example, we triangulate a rectangular mesh and then convert the result by use of the MRST routine `triangleGrid`:

```
[x,y] = meshgrid(1:10,1:8);
t = delaunay(x(:),y(:));
G = triangleGrid([x(:) y(:)],t);
plot(x(:),y(:),'o','MarkerSize',8);
plotGrid(G,'FaceColor','none');
```

Depending on what version you have of MATLAB, the 2D Delaunay routine `delaunay` will produce one of the triangulations shown in Figure 3.7. In older versions of MATLAB, the implementation of `delaunay` was based on Qhull (see www.qhull.org), which produces the unstructured triangulation shown in the right plot. MATLAB 7.9 and newer has

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

improved routines for 2D and 3D computational geometry, and here `delaunay` will produce the structured triangulation shown in the left plot. However, the n-D tessellation routine `delaunayn([x(:)y(:)])` is still based on Qhull and will generally produce an unstructured tessellation, as shown in the right plot.

If the set of generating points is structured, e.g., as one would obtain by calling either `meshgrid` or `ndgrid`, it is straightforward to make a structured triangulation. The following skeleton of a function makes a 2D triangulation and can easily be extended by the interested reader to 3D:

```
function t = mesh2tri(n,m)
[I,J]=ndgrid(1:n-1, 1:m-1); p1=sub2ind([n,m],I(:),J(:));
[I,J]=ndgrid(2:n  , 1:m-1); p2=sub2ind([n,m],I(:),J(:));
[I,J]=ndgrid(1:n-1, 2:m  ); p3=sub2ind([n,m],I(:),J(:));
[I,J]=ndgrid(2:n  , 1:m-1); p4=sub2ind([n,m],I(:),J(:));
[I,J]=ndgrid(2:n  , 2:m  ); p5=sub2ind([n,m],I(:),J(:));
[I,J]=ndgrid(1:n-1, 2:m  ); p6=sub2ind([n,m],I(:),J(:));
t = [p1 p2 p3; p4 p5 p6];
```

In Figure 3.8, we have used the demo case `seamount` supplied with MATLAB as an example of a more complex unstructured grid

```
load seamount;
plot(x(:),y(:),'o');
G = triangleGrid([x(:) y(:)]);
plotGrid(G,'FaceColor',[.8 .8 .8]); axis off;
```

The observant reader will notice that we did not explicitly generate a triangulation before calling `triangleGrid`; if the second argument is omitted, the routine uses MATLAB's built-in `delaunay` triangulation as default.

For 3D grids, MRST supplies a conversion routine `tetrahedralGrid(P, T)` that constructs a valid grid definition from a set of points `P` ($m \times 3$ array of node coordinates)

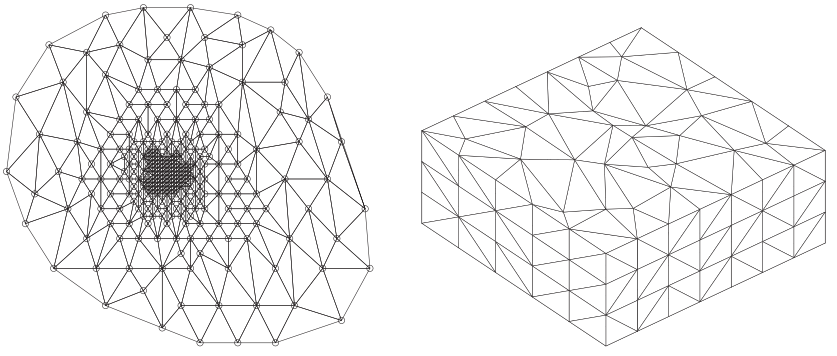


Figure 3.8 The left plot shows the triangular grid from the `seamount` demo case. The right plot shows a tetrahedral tessellation of a 3D point mesh.

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

and a tetrahedron list T (n array of node indices). The tetrahedral tessellation shown to the right in Figure 3.8 was constructed from a set of generating points defined by perturbing a regular hexahedral mesh:

```
N=7; M=5; K=3;
[x,y,z] = ndgrid(0:N,0:M,0:K);
x(2:N,2:M,:) = x(2:N,2:M,:) + 0.3*randn(N-1,M-1,K+1);
y(2:N,2:M,:) = y(2:N,2:M,:) + 0.3*randn(N-1,M-1,K+1);
G = tetrahedralGrid([x(:) y(:) z(:)]);
plotGrid(G, 'FaceColor',[.8 .8 .8]); view(-40,60); axis tight off
```

3.2.2 Voronoi Diagrams

The Voronoi diagram of a set of points $\mathcal{P} = \{x_i\}_{i=1}^n$ is the partitioning of Euclidean space into n (possibly unbounded) convex polytopes[1] such that each polytope contains exactly one generating point x_i and every point inside the given polytope is closer to its generating point than any other point in $\mathcal{P}$. The convex polytopes are called Voronoi cells or Voronoi regions. Mathematically, the *Voronoi region* $V(x_i)$ of a generating point x_i in $\mathcal{P}$ can be defined as

$$V(x_i) = \left\{ x \;\middle|\; \|x - x_i\| < \|x - x_j\| \; \forall j \neq i \right\}. \tag{3.1}$$

A Voronoi region is not closed in the sense that a point that is equally close to two or more generating points does not belong to the region defined by (3.1). Instead, these points are said to lie on the Voronoi segments and can be included in the Voronoi cells by defining the closure of $V(x_i)$, using "$\leq$" rather than "$<$" in (3.1).

The Voronoi regions for all generating points lying at the convex hull of $\mathcal{P}$ are unbounded, all other Voronoi regions are bounded. For each pair of two points x_i and x_j, one can define a hyperplane with co-dimension one consisting of all points that lie equally close to x_i and x_j. This hyperplane is the perpendicular bisector to the line segment between x_i and x_j and passes through the midpoint of the line segment. The Voronoi diagram of a set of points can be derived directly as the *dual* of the Delaunay triangulation of the same points. To understand this, we consider the planar case; see Figure 3.9. For every triangle, there is a polyhedron in which the vertices occupy complementary locations:

- The circumcenter of a Delaunay triangle corresponds to a vertex of a Voronoi cell.
- Each vertex in the Delaunay triangulation corresponds to, and is the center of, a Voronoi cell.

Moreover, for locally orthogonal Voronoi diagrams, an edge in the Delaunay triangulation corresponds to a segment in the Voronoi diagram and the two intersect each other orthogonally. However, as we can see in Figure 3.9, this is not always the case. If the

[1] A polytope is a generic term that refers to a polygon in 2D, a polyhedron in 3D, and so on.

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

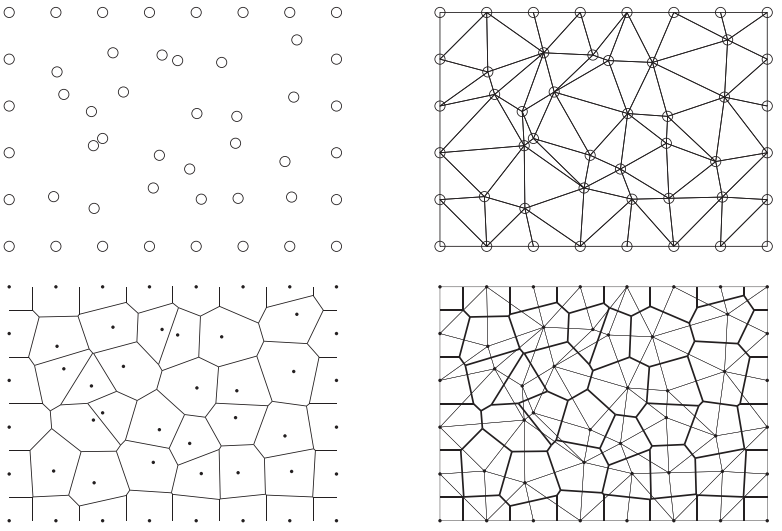


Figure 3.9 Duality between Voronoi diagrams and Delaunay triangulation. From top left across to bottom right: generating points, Delaunay triangulation, Voronoi diagram, and Voronoi diagram (thick lines) and Delaunay triangulation (thin lines).

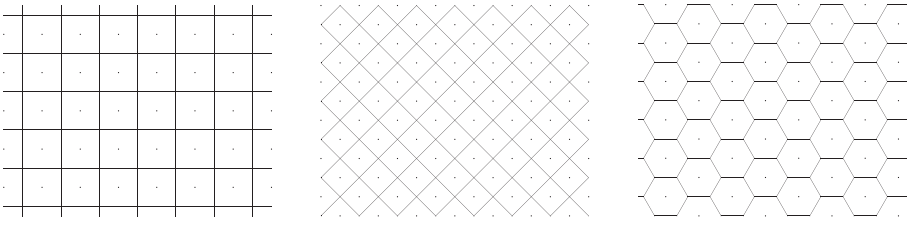


Figure 3.10 Three examples of Voronoi diagrams generated from 2D point lattices. From left to right: square lattice, square lattice rotated 45 degrees, and lattice forming equilateral triangles.

circumcenter of a triangle lies outside the triangle itself, the Voronoi segment does not intersect the corresponding Delaunay edge. To avoid this situation, one can perform a *constrained Delaunay triangulation* and insert additional points where the constraint is not met (i.e., the circumcenter is outside its triangle).

Figure 3.10 shows three examples of planar Voronoi diagrams generated from 2D point lattices using the MATLAB function `voronoi`. MRST does not yet have a similar function that generates a Voronoi grid from a point set, but offers `V=pebi(T)` that generates a locally orthogonal, Voronoi grid `V` as a dual to a triangular grid `T`. The grids are constructed by connecting the perpendicular bisectors of the edges of the Delaunay triangulation, hence the name perpendicular bisector (PEBI) grids. To demonstrate the functionality, we first generate a honeycombed grid similar to the one shown in the right plot in Figure 3.10:

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

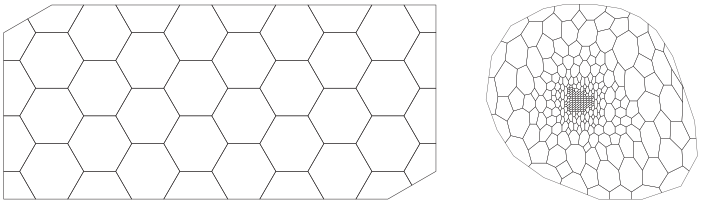


Figure 3.11 Two examples of Voronoi grids. The left plot shows a honeycombed PEBI grid and the right plot shows the PEBI grid derived from the `seamount` demo case.

```
[x,y] = meshgrid([0:4]*2*cos(pi/6),0:3);
x = [x(:); x(:)+cos(pi/6)];
y = [y(:); y(:)+sin(pi/6)];
G = triangleGrid([x,y]);
plotGrid(pebi(G), 'FaceColor','none'); axis equal off
```

Figure 3.11 shows the resulting grid. As a second example, we reiterate the `seamount` examples shown in Figure 3.8:

```
load seamount
V = pebi(triangleGrid([x y]));
plotGrid(V,'FaceColor',[.8 .8 .8]); axis off;
```

Several of the examples discussed in this section can also be found in the fifth video of the second MRST Jolt [189]. Since `pebi` is a 2D code, we cannot apply it directly to the 3D tetrahedral grid shown in Figure 3.8 to generate a dual 3D Voronoi grid. The `upr` module [42, 169] offers several more advanced tools for generating 2D and 3D Voronoi grids; we will return to this in Section 3.5.3. Likewise, Section 3.3.2 discusses how 2D Voronoi grids can be extruded to 3D giving so-called 2.5D grid; this is a standard approach in geological modeling to preserve geological layers.

3.2.3 General Tessellations

Tessellations come in many other forms than the Delaunay and Voronoi types discussed in the two previous sections. Tessellations are more commonly referred to as *tilings* and are patterns made up of geometric forms (tiles) that are repeated over and over without overlapping or leaving any gaps. Such tilings can be found in many patterns of nature, like in honeycombs, giraffe skin, pineapples, snake skin, tortoise shells, to name a few. Tessellations have also been extensively used for artistic purposes since ancient times, from the decorative tiles of Ancient Rome and Islamic art to the amazing artwork of M. C. Esher. For completeness (and fun), MRST offers the function `tessellationGrid` that can take a tessellation consisting of symmetric n-polygonals and turn it into a correct grid structure. While this may not be very useful in modeling petroleum reservoirs, it can

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

easily be used to generate irregular grids that can be used to stress test various discretization methods. To exemplify, the following commands will make a standard $n \times m$ Cartesian mesh:

```
[x,y] = meshgrid(linspace(0,1,n+1),linspace(0,1,m+1));
I = reshape(1:(n+1)*(m+1),m+1,n+1);
T = [reshape(I(1:end-1,1:end-1),[],1)'; reshape(I(1:end-1,2:end  ),[],1)';
     reshape(I(2:end,  2:end  ),[],1)'; reshape(I(2:end,  1:end-1),[],1)']';
G = tessellationGrid([x(:) y(:)], T);
```

Here, the vertices and cells are numbered first in the y direction and then in the x-direction, so that the first two lines in `T` read `[1 m+2 m+3 2; 2 m+3 m+4 3]`, and so on. There are obviously many ways to make more general tilings. The script `showTessellation` in the `book` module shows some simple but amazing examples.

3.2.4 Using an External Mesh Generator

Using a Delaunay triangulation as discussed earlier, we can generate grids that fit a given set of vertices. However, in most applications, the vertex points are not given a priori and all one wants is a reasonable grid that fits an exterior outline and possibly also a set of interior boundaries and/or features, which in the case of subsurface media could be faults, fractures, or geological horizons. To this end, you will typically have to use a mesh generator. There are a large number of mesh generators available online, and in principle any of these can be used in combination with MRST's grid factory routines, as long as they produce triangulations on the form outlined earlier. The `triangle` module implements a simple MATLAB interface to the `Triangle` generator (www.cs.cmu.edu/~quake/triangle.html).

My personal favorite, however, is `DistMesh` by Persson and Strang [248]. While most mesh generators tend to be complex and quite inaccessible codes, `DistMesh` is a relatively short and simple MATLAB code written in the same spirit as MRST. The performance of the code may not be optimal, but the user can go in and inspect all algorithms and modify them to his or her purpose. A slightly modified version of `DistMesh` is, for instance, used on the inside of the `upr` module for generating high-quality Voronoi grids, which will be briefly outlined in Section 3.5.3. In the following, we use `DistMesh` to generate a few examples of more complex triangular and Voronoi grids in 2D.

`DistMesh` is distributed under the GNU GPL license (which is the same license that MRST uses) and can be downloaded from the software's website. The simplest way to integrate `DistMesh` with MRST is to install it as a third-party module. Assuming that you are connected to Internet, this is done as follows:

```
pth = fullfile(ROOTDIR,'utils','3rdparty','distmesh');
mkdir(pth)
unzip('http://persson.berkeley.edu/distmesh/distmesh.zip', pth);
mrstPth('reregister','distmesh', pth);
```

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

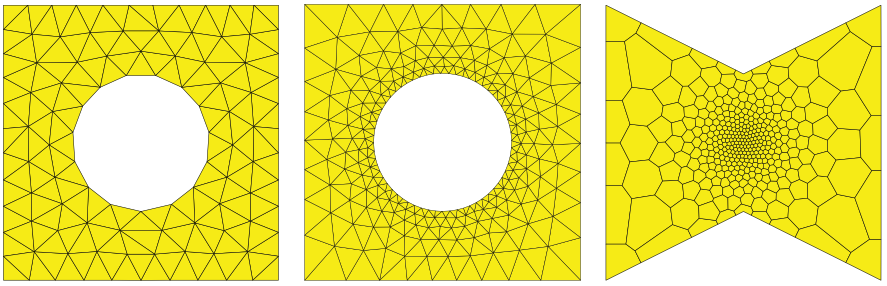


Figure 3.12 Grids generated by the `DistMesh` grid generator.

and you are ready to use `DistMesh`. If you intend to use it many times, you should copy the last line to the `startup_user.m` file in MRST's root directory.

In `DistMesh`, the perimeter of the domain is represented using a signed distance function $d(x, y)$, which by definition is set to be negative inside the region. The software offers a number of utility functions that makes it simple to describe relatively complex geometries, as we shall see in the following. Let us start with a simple example taken from the `DistMesh` website: Consider a square domain $[-1, 1] \times [-1, 1]$ with a circular cutout of radius 0.5 centered at the origin. We start by making a grid that has a uniform target size $h = 0.2$

```
mrstModule add distmesh;
fd=@(p) ddiff(drectangle(p,-1,1,-1,1), dcircle(p,0,0,0.5));
[p,t]=distmesh2d(fd, @huniform, 0.2, [-1,-1;1,1], [-1,-1;-1,1;1,-1;1,1]);
G = triangleGrid(p, t);
```

Here, we have used utility functions `ddif`, `drectangle` and `dcircle` to compute the signed distance from the outer and inner perimeter. Likewise, `huniform` is set to enforce uniform cell size and equal distance between the initial points, here this distance is 0.2. The fourth argument is the bounding box of our domain, and the fifth argument consists of fixed points that the algorithm is not allowed to move. After the triangulation has been computed by `distmesh2d`, we pass it to `triangleGrid` to make an MRST grid structure. The resulting grid is shown to the left in Figure 3.12.

As a second test, let us make a graded grid that has a mesh size of approximately 0.05 at the inner circle and 0.2–0.35 at the outer perimeter. To enforce this, we replace the `huniform` function by another function that gives the correct mesh size distribution:

```
fh=@(p) 0.05+0.3*dcircle(p,0,0,0.5);
[p,t]=distmesh2d(fd, fh, 0.05,[-1,-1;1,1], [-1,-1;-1,1;1,-1;1,1]);
```

The middle plot of Figure 3.12 shows that the resulting grid has the expected grading from the inner boundary and outwards to the perimeter.

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

In our last example, we create a graded triangulation of a polygonal domain and then use `pebi` to compute its Voronoi diagram

```
pv = [-1 -1; 0 -.5; 1 -1; 1 1; 0 .5; -1 1; -1 -1];
fh = @(p,x) 0.025 + 0.375*sum(p.^2,2);
[p,t] = distmesh2d(@dpoly, fh, 0.025, [-1 -1; 1 1], pv, pv);
G = pebi(triangleGrid(p, t));
```

Here, we use the utility function `dpoly(p,pv)` to compute the signed distance of any point set `p` to the polygon with vertices in `pv`. Notice that `pv` must form a closed path. Likewise, since the signed distance function and the grid density function are assumed to take the same number of arguments, `fh` is created with a dummy argument `x`. The argument sent to these functions is passed as the sixth argument to `distmesh2d`. The resulting grid is shown to the right in Figure 3.12.

`DistMesh` also has routines for creating nD triangulations and triangulations of surfaces, but these are beyond the scope of the current presentation.

Computer exercises

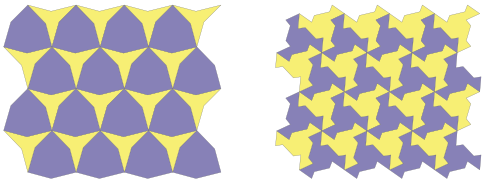
3.2.1 Create MRST grids from the standard data sets `trimesh2d` and `tetmesh`. How would you assign lognormal petrophysical parameters to these grids so that the spatial correlation is preserved?

3.2.2 All grids generated by `triangleGrid` are assumed to be planar so that each vertex can be given by a 2D coordinate. However, triangular grids are commonly used to represent nonplanar surfaces in 3D. Can you extend the function so it can construct both 2D and 3D grids? You can use the data set `trimesh3d` as an example of a triangulated 3D surface.

3.2.3 Load the `triangle` module and use it to mesh the polygonal outline to the right in Figure 3.12. Try to also make other convex and concave outlines.

3.2.4 Download and install `distmesh` and try to make MRST grids from all the triangulations shown in [248, Fig. 5.1].

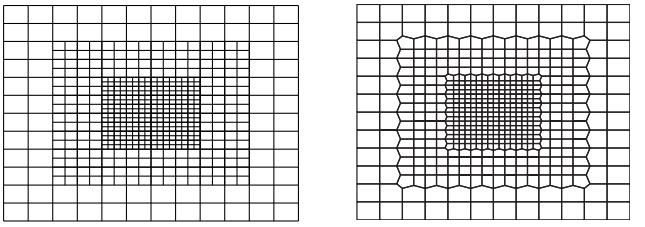
3.2.5 Running the tutorial `showTessellation` from the `book` module will produce figures like these:



What would you do to fit the tessellations so that they fill a rectangular box without leaving any gaps along the border?

3.2.6 MRST does not yet have a grid factory routine to generate structured grids with local, nested refinement, as shown in the figure to the left:

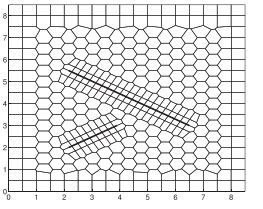
https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press



Try to use a combination of `triangleGrid` and `pebi` to make a good approximation to such a grid as shown to the right in the figure. (Hint: to get rid of artifacts, one layer of cells were removed along the outer boundary.)

3.2.7 The figure shows an unstructured hexagonal grid that has been adapted to two faults in the interior of the domain and padded with rectangular cells near the boundary. Try to implement a routine that generates a similar grid.

Hint: in this case, the strike direction of the faults are $\pm 30^\circ$ and the start and endpoints of the faults have been adjusted so that they coincide with the generating points of hexagonal cells.



3.3 Stratigraphic Grids

When discussing how to model reservoir rocks, we saw that the grid is closely connected to the description of the petrophysical characteristics of a reservoir. Not only does the grid describe the outer geometry of the reservoir, but it is also used to represent internal structures such as fractures and faults, as well as surfaces representing marked changes the rock's physical characteristics (*lithography*). This means that the grid is closely related to the parameter description of the flow model and, unlike in many other disciplines, cannot be chosen independently to provide a certain numerical accuracy. Indeed, the grid is typically chosen by a geologist who tries to describe the rock body by as few volumetric cells as possible and who does not necessarily care too much about potential numerical difficulties the choice of grid may cause in subsequent flow simulations. Although grossly simplified, this observation is important to bear in mind throughout the rest of this chapter.

The industry standard to represent reservoir geology in a flow simulator is to use a *stratigraphic grid* built based on geological horizons and fault surfaces. The volumetric grid is typically built by extruding 2D tessellations of the geological horizons in the vertical direction or in a direction following major fault surfaces. For this reason, some stratigraphic grids, like the PEBI grids you will meet in Section 3.3.2, are often called 2.5D rather than 3D grids. These grids may be unstructured in the lateral direction, but have a clear structure in the vertical direction to reflect the layering of the reservoir.

Because of the role grid models play in representing geological formations, real-life stratigraphic grids tend to be highly complex and have unstructured connections induced

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

by the displacements that have occured over faults. Another characteristic feature is high aspect ratios. Typical reservoirs extend several hundred or thousand meters in the lateral direction, but the zones carrying hydrocarbon may be just a few tens of meters in the vertical direction and consist of several layers with (largely) different rock properties. Getting the stratigraphy correct is crucial, and high-resolution geological modeling will typically result in a high number of (very) thin grid layers in the vertical direction, resulting in two or three orders of magnitude aspect ratios.

A full exposition of stratigraphic grids is far beyond the scope of this book. In the next two subsections, we discuss the basics of the two most commonly used forms of stratigraphic grids. A complementary discussion is given in videos 2, 6, and 7 of the second MRST Jolt [189].

3.3.1 Corner-Point Grids

To model the geological structures of petroleum reservoirs, the industry-standard approach is to introduce what is called a corner-point grid [254], which we already encountered in Section 2.5. A corner-point grid consists of a set of hexahedral cells that are topologically aligned in a Cartesian fashion so that the cells can be numbered using a logical ijk index. In its simplest form, a corner-point grid is specified in terms of a set of vertical or inclined *pillars* defined over an areal Cartesian 2D mesh in the lateral direction. Each cell in the volumetric grid is defined by eight *logical* corner points specified as two *depth coordinates* on each of the four *coordinate lines* that define a pillar; see Figure 3.13. The grid consists of $n_x \times n_y \times n_z$ grid cells and the cells are ordered with the i-index (x-axis) cycling fastest, then the j-index (y-axis), and finally the k-index (negative z-direction). All cellwise property data are assumed to follow the same numbering scheme.

As discussed previously, a fictitious domain approach is used to embed the reservoir in a logically Cartesian shoebox. This means that inactive cells that are not part of the physical model, e.g., as shown in Figure 2.16, are present in the topological ijk-numbering but are

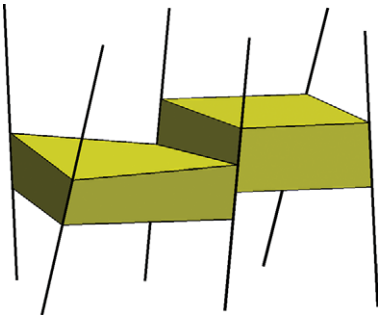


Figure 3.13 Each cell in the corner-point grid is restricted by two points on each of the four grid lines that make up a pillar cells. The plot shows cells and grid lines from two neighboring pillars.

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

indicated by a zero porosity or net-to-gross value, as discussed in Section 2.4, or marked by a special boolean indicator (called `ACTNUM` in the input files).

So far, the topology and geometry of a corner-point grid have not deviated from that of the mapped Cartesian grids studied in the previous section. Somewhat simplified, one may view the logical ijk numbering as a reflection of the sedimentary rock bodies as they may have been deposited in geological time, so that cells with the same k index have been deposited at approximately the same time. To model geological features like erosion and pinch-outs of geological layers, the corner-point format allows point-pairs to collapse along coordinate lines. This creates degenerate hexahedral cells that may have less than six faces, as illustrated in Figure 3.14. The corner points can even collapse along all four lines of a pillar, so that a cell completely disappears. This implicitly introduces a new local topology, which is sometimes referred to as *non-neighboring connections*, in which cells that are not logical ijk neighbors can be neighbors in physical space and share a common face. Figure 3.15 shows an example of a model that contains both eroded geological layers and fully collapsed cells. In a similar manner, (simple) vertical and inclined faults can be easily modeled by aligning the pillars with fault surfaces and displacing the corner points defining neighboring cells on one or both sides of the fault. This way, one creates nonmatching geometries and non-neighboring connections in the underlying ijk topology.

To illustrate the concepts introduced so far, we consider a low-resolution version of the model from Figure 2.11 on page 42 created by the `simpleGrdecl` routine, which generates an input stream containing the basic keywords that describe a corner-point grid in the ECLIPSE input deck [270, 33]

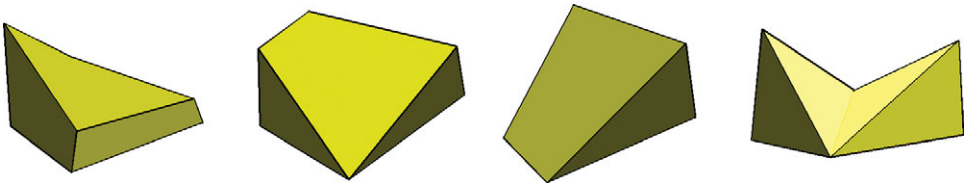


Figure 3.14 Examples of deformed and degenerate hexahedral cells arising in corner-point grid models.

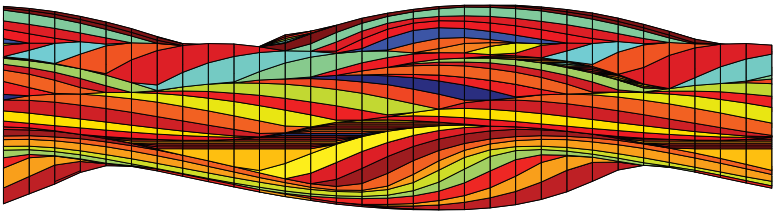


Figure 3.15 Side view in the xz-plane of corner-point grid with vertical pillars modeling a stack of sedimentary beds (each layer indicated by a different color).

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

```
grdecl = simpleGrdecl([4, 2, 3], .12, 'flat', true);
```

```
grdecl =
    cartDims: [4 2 3]
       COORD: [90x1 double]
       ZCORN: [192x1 double]
      ACTNUM: [24x1 int32]
```

The COORD field contains the pairs of 3D coordinates for the 15 coordinate lines that define the 4×2 mesh of pillars, whereas the ZCORN field gives the z-values that determine vertical positions uniquely along each coordinate line for the eight corner-points of the 24 cells. To extract these data, we use two MRST routines

```
[X,Y,Z] = buildCornerPtPillars(grdecl,'Scale',true);
[x,y,z] = buildCornerPtNodes(grdecl);
```

We now plot the coordinate lines and the corner-points and mark those lines on which the corner-points of logical ij neighbors do not coincide,

```
% Plot pillars
plot3(X',Y',Z','k');
set(gca,'zdir','reverse'), view(35,35), axis off, zoom(1.2);

% Plot points on coordinate lines, mark those with faults red
hold on; I=[3 8 13];
hpr = plot3(X(I,:)',Y(I,:)',Z(I,:)','r','LineWidth',2);
hpt = plot3(x(:),y(:),z(:),'o'); hold off;
```

From the plots in the upper row of Figure 3.16 we clearly see how the coordinate lines change slope from the east and west side toward the fault in the middle, and how the grid points sit like beads on a string along each coordinate line.

Cells along a pillar are now defined by connecting pairs of points from four neighboring coordinate lines that make up a cell in the areal mesh. To see this, we plot two pillars as well as the whole grid with the fault surface marked in blue:

```
% Create grid and plot two stacks of cells
G = processGRDECL(grdecl);
args = {'FaceColor', 'r', 'EdgeColor', 'k'};
hcst = plotGrid(G,[1:8:24 7:8:24],'FaceAlpha', .1, args{:});

% Plot cells and fault surface
delete([hpt; hpr; hcst]);
plotGrid(G,'FaceAlpha', .15, args{:});
plotFaces(G, G.faces.tag>0,'FaceColor','b','FaceAlpha',.4);
```

The upper-left plot in Figure 3.17 shows the same model sampled with even fewer cells. We have used different coloring of the cell faces on each side of the fault to highlight nonmatching faces along the fault plane. MRST represents corner-point grids as *matching*

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

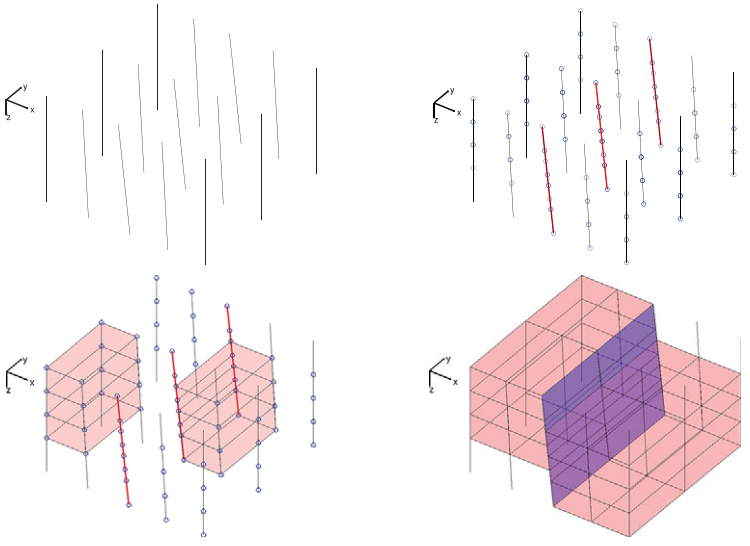


Figure 3.16 Specification of a corner-point grid. Starting from the coordinate lines defining pillars (upper left), we add corner-points and identify lines containing nonmatching corner marked in red (upper right). A stack of cells is created for each set of four lines defining a pillar (lower left), and then the full grid is obtained (lower right). In the last plot, the fault faces have been marked in blue.

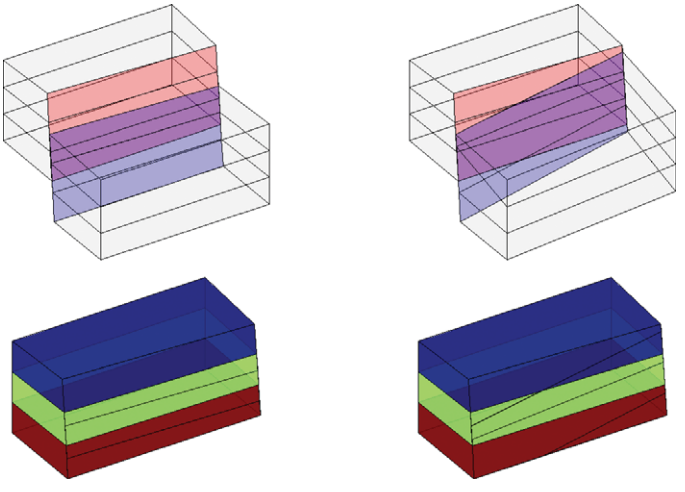


Figure 3.17 Subdivision of fault face in two three-dimensional models. In the left column, the subfaces are all rectangular. In the right columns they are not. In both the upper plots, the faces marked in red belong only to the cells behind the fault surface, the blue faces belong only to the cells in front of the fault surface, and the magenta ones belong to cells on both sides. The lower plot shows the cells behind the surface, where each cell has been given its own color.

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

unstructured grids obtained by subdividing all nonmatching cell faces instead of using the more compact nonmatching hexahedral form. This means that the four cells with non-neighboring connections across the fault plane will have seven and not six faces. For each such cell, two of the seven faces lie along the fault plane. Here, the subdivision resulted in new faces that all have four corners and rectangular geometry. This is generally not the case; the right column of Figure 3.17 shows a grid where we can see cells with six, seven, or eight faces, and faces with three, four, and five corners. Indeed, for real-life models, subdivision of nonmatching fault faces can lead to cells having much more than six faces.

Using the inherent flexibility of the corner-point format, it is possible to construct very complex geological models that come a long way in matching the geologist's perception of the underlying rock formations. Because of their many appealing features, corner-point grids have been an industry standard for years, and the format is supported in most commercial software for reservoir modeling and simulation.

A Simple Grid Generator

The internal and external geology of a real reservoir is usually describe by a large set of horizons and fault surfaces. As explained earlier, horizons are lateral surfaces that deliminate the reservoir in the upward and downward direction and represent the main stratigraphic layers. These surfaces can either be triangulated/tessellated or represented using splines. Major fault surfaces are typically vertical or inclined, but need to be planar. Generating corner-point grids that adapt to these surfaces is a challenging task and requires specialized geomodeling software and integration of a lot of different data types. MRST does not offer any such capabilities, and complex geomodels must therefore be created using external software. However, we do offer a relatively simple grid generator that takes a set of horizons as input and generates a corner-point grid by interpolating vertically between pairs of consecutive horizons. The function `converHorizonsToGrid` cannot model faults, and assumes that each horizon is represented over a rectilinear areal mesh. The horizons need not use the same areal mesh points, but the areal meshes must have a certain minimal overlap inside which the corner-point will be generated. The following example illustrates the basic use of the grid generator:

```
[y,x,z]  = peaks(15); z = z+8;
horizons = {struct('x',x,'y',y,'z',z),struct('x',x,'y',y,'z',z+8)};
grdecl   = convertHorizonsToGrid(horizons,'layers', 4);
G        = processGRDECL(grdecl);
```

Here, we use the `peaks` function from MATLAB to generate a surface and create two horizons that deliminate the reservoir upward and downward. We then create a grid with four layers of cells in the vertical direction. When no extra parameters are specified, the areal mesh defining the corner-point grid has the same number of points as the horizons; see Figure 3.18. When the areal domain of definition for the horizons do not coincide, the interpolation region is set to be the minimum rectangle that contains the areal bounding boxes of all the horizons. Inside this rectangle, the routine creates a Cartesian mesh and

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

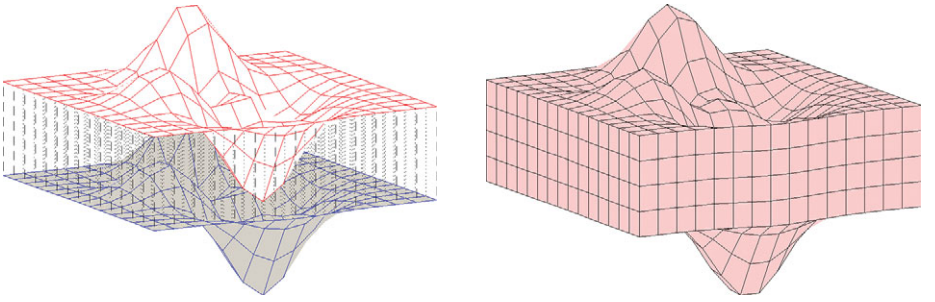


Figure 3.18 Example of a corner-point grid generated by interpolating vertically between two geological horizons.

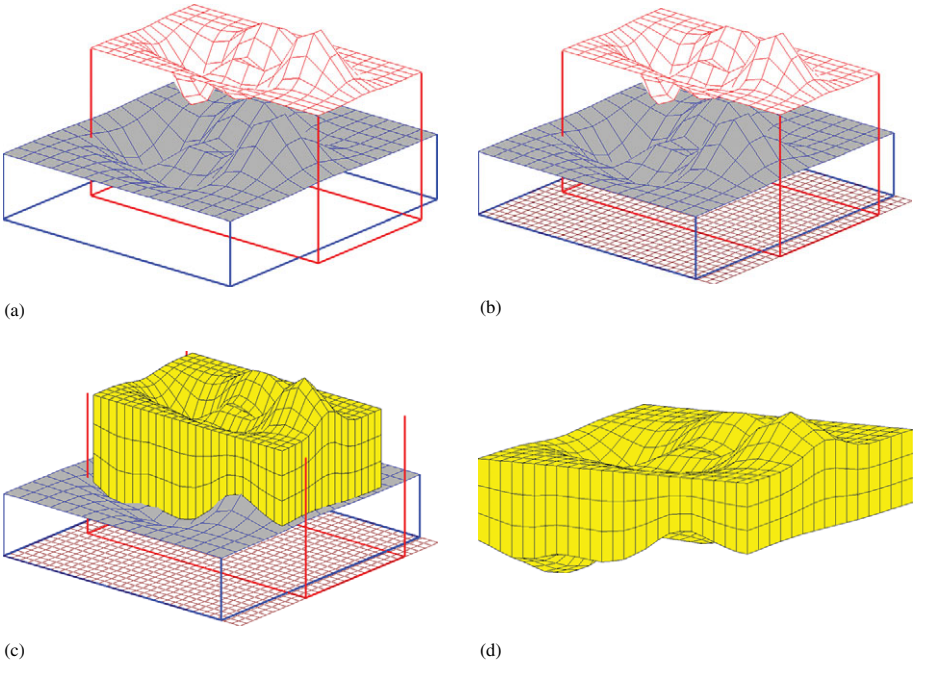


Figure 3.19 Example of a corner-point grid generated by interpolating vertically between two partially overlapping horizons. (a) Horizons and projected bounding boxes. (b) Interpolation region with 31×31 areal mesh covers the two bounding boxes. (c) Corner-point grid generated inside the maximum overlap of the two horizons. (d) The full 3D grid generated by the routine.

uses this to interpolate vertically. In the output grid, the routine discards all cells whose lateral projection is not contained inside the maximum rectangle that fits inside the areal bounding boxes of all horizons. Figure 3.19 shows illustrates how the grid resulting from the following commands is created:

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

```
[n,m]    = deal(30);
horizons = {struct('x',x,'y',y,'z',z),struct('x',x+.5,'y',2*y+1,'z',z+10)};
grdecl   = convertHorizonsToGrid(horizons,'dims',[n m], 'layers', 3);
G        = processGRDECL(grdecl);
```

Here, we have only used two horizons to simplify the illustration. Section 15.6.4 presents a more complex case with multiple horizons.

A Synthetic Faulted Reservoir

In our second example we consider a synthetic model of two intersecting faults that make up the letter Y in the lateral direction. The two fault surfaces are highly deviated, making an angle far from 90 degrees with the horizontal direction. To model this scenario using corner-point grids, we basically have two different choices. The first choice, which is quite common, is to let the coordinate lines (and hence the extrusion direction) follow the main fault surfaces. For highly deviated faults, like in the current case, this will lead to extruded cells that are far from K-orthogonal and hence susceptible to grid-orientation errors in a subsequent simulation, as will be discussed in more detail in Chapter 6. Alternatively, we can choose a vertical extrusion direction and replace deviated fault surfaces by *stair-stepped* approximations so that the faults zigzag in a direction not aligned with the grid. This will create cells that are mostly K-orthogonal and less prone to grid-orientation errors, but will only represent the fault approximately.

Figure 3.20 shows two different grid models, taken from the `CaseB4` data set. In the stair-stepped model, the use of cells with orthogonal faces causes the faults to be represented as zigzag patterns. The pillar grid correctly represents the faults as inclined planes, but has cells with degenerate geometries and cells that deviate strongly from being orthogonal in the lateral direction. Likewise, some coordinate lines have close to 45 degrees inclination, which will likely give significant grid-orientation effects in a standard two-point scheme.

A Simulation Model of the Norne Field

Norne is an oil and gas field lies located in the Norwegian Sea. The reservoir is found in Jurassic sandstone at a depth of 2,500 meters below sea level, and was originally estimated to contain 90.8 million m^3 oil, mainly in the Ile and Tofte formations, and 12 billion m^3 gas in the Garn formation. The field is operated by Statoil and production started in November 1997, using a floating production, storage, and offloading (FPSO) ship connected to seven subsea templates at a water depth of 380 meters. The oil is produced with water injection as the main drive mechanisms, and the expected ultimate oil recovery is more than 60%, which is very high for a subsea oil reservoir. During thirteen years of production, five 4D seismic surveys of high quality have been recorded. In cooperation with NTNU, operator Statoil and partners have released large amounts of subsurface data from the Norne field for

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

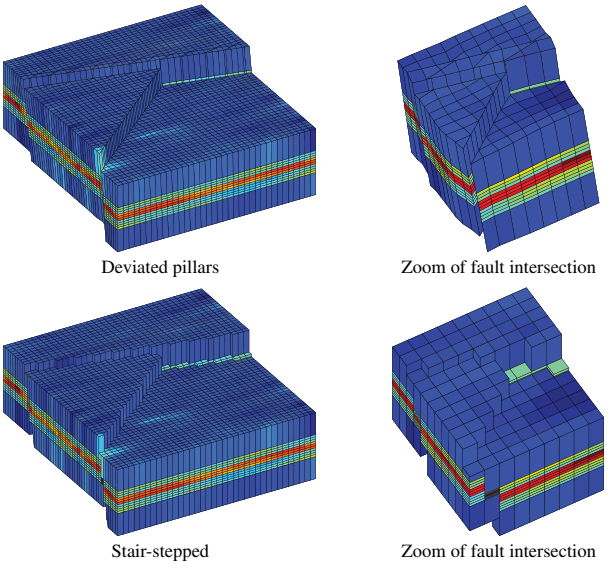


Figure 3.20 Modeling the intersection of two deviated faults using deviated pillars (top) and stair-stepped approximation (bottom). CaseB4 grids courtesy of Statoil.

research and education purposes.[2] More recently, the Open Porous Media (OPM) initiative (opm-project.org) released the full simulation model as an open data set on Github (github.com/OPM/opm-data). For several years, Norne was the only full simulation model of a real oil reservoir that was freely available to the general public. Recently, Equinor (former Statoil) released a much more comprehensive data set of the Volve field. The Norne model can either be downloaded and installed using `mrstDatasetGUI` or directly from the command line:

```
makeNorneSubsetAvailable() && makeNorneGRDECL()
```

Once in place, we can load the grid and the petrophysical data from the simulation model as follows:

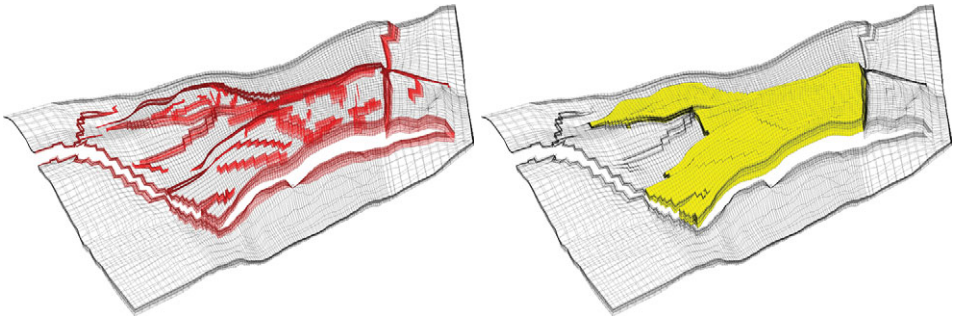
```
grdecl = readGRDECL(fullfile(getDatasetPath('norne'), 'NORNE.GRDECL'));
grdecl = convertInputUnits(grdecl, getUnitSystem('METRIC'));
```

The model consists of $46 \times 112 \times 22$ corner-point cells. We start by plotting the whole model, including inactive cells. To this end, we need to override[3] the `ACTNUM` field before

[2] The Norne Benchmark data sets are hosted and supported by the Center for Integrated Operations in the Petroleum Industry (IO Center) at NTNU (www.ipt.ntnu.no/~norne/). The data set used herein was first released as part of "Package 2: Full field model" (2013)

[3] At this point I hasten to warn you that inactive cells often contain junk data and may generally not be inspected in this manner. Here, however, most inactive cells are defined in a reasonable way. By not performing basic sanity checks on the resulting grid (option `'checkgrid'=false`), we manage to process the grid and produce reasonable graphical output. In general, however, I advise that `'checkgrid'` remains set in its default state of `true`.

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press



```
plotGrid(G,'FaceColor','none','EdgeAlpha',.1);
plotFaces(G, G.faces.tag>0, ...
   'FaceColor','red','FaceAlpha',.2, ...
   'EdgeColor','r','EdgeAlpha',.1);
axis off; view(-155,80); zoom(1.7);
```

```
plotGrid(G, ~actnum(G.cells.indexMap), ...
   'FaceColor','none','EdgeAlpha',.1);
plotGrid(G, actnum(G.cells.indexMap), ...
   'FaceColor','y', 'EdgeAlpha',.1);
axis off; view(-155,80); zoom(1.7);
```

Figure 3.21 The Norne field from the Norwegian Sea. The plots show the whole grid with fault faces marked in red (left) and active cells marked in yellow (right).

we start processing the input, because if the ACTNUM flag is set, all inactive cells will be ignored when the unstructured grid is built

```
actnum = grdecl.ACTNUM;
grdecl.ACTNUM = ones(prod(grdecl.cartDims),1);
G  = processGRDECL(grdecl, 'checkgrid', false);
```

Having processed and converted the grid to the correct unstructured format, we first plot the outline of the whole model and highlight all faults and the active part of the model; see Figure 3.21. During the processing, all fault faces are tagged with a positive number. This can be utilized to highlight the faults: we simply find all faces with a positive tag, and color them with a specific color as shown in the left box in the figure. To continue with the active model only, we reset the ACTNUM field to its original values so that inactive cells are ignored when we process the ECLIPSE input stream [270, 33]. This gives a grid with two components: the main reservoir and a detached stack of twelve cells that should be ignored.

To examine some parts of the model in more detail, we can use the function cutGrdecl to extract a rectangular box in index space from the ECLIPSE input stream, e.g., as follows:

```
cut_grdecl = cutGrdecl(grdecl, [6 15; 80 100; 1 22]);
g = processGRDECL(cut_grdecl);
```

Figure 3.22 shows a zoom on four different regions. The first region (red color), is sampled near a laterally stair-stepped fault, i.e., a curved fault surface that has been approximated by a surface that zigzags in the lateral direction. We also notice how the fault

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

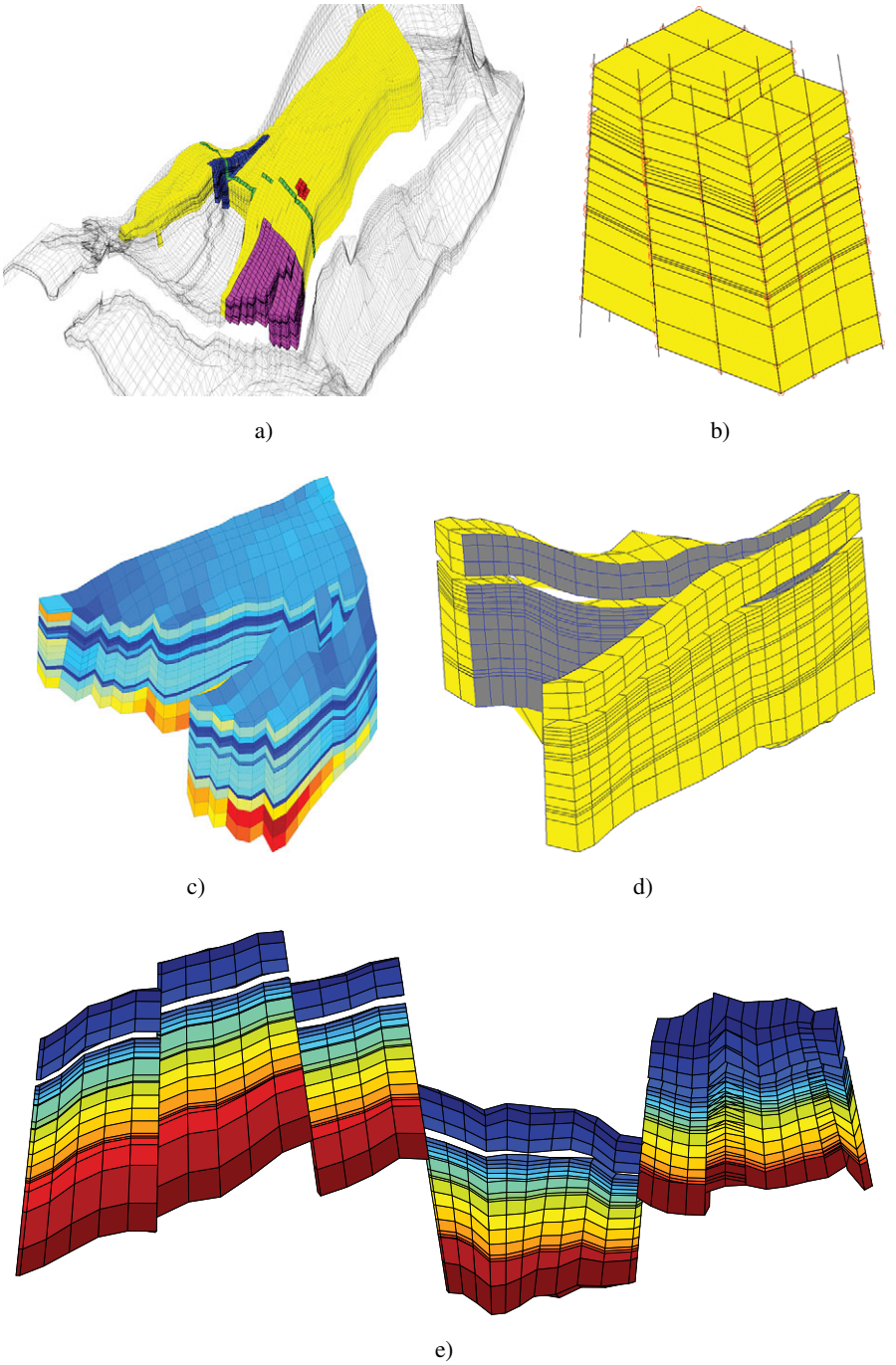


Figure 3.22 Detailed view of subsets from the Norne simulation model. a) The whole model with active and inactive cells and four regions of interest marked in different colors. b) Zoom of the red region with pillars and corner-points shown as red circles. c) The magenta region with coloring according to cell volumes, which vary by a factor 700. d) The blue region in which fault faces have been colored gray and the corresponding coordinate lines have been colored blue. e) The green cross-section with coloring according to layer number from top to bottom.

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

displacement leads to cells that are nonmatching across the fault surface, and the presence of some very thin layers (the thinnest layers may actually appear as thick lines in the plot). The thin layers are also clearly seen in the second region (magenta color), which represents a somewhat larger sample from an area near the tip of one of the "fingers" in the model. Here, we clearly see how similar layers have been strongly displaced across the fault zone. In the third (blue) region, we have colored the fault faces to clearly show the displacement and the hole through the model in the vertical direction, which likely corresponds to a shale layer that has been eliminated from the active model. Gaps and holes, and displacement along fault faces, are even more evident for the vertical cross section (green region) for which the layers have been given different colors, as in Figure 3.15. Altogether, the four views of the model demonstrate typical patterns that you can see in realistic models. We return to this simulation model in Section 10.3.8.

Extensions, Difficulties, and Challenges

The original corner-point format has been extended in several ways, e.g., to enable vertical intersection of two coordinate lines in the shape of the letter Y. Coordinate lines may also be defined by piecewise polynomial curves, resulting in what is sometimes called S-faulted grids. Likewise, two neighboring coordinate lines can collapse so that the basic grid shape becomes a prism rather than a hexahedron. However, there are several features you cannot easily model, like multiple fault intersections (e.g., as in the letter F), and the industry is thus constantly in search for improved griding methods. One example will be discussed in the next subsection. First, however, we will discuss some difficulties and challenges, seen from the side of a computational scientist seeking to use corner-point grids for his or her computations.

The flexible cell geometry of the corner-point format poses several challenges for numerical implementations. Indeed, a geocellular grid is typically chosen by a geologist who tries to describe the rock body by as few volumetric cells as possible and who may not be aware of potential numerical difficulties his or her choice of geometries and topologies may cause in subsequent flow simulations.

Writing a robust grid-processing algorithm to compute geometry and topology or to determine an equivalent matching, polyhedral grid proved to be much more challenging than we anticipated when we started developing MRST. Displacements across faults will lead to geometrically complex, nonconforming grids, e.g., as illustrated in Figure 3.22. Since each face of a grid cell is specified by four (arbitrary) points, the cell interfaces in the grid will generally be bilinear and possibly strongly curved surfaces. Geometrically, this can lead to several complications. Cell faces on different sides of a fault may intersect each other so that cells overlap volumetrically, or they may leave void spaces in between them. There may be tiny overlap areas between cell faces on different sides a fault, and so on. All these factors contribute to make fault geometries hard to interpret in a geometrically consistent way: a subdivision into triangles is, for instance, not unique. Likewise, top and bottom surfaces may intersect for highly curved cells with high aspect ratios, cell centroids may be outside the cell volume, etc.

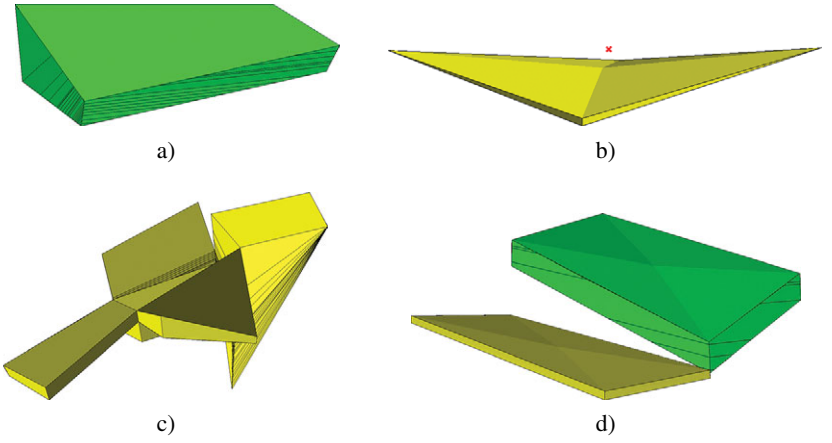


Figure 3.23 Illustration of difficulties associated with real-life corner-point geometries. a) Many faces resulting from subdivision to give matching grid at faults. b) A curved $800 \times 800 \times 0.25$ m cell, whose centroid lies outside the cell. c) Difficult geometries. d) Small interface between two cells.

The presence of collapsed corner-points implies that the grid cells will generally be polyhedral and possibly contain both triangular and bilinear faces (see Figure 3.14). Likewise, individual cells can have many neighbors; the Norne model has cells with up to 20 neighbors. Altogether, this calls for flexible discretizations that work on unstructured topologies and are not sensitive to the geometry of each cell or the number of faces and corner points. Simple discretization methods that only account for pressure differences between cells sharing a common face (see Section 4.4.1 are fairly easy to implement for general unstructured grids, but more advanced discretization methods that rely on the use of dual grids or reference elements can be quite challenging to implement on highly irregular grids. Figure 3.23 illustrates some geometrical and topological challenges seen in standard grid models.

Cell geometries will often deviate significantly from being orthogonal to adapt to complex horizons and fault surfaces. This may introduce unphysical preferential flow directions in the numerical methods (as we will see later). Stratigraphic grids will also typically have aspect ratios that are two or three orders of magnitude, which can introduce severe numerical difficulties because the majority of the flow in and out of a cell occurs across the faces with the smallest area. Similarly, the possible presence of strong heterogeneities and anisotropies in the permeability fields, e.g., as seen in the SPE 10 example in Chapter 2, typically introduces large condition numbers in the discretized flow equations, which make them hard to solve.

Corner-point grids generated by geological modeling typically contain too many cells. Once created by the geologist, the grid is handed to a reservoir engineer, whose first job is to reduce the number of cells if he or she is to have any hope of getting the model through a simulator. The generation of good coarse grids for use in upscaling, and the upscaling procedure itself, is generally work-intensive, error-prone, and not always sufficiently robust, as we will come back to in Chapter 15.

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

3.3.2 2.5D Unstructured Grids

So-called 2.5D grids have been designed to combine the advantages of two different griding methods: the (areal) flexibility of unstructured grids and the simple topology of Cartesian grids in the vertical direction. These grids are constructed in much the same way as corner-point grids, except that coordinate lines now are defined over an unstructured lateral grid so that pillars will have polygonal base area. One starts by defining an areal tessellation on a surface that either aligns with the lateral direction or one of the major geological horizons. Then a coordinate line is introduced through each vertex in the areal grid. The lines can either be vertical, inclined to gradually align with major fault planes, as shown for the corner-point grid in Figure 3.20, or be defined so that they connect pairs of vertices in areal tessellations of two different geological horizons.

Figure 3.24 shows the key steps in the construction of a simple 2.5D PEBI grid. Starting from a Delaunay triangulation of set of generating points, we form a perpendicular bisector grid. Through each vertex in this areal tessellation, we define a coordinate line, whose angle of inclination will change from 90 degrees for vertices on the far left to 45 degrees for vertices on the far right. coordinate lines associated with individual areas cells form pillars, and are next used to extrude the areal tessellation to a volumetric grid. The resulting volumetric grid is unstructured in the lateral direction, but has a layered structure in the vertical direction and can thus be indexed using an ik index pair. This format has quite large freedom in choosing the size and shape of the grid cells to adapt to complex features such as curved faults or to improve the areal resolution in near-well zones.

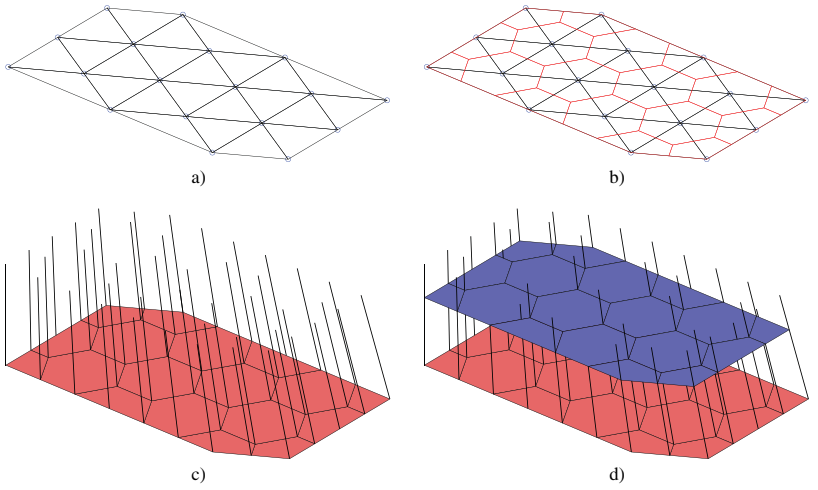


Figure 3.24 Illustration of a typical process for generating 2.5D PEBI grids. a) Triangulated point set. b) Perpendicular bisector grid. c) Coordinate lines aligned with faults. d) Volumetric extrusion.

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

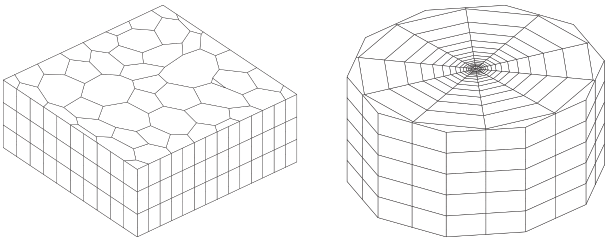


Figure 3.25 The left plot shows a 2.5D Voronoi grid derived from a perturbed 2D point mesh extruded in the z-direction, whereas the right plot shows a radial grid.

Example 1: We first construct an areal Voronoi grid from a set of generating points obtained by perturbing the vertices of a regular Cartesian grid and then use the function `makeLayeredGrid` to extrude this Voronoi grid to 3D along vertical pillars in the z-direction.

```
N=7; M=5; [x,y] = ndgrid(0:N,0:M);
x(2:N,2:M) = x(2:N,2:M) + 0.3*randn(N-1,M-1);
y(2:N,2:M) = y(2:N,2:M) + 0.3*randn(N-1,M-1);
aG = pebi(triangleGrid([x(:) y(:)]));
G = makeLayeredGrid(aG, 3);
plotGrid(G, 'FaceColor',[.8 .8 .8]);  view(-40,60); axis tight off
```

The resulting grid is shown in the left plot of Figure 3.25 and should be contrasted to the 3D tetrahedral tessellation shown to the right in Figure 3.8

Example 2: Radial symmetric grids graded towards the origin are commonly used to increase resolution near wells. Figure 3.25 shows one example that can be generated as follows:

```
P = [];
for r = exp(-3.5:.25:0),
   [x,y,z] = cylinder(r,16); P = [P [x(1,:); y(1,:)]];
end
P = unique([P'; 0 0],'rows');
G = makeLayeredGrid(pebi(triangleGrid(P)), 5);
plotGrid(G,'FaceColor',[.8 .8 .8]); view(30,50), axis tight off
```

Example 3: The model shown in Figure 3.26 is a realistic representation of a reservoir in which an unstructured areal grid has been extruded vertically to model different geological layers. Some of the layers are very thin and appear as if they were thick lines in plot **a**. Near the perimeter of the model (plot **b**), we notice that the lower layers (yellow to red colors) have been eroded away in most of the grid columns, and although the vertical dimension is strongly exaggerated, we see that the layers contain steep slopes. To a non-geologist looking at subplot **e**, it may appear as if the reservoir was formed by sediments being deposited along a sloping valley that ends in a flat plain. Subplots **c** and **d** show more details of the

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

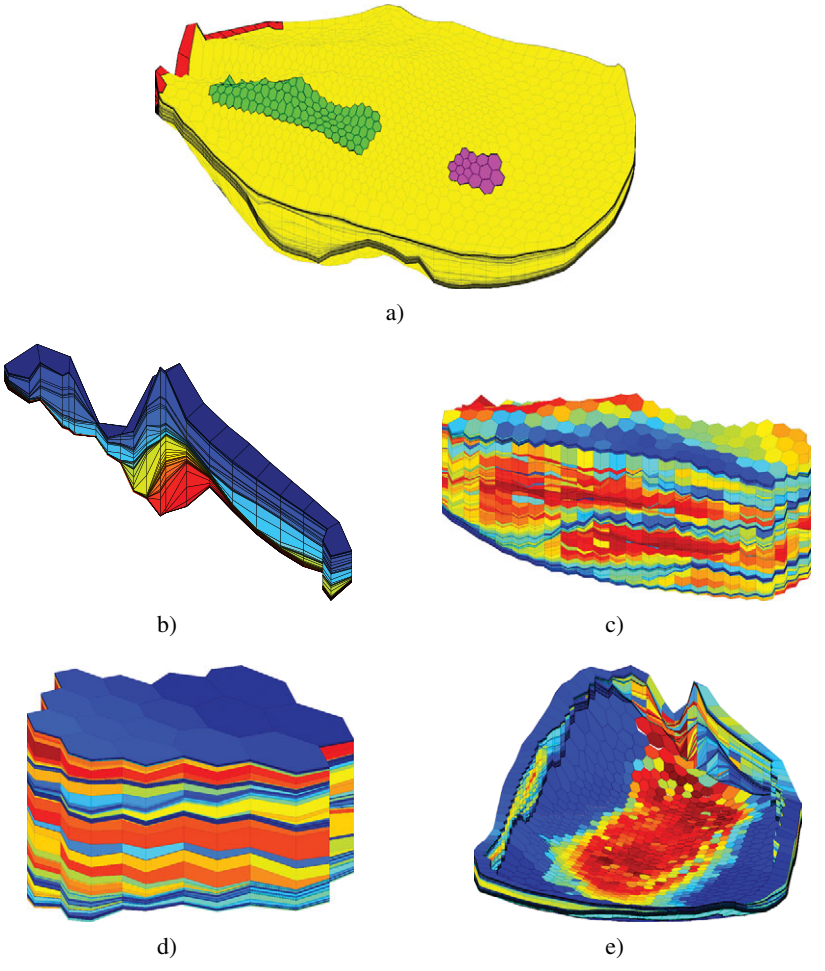


Figure 3.26 A real petroleum reservoir modeled by a 2.5D PEBI grid having 1,174 cells in the lateral direction and 150 cells along each pillar. Only 90,644 out of the 176,100 cells are active. The plots show the whole model as well as selected details. a) The whole model with three areas of interest marked in different colors. b) Layers 41 to 99 of the red region with colors representing the k-index. c) Horizontal permeability in the green region. d) Horizontal permeability in the magenta region. e) Horizontal permeability along the perimeter and bottom of the model.

permeability field inside the model. The layering is particularly distinct in plot **d**, which is sampled from the flatter part of the model. The cells in plot **c**, on the other hand, show examples of pinch-outs. The layering provides a certain ik structure in the model, similar to the logical ijk index for corner-point grids, where i refers to the areal numbering and k to the different layers. Some also prefer to associate a virtual logically Cartesian grid as an overlay to the 2.5D grid that can be used, e.g., to simplify lookup of cells in visualization. In this setup, more than one grid cell may be associated with a cell in the virtual grid.

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

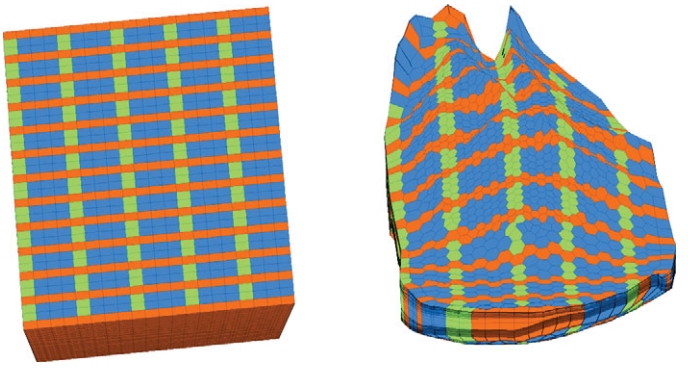


Figure 3.27 Example of a virtual $37 \times 20 \times 150$ grid used for fast lookup in a 2.5D PEBI grid with dimensions 1174×150.

3.4 Grid Structure in MRST

The two previous sections introduced various structured and unstructured grid types that can be created using the many grid-factory routines in MRST. In this section, we go into more detail about the internal data structure used to represent all these different grid formats. This data structure is in many ways the most fundamental part of MRST since almost all solvers, workflow tools, and visualization routines require an instance of a grid as input argument. By convention, instances of the grid structure are denoted `G`. If you are mainly interested in *using* solvers and visualization routines already available in MRST, you need no further knowledge of the grid structure beyond what has been encountered in the examples presented so far, and you can safely skip the remains of this section. On the other hand, if you wish to use the software to prototype new computational methods, knowledge of the inner workings of the grid structure is essential.

To provide a uniform way of accessing different grid types, all grids are stored using the same unstructured grid format that explicitly represents cells, faces, vertices, and connections between cells and faces. The main grid structure `G` contains three fields – `cells`, `faces`, and `nodes` – that specify individual properties for each individual cell/face/vertex in the grid. Grids in MRST can either be volumetric or lie on a 2D or 3D surface. The field `griddim` distinguishes volumetric and surface grids; all cells in a grid are polygonal surface patches if `griddim=2` and polyhedral volumetric entities otherwise. In addition, the grid contains a field `type` consisting of a cell array of strings describing the history of grid constructors and modifier functions that have been applied to create and modify the grid, e.g., `'tensorGrid'`. For grids having an underlying logical Cartesian structure, we also include the field `cartDims`.

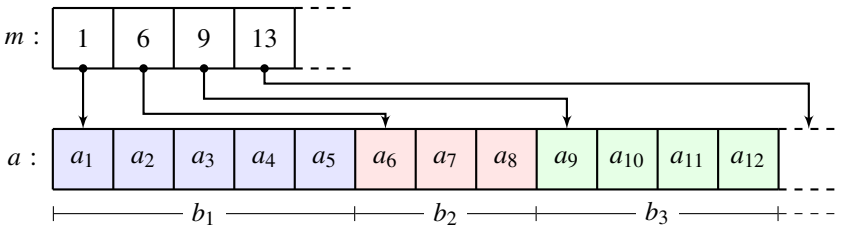
As with the rest of the basic functionality in MRST, the grid structure is thoroughly documented in the code. To read the documentation, type

```
help grid_structure
```

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

Before we continue to discuss the individual data members in the structure, we first have to introduce you to two essential implementation tricks in MATLAB.

Indirection maps and run-length encoding. In an unstructured grid, individual cells may have a different number of faces, and faces may have different number of edges/vertices. A compact way to represent such data is to use a data container and an indirection map. To illustrated, let us consider a simple example of a data object b, whose first members consist of five, three, and four numbers. This can be stored using an indirection map m and a data array $a = [a_1, \ldots, a_{12}]$, as illustrated:



Using this data structure, we can quickly find all the a values that belong to a particular b_i. In some cases, we may wish to go the other way around and determine the b_i element a particular value a_j corresponds to. A simple solution would be to introduce an extra row in a that contains the values $[1,1,1,1,1,2,2,2,3,3,3,3]$. However, this contains a lot of redundant information and can be stored much more compactly using run-length encoding, which is a simple lossless data compression method that removes repeated values. In this particular example, the run-length encoding would give a compressed array $b = [1,2,3]$ and a repetition count $n = [5,3,4]$. This data compression and decompression technique is used abundantly at a low level in MRST to avoid storing redundant information, as you will see shortly.

Representing grid cells. The cell structure `G.cells` consists of the mandatory fields:

– **`num`:** the number n_c of cells in the global grid.

– **`facePos`:** an indirection map of size `[num+1,1]` into the `faces` array. Specifically, the face information of cell `i` is found in the submatrix

```
faces(facePos(i): facePos(i+1)-1, :)
```

The number of faces per cell may be computed as `diff(facePos)`, whereas the total number of faces is $n_f =$ `facePos(end)-1`.

– **`faces`:** an $n_f \times 2$ array specifying the global faces connected to a given cell. Specifically, `faces(i,1)==j` if face with global number `faces(i,2)` is connected to cell number j. The array may optionally have a third component, `faces(i,3)`, that for certain types of grids contains a name tag used to distinguish face directions: `West`, `East`, `South`, `North`, `Bottom`, `Top`.

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

The first column of `faces` is redundant: it consists of each cell index j repeated `facePos(j+1)-facePos(j)` times and can therefore be reconstructed by decompressing a run-length encoding with the cell indices `1:num` as the encoded vector and the number of faces per cell as repetition count. Hence, to conserve memory, only the last two columns of `faces` are stored, while the first column can be reconstructed using the statement:

```
rldecode(1:G.cells.num, diff(G.cells.facePos), 2) .'
```

This construction is used a lot throughout MRST and has therefore been implemented as a utility function inside `mrst-core/utils/gridtools`

```
f2cn = gridCellNo(G);
```

– **`indexMap`:** an optional $n_c \times 1$ array that maps internal cell indices to external cell indices. For models with no inactive cells, `indexMap` equals $1 : n_c$. For cases with inactive cells, `indexMap` contains the indices of the active cells sorted in ascending order. For grids with an underlying Cartesian organization, a map of cell numbers to logical indices can be constructed using the following statements in 2D and 3D:

```
[ij{1:2}] = ind2sub(dims, G.cells.indexMap(:));    % In 2D
ij        = [ij{:}];
[ijk{1:3}] = ind2sub(dims, G.cells.indexMap(:));   % In 3D
ijk        = [ijk{:}];
```

In the latter case, `ijk(i:)` is the global (I, J, K) index of cell i.

In addition, the cell structure can contain the following optional fields that typically will be added by a call to `computeGeometry`:

– **`volumes`:** an $n_c \times 1$ array of cell volumes
– **`centroids`:** an $n_c \times d$ array of cell centroids in $\mathbb{R}^d$

Representing cell faces. The mandatory fields of the face structure, `G.faces`, are quite similar to those of the cells in the sense that they provide mappings to the lower-dimensional objects (nodes) that delimit the faces:

– **`num`:** the number n_f of global faces in the grid.
– **`nodePos`:** an indirection map of size `[num+1,1]` into the `nodes` array. Specifically, the node information of face `i` is found in the submatrix
`nodes(nodePos(i): nodePos(i+1)-1, :)`
The number of nodes of each face may be computed using the statement `diff (nodePos)`. Likewise, the total number of nodes is given as $n_n =$ `nodePos(end)-1`.
– **`nodes`:** an $n_n \times 2$ array of vertices in the grid. If `nodes(i,1)==j`, the local vertex i is part of global face number j and corresponds to global vertex `nodes(i,2)`. For each face the nodes are assumed to be oriented such that a right-hand rule

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

determines the direction of the face normal. As for `cells.faces`, the first column of `nodes` is redundant and can be easily reconstructed. Hence, to conserve memory, only the last column is stored, while the first column can be constructed using the statement:

```
rldecode(1:G.faces.num, diff(G.faces.nodePos), 2) .'
```

– **`neighbors`:** an $n_f \times 2$ array of neighboring information. Global face i is shared by global cells `neighbors(i,1)` and `neighbors(i,2)`. One of the entries in `neighbors(i,:)`, but not both, can be zero, to indicate that face i is an external face that belongs to only one cell (the nonzero entry).

In addition to the mandatory fields, `G.faces` has optional fields containing geometry information that typically are added by a call to `computeGeometry`:

– **`areas`:** an $n_f \times 1$ array of face areas.
– **`normals`:** an $n_f \times d$ array of **area weighted**, directed face normals in $\mathbb{R}^d$. The normal on face i points from cell `neighbors(i,1)` to cell `neighbors(i,2)`.
– **`centroids`:** an $n_f \times d$ array of face centroids in $\mathbb{R}^d$.

Moreover, `G.faces` can sometimes contain an $n_f \times 1$ (int8) array, `G.faces.tag`, with user-defined face indicators used, e.g., to specify that the face belongs to a fault.

Representing vertices. The vertex structure, `G.nodes`, consists of two fields:

– **`num`:** number n_n of global nodes (vertices) in the grid,
– **`coords`:** an $n_n \times d$ array of physical nodal coordinates in $\mathbb{R}^d$. Global node i is at physical coordinate `coords(i,:)`.

The grid is constructed according to a right-handed coordinate system where the z coordinate is interpreted as depth. Consequently, plotting routines such as `plotGrid` display the grid with a reversed z axis. To illustrate how the grid structure works, we consider two examples.

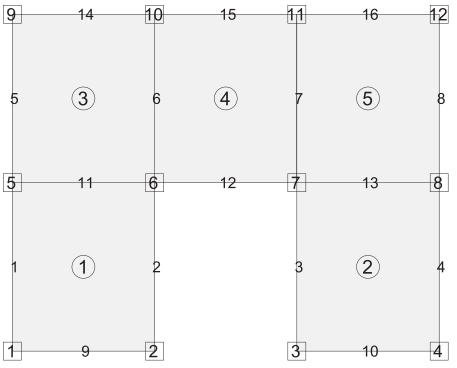
Example 1: We start by considering a regular 3×2 grid, in which we take away the second cell in the logical numbering,

```
G = removeCells( cartGrid([3,2]), 2)
```

This produces the output

```
G =
       cells: [1x1 struct]
       faces: [1x1 struct]
       nodes: [1x1 struct]
    cartDims: [3 2]
        type: {'tensorGrid'  'cartGrid'  'removeCells'}
     griddim: 2
```

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press




```
cells.faces =          faces.nodes =      faces.neighbors =
 1   1   1 (West)          1   1              0   1
 1   9   3 (South)         1   5              1   0
 1   2   2 (East)          2   2              0   2
 1  11   4 (North)         2   6              2   0
 2   3   1 (West)          3   3              0   3
 2  10   3 (South)         3   7              3   4
 2   4   2 (East)          4   4              4   5
 2  13   4 (North)         4   8              5   0
 3   5   1 (West)          5   5              0   1
 3  11   3 (South)         5   9              0   2
 3   6   2 (East)          6   6              1   3
 3  14   4 (North)         6  10              0   4
 4   6   1 (West)          7   7              2   5
 4  12   3 (South)         7  11              3   0
 4   7   2 (East)          8   8              4   0
 4  15   4 (North)         8  12              5   0
 5   7   1 (West)          9   2
 5  13   3 (South)         9   1
 5   8   2 (East)          :   :
 5  16   4 (North)         :   :
```

Figure 3.28 Illustration of the `cell` and `faces` fields of the grid structure: cell numbers are marked by circles, node numbers by squares, and face numbers have no marker.

Examining the output from the call, we notice that the field `G.type` contains three values, `'cartGrid'` indicates the creator of the grid, which again relies on `'tensorGrid'`, whereas the field `'removeCells'` indicates that cells have been removed from the Cartesian topology. The resulting 2D geometry consists of five cells, twelve nodes, and sixteen faces. All cells have four faces and hence `G.cells.facePos = [1 5 9 13 17 21]`. Figure 3.28 shows[4] the geometry and topology of the grid, including the content of the fields `cells.faces`, `faces.nodes`, and `faces.neighbors`. We notice, in particular, that all interior faces (6, 7, 11, and 13) are represented twice in `cells.faces` as they belong to two different cells. Likewise, for all exterior faces, the corresponding row in `faces.neighbors` has one zero entry. Finally, being logically Cartesian, the grid structure contains a few optional fields:

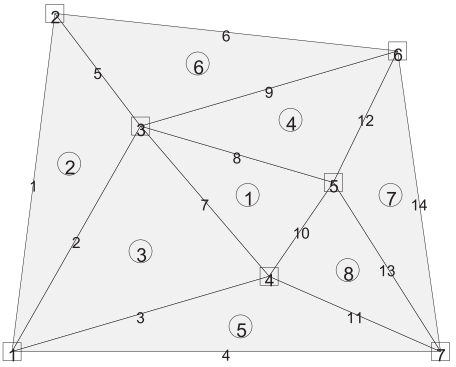
- `G.cartDims` equals `[3 2]`,
- `G.cells.indexMap` equals `[1 3 4 5 6]`, since the second cell in the logical numbering has been removed from the model, and
- `G.cells.faces` contains a third column with tags that distinguish global directions for the individual faces.

Example 2: We consider the Delaunay triangulation of seven points in 2D:

```
p = [ 0.0, 1.0, 0.9, 0.1, 0.6, 0.3, 0.75; ...
      0.0, 0.0, 0.8, 0.9, 0.2, 0.6, 0.45]'; p = sortrows(p);
G = triangleGrid(p)
```

[4] To create the plot in Figure 3.28, we first called `plotGrid` to plot the grid, then called `computeGeometry` to compute cell and face centroids, which were used to place a marker and a text label with the cell/face number in the correct position.

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press



```
cells.faces =
   1  10
   1   8
   1   7
   2   1
   2   2
   2   5
   3   3
   3   7
   3   2
   4   8
   4  12
   4   9
   5   3
   5   4
   5  11
   6   9
   6   6
   6   5
   7  13
   7  14
   :   :
```

```
faces.nodes =
   1   1
   1   2
   2   1
   2   3
   3   1
   3   4
   4   1
   4   7
   5   2
   5   3
   6   2
   6   6
   7   3
   7   4
   8   3
   8   5
   9   3
   9   6
  10   4
  10   5
   :   :
```

```
faces.neighbors =
   0   2
   2   3
   3   5
   5   0
   6   2
   0   6
   1   3
   4   1
   6   4
   1   8
   8   5
   4   7
   7   8
   0   7
```

Figure 3.29 Illustration of the `cell` and `faces` fields of the grid structure: cell numbers are marked by circles, node numbers by squares, and face numbers have no marker.

which produces the output

```
G =
    faces: [1x1 struct]
    cells: [1x1 struct]
    nodes: [1x1 struct]
     type: {'triangleGrid'}
  griddim: 2
```

The grid contains no structured parts, and thus `G` consists of the three mandatory fields, `cells`, `faces`, and `nodes`, that are sufficient to determine the geometry and topology of the grid; the `type` tag that names its creator, and `griddim` that gives that it is a surface grid. Altogether, the grid consists of eight cells, fourteen faces, and seven nodes, which are shown in Figure 3.29, along with the contents of the fields `cells.faces`, `faces.nodes`, and `faces.neighbors`. Notice, in particular, the absence of the third column in `cells.faces`, since face tags associated with the axial directions generally do not make sense for a (fully) unstructured grid. Likewise, the `cells` structure does not contain any `indexMap`, as all cells in the model are active.

Computing geometry information. All grid-factory routines in MRST generate the basic geometry and topology of a grid, that is, how nodes are connected to make up faces, how faces are connected to form cells, and how cells are connected over common faces. Whereas this information is sufficient for many purposes, more geometric information may be required in many cases. As explained earlier, such information is provided by the routine `computeGeometry`, which computes cell centroids and volumes as well as face areas, centroids, and normals. Computing this information is straightforward for simplexes and Cartesian grids, but is not so for general polyhedral grids that may contain curved polygonal faces. In the following we will therefore go through how it is done in MRST. Our discussion follows the original implementation, which in recent releases has been

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

superseded by a more optimized routine. You can find the complete original code in `computeGeometryOrig`.

For each cell, the basic grid structure provides us with a list of vertices, a list of cell faces, etc., as shown in plots **a** and **e** of Figure 3.30. The routine starts by computing face quantities (areas, centroids, and normals). To utilize MATLAB efficiently, the computations are programmed using vectorization so that each derived quantity is computed for all points, all faces, and all cells in one go. To keep the current presentation as simple as possible, we will herein only give formulas for a single face and a single cell. Let us consider a single face given by the points $\vec{p}(i_1), \ldots, \vec{p}(i_m)$ and let $\alpha = (\alpha_1, \ldots, \alpha_m)$ denote a multi-index describing how these points are connected to form the outline of the faces. For the face with global number j, the multi-index is given by the vector

```
G.faces.nodes(G.faces.nodePos(j):G.faces.nodePos(j+1)-1)
```

Let us consider two faces. Global face number two in Figure 3.30a is planar and consists of points $\vec{p}(2), \vec{p}(4), \vec{p}(6), \vec{p}(8)$ with the ordering $\alpha = (2, 4, 8, 6)$. Likewise, we consider global face number one in Figure 3.30e, which is curved and consists of points $\vec{p}(1), \ldots, \vec{p}(5)$ with the ordering $\alpha = (4, 3, 2, 1, 5)$. For curved faces, we need to make a choice of how to interpret the surface spanned by the node points. In MRST (and some commercial simulators) this is done as follows: we start by defining a so-called *hinge point* $\vec{p}_h$, which is often given as part of the input specification of the grid. If not, the hinge point can be computed as the center point of the m points that make up the face, $\vec{p}_h = \sum_{k=1}^{m} \vec{p}(\alpha_k)/m$. The hinge point can now be used to tessellate the face into m triangles, as shown in Figures 3.30b and 3.30f. The triangles are defined by the points $\vec{p}(\alpha_k)$, $\vec{p}(\alpha_{\mathrm{mod}(k,m)+1})$, and $\vec{p}_h$ for $k = 1, \ldots, m$. Each triangle has a center point $\vec{p}_c^{\,k}$ defined in the usual way as the average of its three vertexes and a normal vector and area given by

$$\vec{n}^k = \left(\vec{p}(\alpha_{\mathrm{mod}(k,m)+1}) - \vec{p}(\alpha_k)\right) \times \left(\vec{p}_h - \vec{p}(\alpha_k)\right) = \vec{v}_1^{\,k} \times \vec{v}_2^{\,k}$$
$$A^k = \sqrt{\vec{n}^k \cdot \vec{n}^k}.$$

The face area, centroid, and normal are now computed as follows

$$A_f = \sum_{k=1}^{m} A^k, \qquad \vec{c}_f = (A_f)^{-1} \sum_{k=1}^{m} \vec{p}_c^{\,k} A^k, \qquad \vec{n}_f = \sum_{k=1}^{m} \vec{n}^k. \tag{3.2}$$

The result is shown in plot **c** of Figure 3.30, where the observant reader will see that the centroid $\vec{c}_f$ does not coincide with the hinge point $\vec{p}_h$ unless the planar face is a square. This effect is more pronounced for the curved faces of the PEBI cell in Figure 3.30g.

The computation of centroids in (3.2) is only valid if the grid does not have faces with zero area, because otherwise the second formula would involve a division by zero and hence incur centroids with `NaN` values. The reader interested in creating his/her own grid-factory routines for grids that may contain degenerate (pinched) cells should be aware of this and make sure that all faces with zero area are removed in a preprocessing step.

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

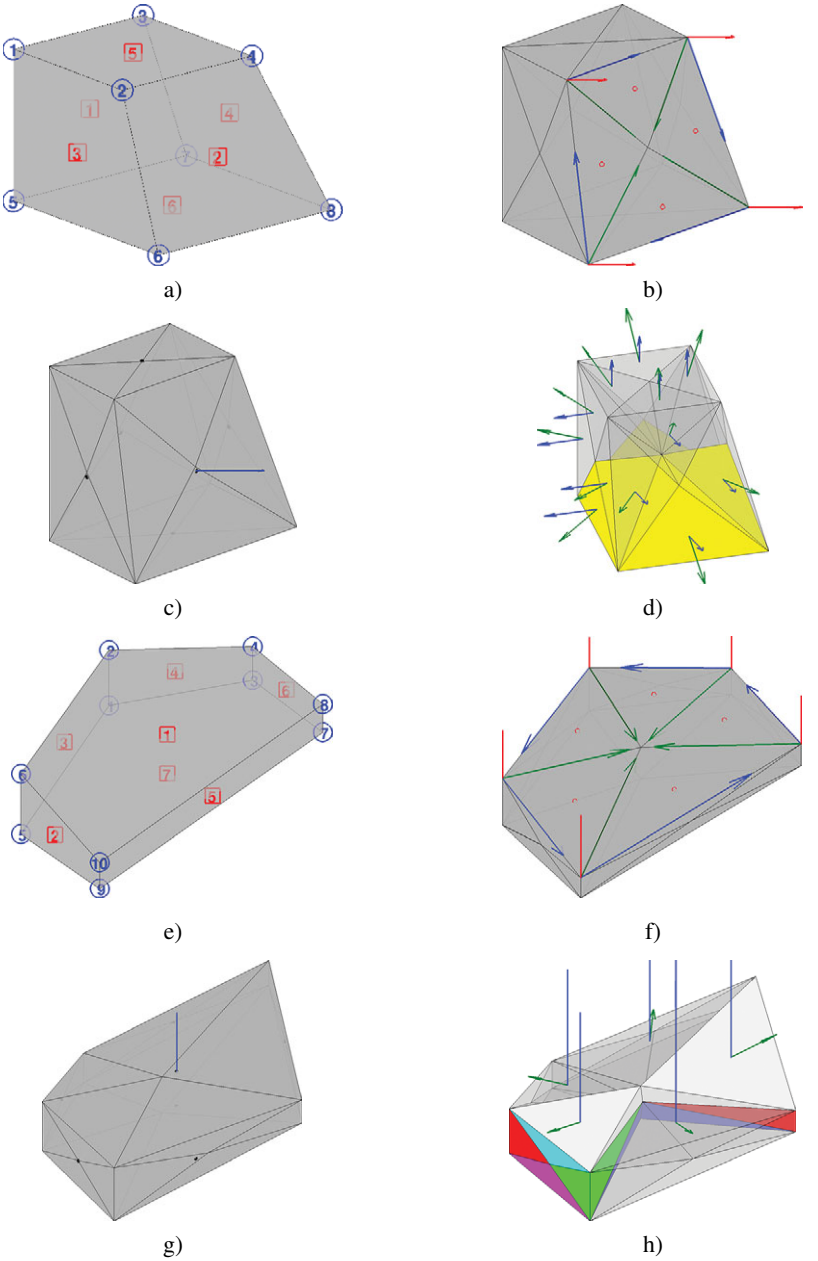


Figure 3.30 Steps in the computation of geometry information for a single corner-point cell and a single PEBI cell using `computeGeometry`. a) A single corner-point cell with face numbers (squares) and node numbers (circles). b) Tessellation of faces with vectors $\vec{v}_1^k$ (blue), $\vec{v}_2^k$ (green), and $\vec{n}^k$ (red). c) Face centroids and normal vectors computed from tessellation. d) Triangulation of cell volume with vectors $\vec{n}^k$ (blue) and $\vec{c}_r^k$ (green). e) A single PEBI cell with face numbers (squares) and node numbers (circles). f) Tessellation of faces with vectors $\vec{v}_1^k$ (blue), $\vec{v}_2^k$ (green), and $\vec{n}^k$ (red). g) Face centroids and normal vectors computed from tessellation. h) Triangulation of cell volume with vectors $\vec{n}^k$ (blue) and $\vec{c}_r^k$ (green).

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

To compute the cell centroid and volume, we start by computing the center point $\vec{c}_c$ of the cell, which we define as the average of the face centroids, $\vec{c}_c = \sum_{k=1}^{m_f} \vec{c}_f / m_f$, where m_f is the number of faces of the cell. By connecting this center point to the m_t face triangles, we define a unique triangulation of the cell volume, as shown in Figures 3.30d and 3.30h. For each tetrahedron, we define the vector $\vec{c}_r^k = \vec{p}_c^k - \vec{c}_c$ and compute the volume (which may be negative if the center point $\vec{c}_c$ lies outside the cell)

$$V^k = \tfrac{1}{3}\vec{c}_r^k \cdot \vec{n}^k.$$

The triangle normals $\vec{n}^k$ will point outward or inward depending upon the orientation of the points used to calculate them, and to get a correct computation we therefore must modify the triangle normals so that they point outward. Finally, we can define the volume and the centroid of the cell as follows

$$V = \sum_{k=1}^{m_t} V^k, \qquad \vec{c} = \vec{c}_c + \frac{3}{4V} \sum_{k=1}^{m_t} V^k \vec{c}_r^k. \tag{3.3}$$

In the original implementation, all cell quantities were computed inside a loop, which may not be as efficient as the computation of the face quantities. This has been improved in recent releases.

Computer exercises

3.4.1 Go back to Exercise 3.1.1. What would you do to randomly perturb all nodes in the grid except for those that lie on an outer face whose normal vector has no component in the y-direction?

3.4.2 Exercise 3.2.2 extended the function `triangleGrid` to triangulated surfaces in 3D. Verify that `computeGeometry` computes cell areas, cell centroids, face centroids, and face lengths correctly for 3D surfaces.

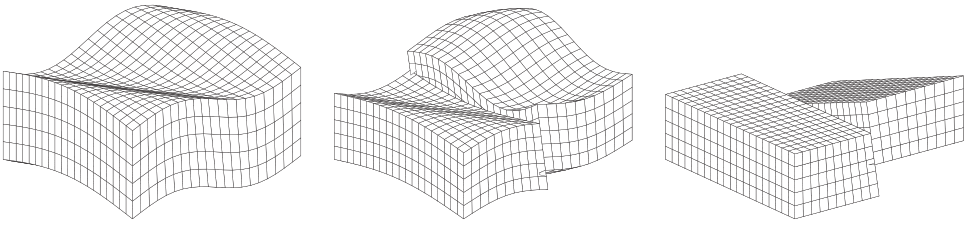
3.4.3 How would you write a function that purges all cells that have an invalid vertex (with value `NaN`) from a grid?

3.5 Examples of More Complex Grids

To help users generate test cases, MRST supplies a number of routines for generating example grids. We have previously encountered `twister`, which perturbs the x and y coordinates in a grid. Likewise, in Section 2.5 we used `simpleGrdecl` to generate a simple ECLIPSE input stream for a stratigraphic grid describing a wavy structure with a single deviated fault. The routine has several options that enable you to specify the magnitude of the fault displacement, flat rather than a wavy top and bottom surfaces, and vertical rather than inclined coordinate lines; see Figure 3.31.

The routine with the somewhat cryptic name `makeModel3` generates a corner-point input stream that models parts of an anticline structure that is cut through by two faults; see Figure 3.32. Similarly, `extrudedTriangleGrid` generates a 2.5D prismatic grid with a laterally

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

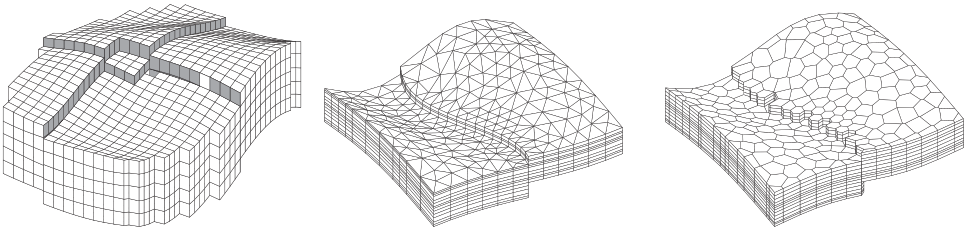


```
simpleGrdecl([20, 20, 5]);
```

```
simpleGrdecl([20, 20, 5], ...
  @(x) .05*(sin(2*pi*x)-1.5));
```

```
simpleGrdecl([20, 20, 5], ...
  @(x) .25*(x-.5),'flat',true);
```

Figure 3.31 The `simpleGrdecl` routine can be used to produce faulted, two-block grids of different shapes.



```
makeModel3([30,20,5]);
```

```
extrudedTriangleGrid(50);
```

```
extrudedTriangleGrid(50,true);
```

Figure 3.32 Three different example grids created by the grid example functions `makeModel3` and `extrudedTriangleGrid`.

curved fault in the middle. Alternatively, the routine can generate a 2.5D PEBI grid in which the curved fault is laterally stair-stepped; see Figure 3.32.

3.5.1 SAIGUP: Model of a Shallow-Marine Reservoir

Having discussed the corner-point format in some detail, it is now time to return to the SAIGUP model. In the following discussion, we will look at the grid representation in more detail and show some examples of how to interact and visualize different features of the grid (see also the last video of the second MRST Jolt on grids and petrophysical data [189]). You can find the complete source code `showGridSAIGUP`. In Section 2.5, we saw that parsing the input file creates the following structure

```
grdecl =
    cartDims: [40 120 20]
       COORD: [29766x1 double]
       ZCORN: [768000x1 double]
      ACTNUM: [96000x1 int32]
       PERMX: [96000x1 double]
         :   :    :
```

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

The first four fields describe the grid and the remaining the petrophysical propertes. Here, we will (mostly) consider the first four fields, which represent the following information:

1. The dimension of the underlying logical Cartesian grid: ECLIPSE keyword `SPECGRID`, equal $40 \times 120 \times 20$.
2. The coordinate lines: ECLIPSE keyword `COORD`, top and bottom coordinate per vertex in the logical 40×120 areal grid, i.e., $6 \times 41 \times 121$ values.
3. The coordinates along the individual coordinate lines: ECLIPSE keyword `ZCORN`, eight values per cell, i.e., $8 \times 40 \times 120 \times 20$ values.
4. The Boolean indicator for active cells: ECLIPSE keyword `ACTNUM`, one value per cell, i.e., $40 \times 120 \times 20$ values.

To process the ECLIPSE input stream and turn the corner-point grid into MRST's unstructured description, we use the `processGRDECL` routine. The interested reader may ask the processing routine to display diagnostic output

```
G = processGRDECL(grdecl, 'Verbose', true);
G = computeGeometry(G)
```

and consult the SAIGUP tutorial (`showSAIGUP`) or the technical documentation of the processing routine for a more detailed explanation of the resulting output.

The model has been created with vertical pillars having lateral resolution of 75 meters and vertical resolution of 4 meters, giving a typical aspect ratio of 18.75. You can see this, e.g., by extracting the pillars and corner points and analyzing the results as follows:

```
[X,Y,Z] = buildCornerPtPillars(grdecl,'Scale',true);
dx = unique(diff(X)).'
[x,y,z] = buildCornerPtNodes(grdecl);
dz = unique(reshape(diff(z,1,3),1,[]))
```

The resulting grid has 78,720 cells that are almost equal in size (consult the histogram `hist(G.cells.volumes)`), with cell volumes varying between 22,500 m^3 and 24,915 m^3. Altogether, the model has 264,305 faces: 181,649 vertical faces on the outer boundary and between lateral neighbors, and 82,656 lateral faces on the outer boundary and between vertical neighbors. Most of the vertical faces are not part of a fault and are therefore parallelograms with area equal 300 m^2. However, the remaining 26,000–27,000 faces are a result of the subdivision introduced to create a matching grid along the (stair-stepped) faults. Figure 3.33 shows where these faces appear in the model and a histogram of their areas: the smallest face has an area of $5.77 \cdot 10^{-4}$ m^2 and there are 43, 202, and 868 faces with areas smaller than 0.01, 0.1, and 1 m^2, respectively. The `processGRDECL` routine has an optional parameter `'Tolerance'` that sets the minimum distance for distinguishing points along coordinate lines (default value is zero). By setting this to parameter to 5, 10, 25, or 50 cm, the area of the smallest face is increased to 0.032, 0.027, 0.097, or 0.604 m^2, respectively. In general, we advice against aggressive use of this tolerance parameter; one should

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

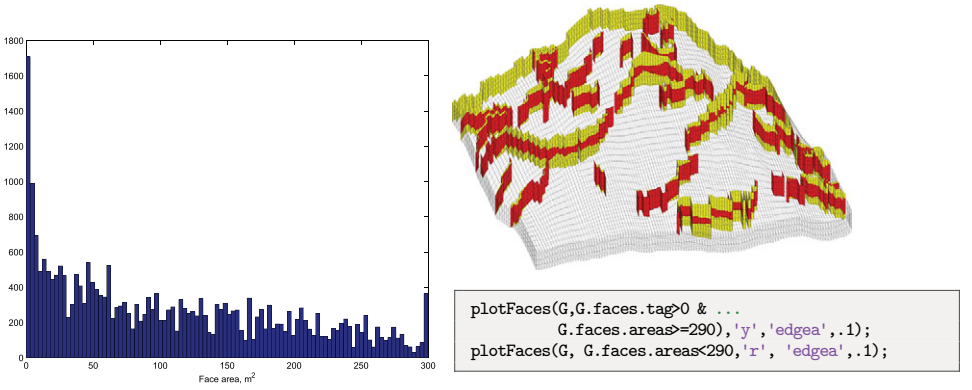


Figure 3.33 Faces that have been subdivided for the SAIGUP mode. The left plot shows a histogram of the faces areas. The right plot shows all fault faces (yellow) and fault faces having area less than 290 m^2 (red).

instead develop robust discretization schemes and, if necessary, suitable postprocessing methods that eliminate or ignore faces with small areas.

Visualizing subsets of the grid. Next, we show a few examples of visualizations highlighting various mechanisms for interacting with the grid and accessing parts of it. As a first example, we start by plotting the layered structure of the model. To this end, we use a simple trick: create a matrix with ones in all cells of the logical Cartesian grid and then do a cumulative summation in the vertical direction to get increasing values,

```
val = cumsum(ones(G.cartDims),3);
```

which we then plot using a standard call to `plotCellData`; see the left plot in Figure 3.34. To reveal the layering in the interior of the model, we extract and visualize only the cells that are adjacent to a fault:

```
cellList = G.faces.neighbors(G.faces.tag>0, :);
cells    = unique(cellList(cellList>0));
```

The first statement uses logical indexing to loop through all faces and extract the neighboring cells of all faces that are marked with a tag (i.e., lie at a fault face). The list may have repeated entries if a cell is attached to more than one fault face and contain zeros if a fault face is part of the outer boundary. We get rid of these in the second statement, and can then plot the result list of cells, giving the plot shown to the right in Figure 3.34

```
plotCellData(G,val(G.cells.indexMap),cells)
```

Next, let us inspect the fault structure in the lower-right corner of the plot. If we disregard using `cutGrdecl` as discussed on page 81, there are basically two ways we can extract parts

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

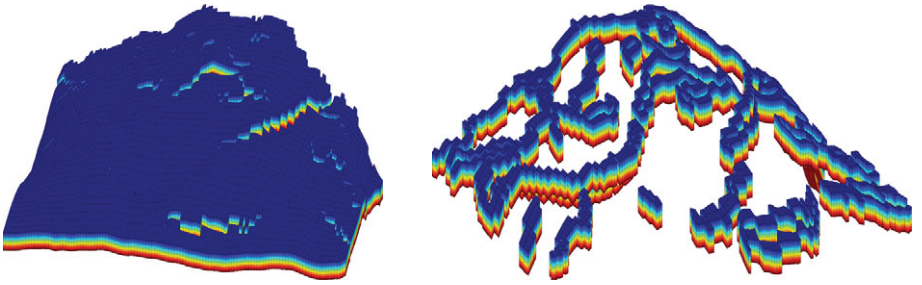


Figure 3.34 Visualizing the layered structure of the SAIGUP model.

of the model, which both rely on the construction of a map of cell numbers of logical indices. In the first method, we first construct a logical set for the cells in a logically Cartesian bounding box and then use the built-in function `ismember` to extract the members of `cells` that lie within this bounding box:

```
[ijk{1:3}] = ind2sub(G.cartDims, G.cells.indexMap); IJK = [ijk{:}];
[I,J,K] = meshgrid(1:9,1:30,1:20);
bndBox = find(ismember(IJK,[I(:), J(:), K(:)],'rows'));
inspect = cells(ismember(cells,bndBox));
```

The `ismember` function has an operational count of $\mathcal{O}(n \log n)$. A faster alternative is to use logical operations having an operational count of $\mathcal{O}(n)$. That is, we construct a vector of boolean numbers that are `true` for the entries we want to extract and `false` for the remaining entries

```
I = false(G.cartDims(1),1); I(1:9)=true;
J = false(G.cartDims(2),1); J(1:30)=true;
K = false(G.cartDims(3),1); K(1:20)=true;
pick  = I(ijk{1}) & J(ijk{2}) & K(ijk{3});
pick2 = false(G.cells.num,1); pick2(cells) = true;
inspect = find(pick & pick2);
```

Both approaches produce the same index set; the resulting plot is shown in Figure 3.35. To mark the fault faces in this subset of the model, we do the following steps

```
cellno  = gridCellNo(G);
faces   = unique(G.cells.faces(pick(cellno), 1));
inspect = faces(G.faces.tag(faces)>0);
plotFaces(G, inspect, [.7 .7 .7], 'EdgeColor','r');
```

The first statement constructs a list of cells attached to all faces in the model, the second extracts a unique list of face numbers associated with the cells in the logical vector `pick` (which represents the bounding box in logical index space), and the third statement extracts the faces within this bounding box that are marked as fault faces.

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

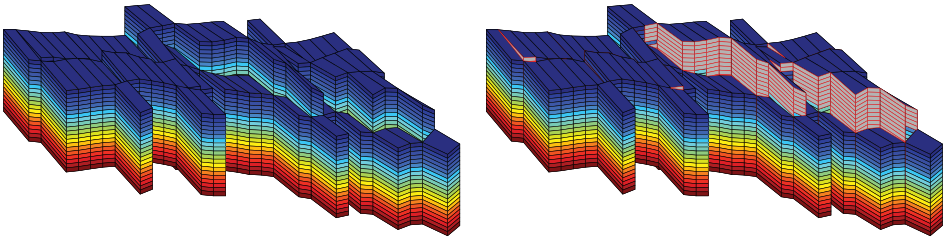


Figure 3.35 Details from the SAIGUP model showing a zoom of the fault structure in the lower-right corner of the right plot in Figure 3.34. The left plot shows the cells attached to the fault faces, and in the right plot the fault faces have been marked with gray color and red edges.

Logical operations are also useful in other circumstances. As an example, we will extract a subset of cells forming a sieve that can be used to visualize the petrophysical quantities in the interior of the model:

```
% Every fifth cell in the x-direction
I = false(G.cartDims(1),1); I(1:5:end)=true;
J = true(G.cartDims(2),1);
K = true(G.cartDims(3),1);
pickX = I(ijk{1}) & J(ijk{2}) & K(ijk{3});

% Every tenth cell in the y-direction
I = true(G.cartDims(1),1);
J = false(G.cartDims(2),1); J(1:10:end) = true;
pickY = I(ijk{1}) & J(ijk{2}) & K(ijk{3});

% Combine the two picks
plotCellData(G,rock.poro, pickX | pickY, 'EdgeColor','k','EdgeAlpha',.1);
```

3.5.2 Composite Grids

Using an unstructured grid description enables you to define composite grids whose geometries and topologies vary throughout the model. That is, different types of cells or different grid resolution may be used locally to adapt to well trajectories and flow and geological constraints; see e.g., [123, 209, 113, 202, 51, 83, 292] and references therein. You may already have encountered a composite grid if you did Exercise 3.2.7 on page 72, where we sought an unstructured grid that adapted to two skew faults and was padded with rectangular cells near the boundary. As another example, we will generate a Cartesian grid that has a radial refinement around two wells in the interior of the domain. This composite grid will be constructed from a set of control points using the `pebi` routine. To this end, we first construct the generating points for a unit refinement, as discussed in Figure 3.25 on page 86

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

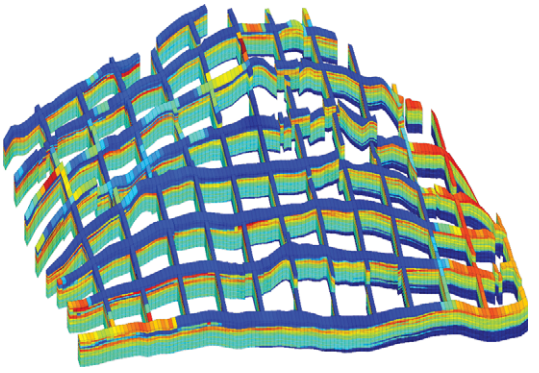


Figure 3.36 A "sieve" plot of the porosity in the SAIGUP model. Using this technique, one can more easily see the structure in the interior of the model.

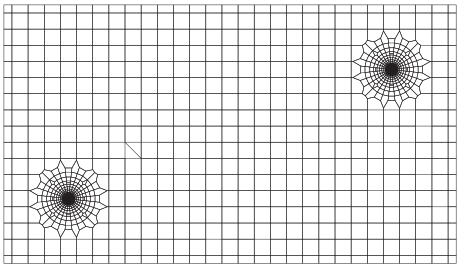


Figure 3.37 A composite grid consisting of a regular Cartesian mesh with radial refinement around two well positions.

```
Pw = [];
for r = exp(-3.5:.2:0),
  [x,y,z] = cylinder(r,28); Pw = [Pw [x(1,:); y(1,:)]];
end
Pw = [Pw [0; 0]];
```

Then this point set is translated to the positions of the wells and glued into a standard regular point lattice (generated using `meshgrid`):

```
Pw1 = bsxfun(@plus, Pw, [2; 2]);
Pw2 = bsxfun(@plus, Pw, [12; 6]);
[x,y] = meshgrid(0:.5:14, 0:.5:8);
P = unique([Pw1'; Pw2'; x(:) y(:)], 'rows');
G = pebi(triangleGrid(P));
```

Figure 3.37 shows the resulting grid. To get a good grid, it is important that the number of points around the cylinder has a reasonable match with the density of the points in the regular lattice. If not, the transition cells between the radial and the regular grid may exhibit

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

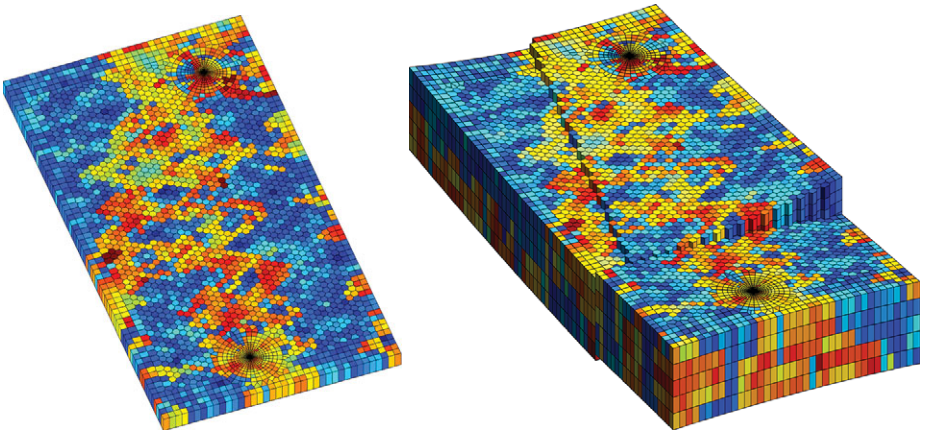


Figure 3.38 Examples of composite grids. The left plot shows an areal grid consisting of Cartesian, hexagonal, and radial parts. The right plot shows the same grid extruded to 3D with two faults added. Both faults are stair-stepped in the areal direction. The fault in the north–south direction has no displacement and thus the adjacent cells have no extra non-neighboring connections.

quite unfeasible geometries. The observant reader will also notice the layer of small cells at the boundary, which is an effect of the particular distribution of the generating points (see the left plot in Figure 3.10 on page 67) and can, if necessary be avoided by a more meticulous choice of points.

In the left plot of Figure 3.38, we have combined these two approaches to generate an areal grid consisting of three characteristic components: Cartesian grid cells at the outer boundary, hexagonal cells in the interior, and a radial grid with exponential radial refinement around two wells. The right plot shows a 2.5D grid in which the areal Voronoi grid has been extruded to 3D along vertical pillars. In addition, structural displacement has been modeled along two areally stair-stepped faults that intersect near the west boundary. Petrophysical parameters have been sampled from layers 40–44 of the SPE 10 data set [71].

3.5.3 Control-Point and Boundary Conformal Grids

To correctly split the reservoir volume into sub-volumes, build flow units with similar or correlated petrophysical properties, and resolve flow patterns, it is important that a volumetric simulation grid conforms as closely as possible to fault surfaces and horizons representing the structural architecture of the reservoir. This can be achieved by introducing unstructured grids whose cell interfaces adapt to curved lines in 2D and triangulated surfaces in 3D. Likewise, it is of interest to have grids that can adapt the centroids of certain cells to 2D lines or 3D lines or planes. This is particularly relevant if one were to provide accurate description of well paths and the near-well region. Modern wells are typically long and horizontal and may have complex geometry that consists of multiple branches. Such

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

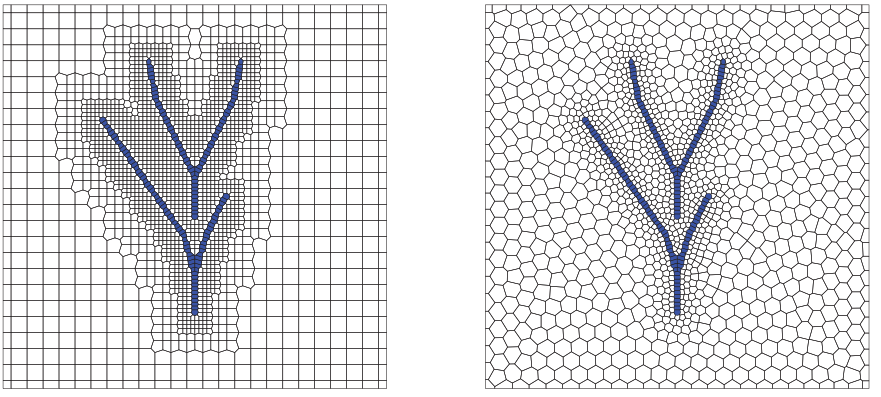


Figure 3.39 Two different 2D grids adapting to curved well paths. The hybrid grid shown to the left has Cartesian cells away from the well path, with two levels of refinement in the near-well zone. Polygonal cells are used on the transition between refinement levels and to track the well path. The Voronoi grid shown to the right has gradual refinement towards the well and is computed as the dual to a triangulation generated by `DistMesh`.

wells are increasingly becoming the main determining factor for reservoir flow patterns because of their long reach, but also as a result of various techniques for modifying the near-well region to increase injectivity or use of (intelligent) inflow devices to control fluid production.

MRST has a third-party module called `upr`, which offers such adaptivity in the form of 2D Voronoi grids and/or composite grids as well as fully 3D Voronoi grids. The module was initially developed by one of my previous students as part of his master thesis [42]. The most recent version offers two types of conformity to lower-dimensional objects: (i) control-point alignment of cell centroids to accurately represent horizontal and multilateral wells or fracture networks and (ii) boundary alignment of cell faces to accurately preserve layers, fractures, faults, pinchouts, etc.

Each constraint is represented as a lower-dimensional grid, and intersections of two objects is associated with a grid with dimension one lower than the objects. The grids are hierarchically consistent in the sense that cell faces of a higher-dimensional grid conform to the cells of all lower-dimensional grids. I will not go into more details about the underlying methods; you can find more details, as well as references to other related work, in [42, 169]. Instead, I have included Figures 3.39– 3.42 to illustrate the flexibility in geometric modeling inherent in such grids.

3.5.4 Multiblock Grids

A somewhat different approach to get grids whose geometry and topology vary throughout the physical domain is to use multiblock grids in which different types of structured or

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

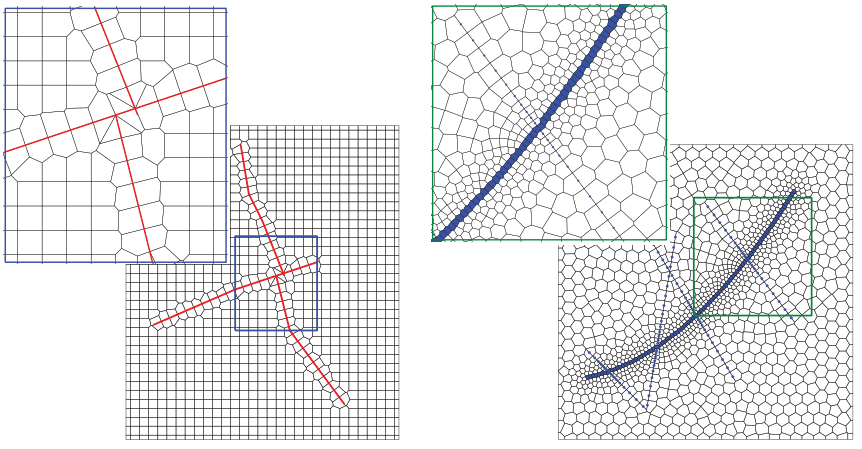


Figure 3.40 The left grid adapts to three intersecting faults, whereas the grid to the right adapts to both cell and face constraints.

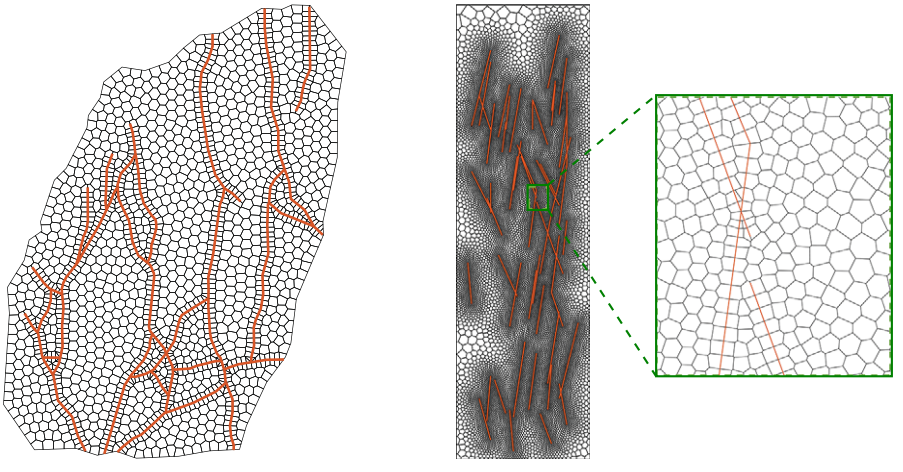


Figure 3.41 The left grid is an areal description of a complex fault network taken from Branets et al. [46] and is generated as a 2D centroidal Voronoi diagram (CVD) found by minimizing an energy functional. In the right grid, there are 51 randomly distributed fractures, and the resolution is graded toward these.

unstructured subgrids are glued together. The resulting grid can be nonmatching across block interfaces (see e.g., [313, 25, 312]) or have grid lines that are continuous (see e.g., [148, 180]). MRST does not have any grid-factory routine for generating advanced multi-block grids, but offers the function `glue2DGrid` for gluing together rectangular blocks in

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

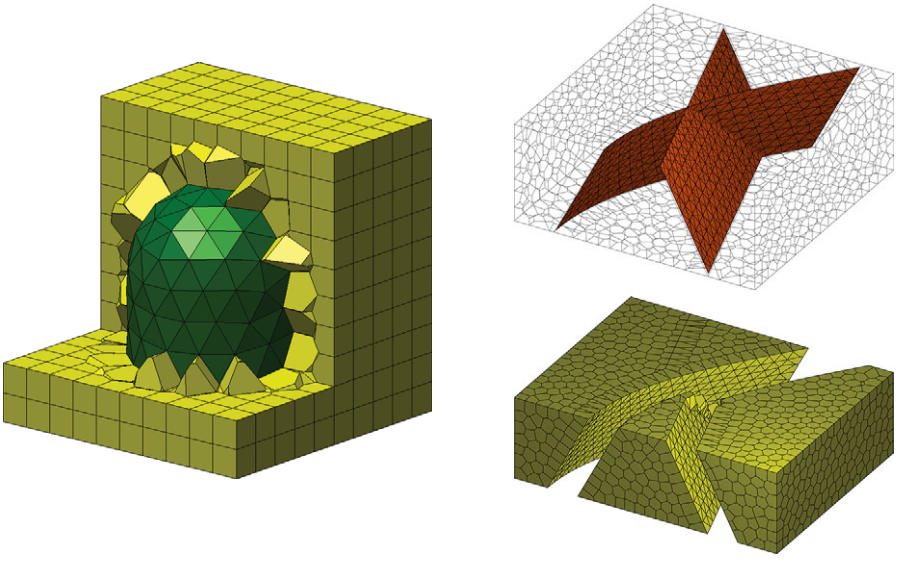


Figure 3.42 The left grid shows a conceptual model of a salt dome, with triangulated surface extracted from a 3D triangulation generated by `DistMesh`. The plots to the right shows a 3D Voronoi grid adapted to a curved and a planar fault. At the intersection of these, only the curved fault is represented exactly.

2D. These grids can, if necessary, be extruded to 3D by the use of `makeLayeredGrid`. In the following, we will show a few examples of such grids.

As our first example, let us generate a curvilinear grid that has a local refinement at its center as shown in Figure 3.43 (see also Exercise 3.2.6 on page 71). To this end, we start by generating three different block types shown in red, green, and blue colors to the left in the figure:

```
G1 = cartGrid([ 5  5],[1 1]);
G2 = cartGrid([20 20],[1 1]);
G3 = cartGrid([15  5],[3 1]);
```

Once these are in place, we translate the blocks and glue them together and then apply the `twister` function to make a curvilinear transformation of each grid line:

```
G = glue2DGrid(G1, translateGrid(G2,[1 0]));
G = glue2DGrid(G,  translateGrid(G1,[2 0]));
G = glue2DGrid(G3, translateGrid(G, [0 1]));
G = glue2DGrid(G,  translateGrid(G3,[0 2]));
G = twister(G);
```

Let us now replace the central block by a patch consisting of triangular cells. To this end, we start by generating a new grid `G2`

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

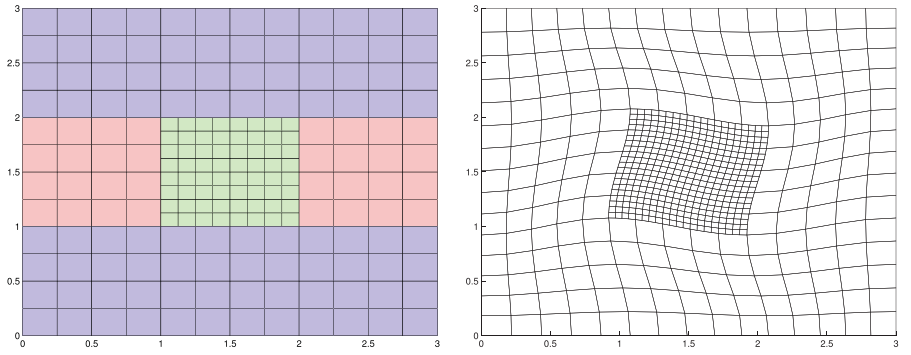


Figure 3.43 Using a multiblock approach to construct a rectilinear grid with refinement.

```
[N,M]=deal(10,15);
[x,y] = ndgrid( linspace(0,1,N+1), linspace(0,1,M+1));
x(2:N,2:M) = x(2:N,2:M) + 0.3*randn(N-1,M-1)*max(diff(xv));
y(2:N,2:M) = y(2:N,2:M) + 0.3*randn(N-1,M-1)*max(diff(yv));

G2 = computeGeometry(triangleGrid([x(:) y(:)]));
```

The `glue2DGrid` routine relies on face tags as explained in Figure 3.4 on page 89 that can be used to identify the external faces facing east, west, north, and south. Generally, such tags do not make much sense for triangular grids and are therefore not supplied. However, to be able to find the correct interface to glue together, we need to supply tags on the perimeter of the triangular patch, where the normal vectors follow the axial directions and tags therefore make sense. To this end, we start by computing the true normal vectors:

```
hf = G2.cells.faces(:,1);
hf2cn = gridCellNo(G2);
sgn = 2*(hf2cn == G2.faces.neighbors(hf, 1)) - 1;
N   = bsxfun(@times, sgn, G2.faces.normals(hf,:));
N   = bsxfun(@rdivide, N, G2.faces.areas(hf,:));
n   = zeros(numel(hf),2); n(:,1)=1;
```

Then, all interfaces that face east are those whose dot-product with the vector $(1,0)$ is identical to -1:

```
G2.cells.faces(:,2) = zeros(size(hf));
i = sum(N.*n,2)==-1; G2.cells.faces(i,2) = 1;
```

Similarly, we can identify all the interfaces facing west, north, and south. The left plot in Figure 3.44 shows the resulting multiblock grid. Likewise, the middle plot shows another multiblock grid where the central refinement is the dual to the triangular patch and `G3` has

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

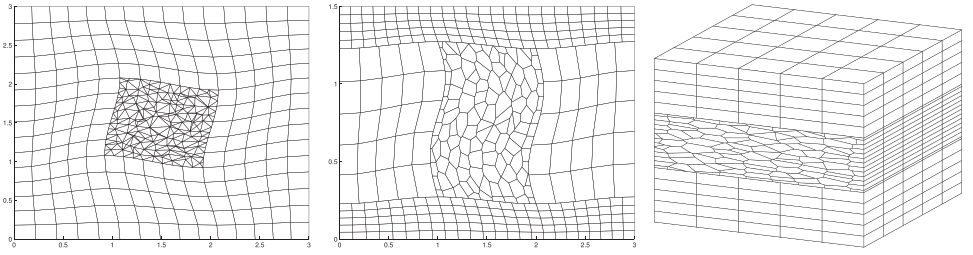


Figure 3.44 Using a multiblock approach to construct rectilinear grid with triangular (left) and polygonal refinements (middle). The right plot shows a 3D multiblock grid.

been scaled in the y-direction and refined in the x-direction so that the grid lines are no longer matching with the grid lines of `G1`.

As a last example, let us use this technique to generate a 3D multiblock grid that consists of three blocks in the vertical direction

```
G = glue2DGrid(G1, translateGrid(G2,[0 1]));
G = glue2DGrid(G,  translateGrid(G1,[0 2]));
G = makeLayeredGrid(G, 5);
G.nodes.coords = G.nodes.coords(:,[3 1 2]);
```

That is, we first generate an areal grid in the xy-plane, extrude it to 3D along the z direction, and then permute the axis so that their relative orientation is correctly preserved (notice that simply flipping `[1 2 3]`$\rightarrow$`[1 3 2]`, for instance, will *not* create a functional grid).

Computer exercises

3.5.1 How would you populate the grids shown in Figures 3.31 and 3.32 with petrophysical properties so that spatial correlation and displacement across the fault(s) is correctly accounted for? As an illustrative example, you can try to sample petrophysical properties from the SPE 10 data set.

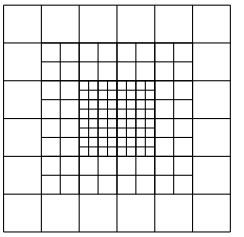
3.5.2 Select at least one of the models in the data sets `BedModels1` or `BedModel2` and try to find all inactive cells and then all cells that do not have six faces. Hint: it may be instructive to visualize these models both in physical space and in index space.

3.5.3 Extend your function from Exercise 3.2.7 on page 72 to also include radial refinement in near-well regions as shown in Figure 3.37.

3.5.4 Make a grid similar to the one shown to the right in Figure 3.38. Hint: although it is not easy to see, the grid is matching across the fault, which means that you can use the method of fictitious domain to make the fault structure.

https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press

3.5.5 As pointed out in Exercise 3.2.6, MRST does not yet have a grid factory routine to generate structured grids with local nested refinement as shown in the figure to the right. While it is not very difficult to generate the necessary vertices if each refinement patch is rectangular and matches the grid cells on the coarser level, building the grid structure may prove to be a challenge. Try to develop an efficient algorithm and implement it in MRST.



https://doi.org/10.1017/9781108591416.005 Published online by Cambridge University Press