14
Grid Coarsening

Over the last decades, methods for characterizing subsurface rock formations have improved tremendously. This has, together with a dramatic increase in computational power, enabled the industry to build increasingly detailed and complex models to account for heterogeneous structures on different spatial scales. Using gridding techniques similar to the ones outlined in Chapter 3, today one can easily build complex geological models consisting of multiple millions of cells to account for most of the features seen in typical reservoirs. In most cases, geocellular models used for reservoir characterization contain more geological layers and model fine-scale heterogeneity with higher resolution than what is used for flow simulations.

Through parallelization and use of massively parallel computers it is possible to simulate fluid flow on grid models with up to a billion cells [84, 85, 240], but such simulations require expensive infrastructure and a very high power budget and are rarely seen in practice. Contemporary high-fidelity models seem to be in the range of a few million cells, whereas the majority of asset models have ten times fewer cells, since engineers usually want to spend available computational power on more advanced flow physics or on running a large number of model realizations instead of a few highly resolved ones. To obtain computationally tractable simulation models, it is therefore common to develop reduced models through some kind of upscaling (homogenization) procedure that removes spatial detail from the geological description. Typically, a coarser model is developed by identifying regions consisting of several cells and then replacing each region by a single, coarse cell with homogeneous properties that represent the heterogeneity inside the region in some averaged sense. We discuss such upscaling methods in more detail in Chapter 15.

14.1 Grid Partitions

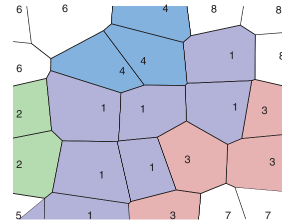
You can obviously generate coarse grids in the same way as the original by simply specify a lower spatial resolution. However, this approach has two obvious disadvantages: first of all, if the original grid has complex geometry (and topology), it is very challenging to preserve the exact geometry of the fine grid for an arbitrary coarse resolution. Second, except in simple cases, there is generally not a one-to-one mapping between cells in the fine and

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

coarse grids. Herein, we therefore choose a different approach. *In MRST, a* coarse grid *always refers to a grid that is defined as a partition of another grid, which is referred to as the* fine grid.

Tools for partitioning and coarsening of grids are found in two different modules of MRST: The `coarsegrid` module defines a basic grid structure for representing coarse grids and supplies simple routines for partitioning grids with an underlying Cartesian topology. The `agglom` module offers tools for defining flexible coarse grids that adapt to geological features and flow patterns, e.g., as discussed in [125, 124, 194, 187]. Coarse partitions also form a basis for contemporary multiscale methods [195] and are used extensively in the various multiscale modules of MRST (`msrsb`, `msmfem`, `msfv`, etc). In this chapter, we first discuss functionality for generating and representing coarse grids found in the `coarsegrid` module and then briefly outline some of the more advanced functions found in the `agglom` module. In an attempt to distinguish fine and coarse grids, we henceforth refer to fine grids as consisting of *cells*, whereas coarse grids are said to consist of *blocks*.

Coarse grids in MRST are represented by a structure that consists entirely of topological information stored in the same fields as for the general grid structure introduced in Section 3.4. As a naming convention, we use `CG` to refer to a coarse-grid structure and `G` to refer to the usual (fine) grid. A coarse grid is always related to a fine grid in the sense that



- each cell in the fine grid `G` belongs to one, and only one, block in the coarse grid `CG`;
- each block in `CG` consists of a *connected* subset of cells from `G`; and
- `CG` is defined by a *partition vector* p defined such that $p(i) = \ell$ if cell i in `G` belongs to block ℓ in `CG`.

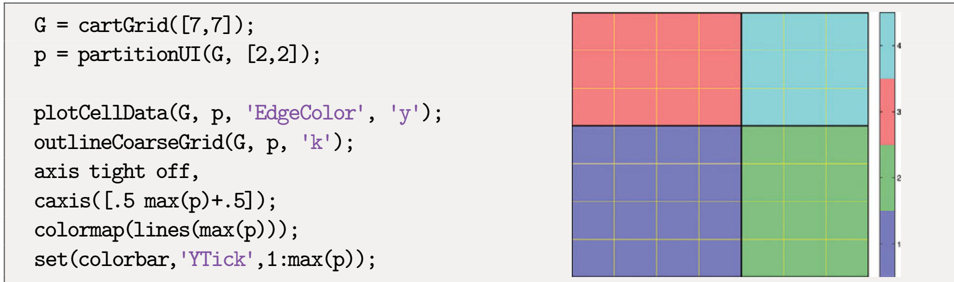
This concept is quite simple, but has proved to be very powerful in defining coarse grids that can be applied in a large variety of computational algorithms. We will come back to the details of the grid structure in Section 14.2. First, let us discuss how to define partition vectors in some detail, as this is more useful from a user perspective than understanding the details of how the `CG` structure is implemented.

To demonstrate the simplicity and power of using partition vectors to define coarse grids, we go through a set of examples. You can find complete codes for all the following examples in `showPartitions.m` from the `book` module.

14.1.1 Uniform Partitions

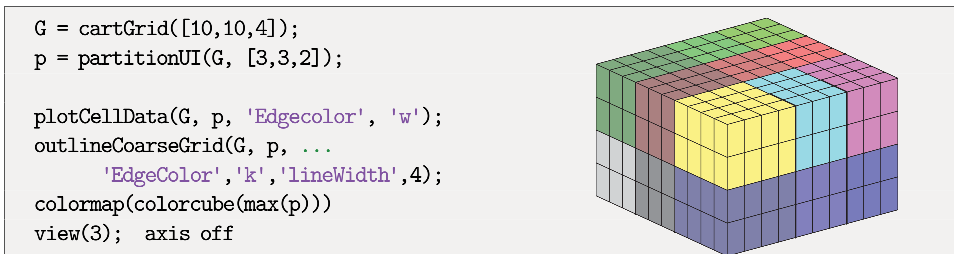
For all grids having a logically Cartesian topology, i.e., grids that have a valid field `G.cartDims`, we can use the function `partitionUI` to generate a relatively uniform partition that consists of the tensor product of a load-balanced linear partition in each index direction. As an example, let us partition a 7 × 7 fine grid into a 2 × 2 coarse grid:

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press



The call to `partitionUI` returns a vector with one element per cell taking one of the integer values $1, 2, 3, 4$ that represent the four blocks. Since seven is not divisible by two, the coarse blocks do not have the same size but consist of 4×4, 3×3, 4×3, and 3×4 cells. To better distinguish different blocks in the plot, we have used `outlineCoarseGrid(G, p)` to find and plot all faces in `G` whose neighboring cells have different values of `p`.

The same procedure can, of course, also be applied to partition any grid in 3D that has a Cartesian topology. As an example, we consider a simple box geometry:

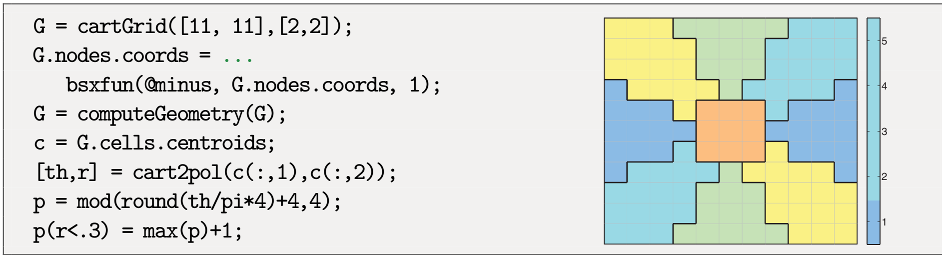


Here, we have used the `colorcube` colormap, which is particularly useful for visualizing partition vectors, since it contains as many regularly spaced colors in RGB color space as possible. The careful reader will also observe that the arguments to `outlineCoarseGrid` changes somewhat for 3D grids.

14.1.2 Connected Partitions

All you need to partition a grid is a partition vector. This vector can be given by the user, read from a file, generated by evaluating a geometric function, or given as the output of some user-specified algorithm. As a simple example of the latter, let us partition the box model $[-1, 1] \times [-1, 1]$ into nine different blocks using the polar coordinates of the cell centroids. The first block is defined as $r \leq 0.3$, whereas the remaining eight are defined by segmenting $4\theta/\pi$:

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press



```
G = cartGrid([11, 11],[2,2]);
G.nodes.coords = ...
   bsxfun(@minus, G.nodes.coords, 1);
G = computeGeometry(G);
c = G.cells.centroids;
[th,r] = cart2pol(c(:,1),c(:,2));
p = mod(round(th/pi*4)+4,4);
p(r<.3) = max(p)+1;
```


In the second-to-last line, the purpose of the modulus operation is to avoid wrap-around effects as θ jumps from $-\pi$ to π.

The human eye should be able to distinguish nine different coarse blocks in the plot above, but the partition does unfortunately not satisfy all the criteria we prescribed on page 519. Indeed, as you can see from the colorbar, the partition vector only has five unique values and thus corresponds to five blocks, according to our definition of p: cell i belongs to block ℓ if $p(i) = \ell$. Hence, what the partition describes is one connected block at the center surrounded by four *disconnected blocks*. To determine whether a block is connected or not, we will use some concepts from graph theory. We first form a local undirected graph (or a network) in which nodes are cells and edges are cell faces connecting two cells with the same partition value. A block is then said to be disconnected if the graph has multiple *connected components*. A connected component is defined as the subgraph in which any two nodes are connected to each other through a path. In other words, a coarse block is disconnected if there exists at least one pair of cells that cannot be connected by a continuous path in the local grid graph. To get a partition satisfiying our requirements, we must split the four disconnected blocks. This is done by the following call:

```
q = processPartition(G, p);
```

which splits disconnected components that have the same p-value into separate blocks and updates the partition vector accordingly. In the current case, Blocks 1–4 will be split in two, whereas Block 5 remains unchanged, giving the nine blocks separated by solid lines in the figure.

Let us quickly outline how this routine works. We start by identifying all cells having the same partition value; the left plot in Figure 14.1 shows these cells for the first block. The grid graph is the same that we used to construct the discrete divergence and gradient operators, with the edge list given in terms of the two columns $C_1(f)$ and $C_2(f)$ from `G.faces.neighbors` that specify the cells connected by face f. The only exception is that we now have to exclude any connections between cells having different p-values. From this list, we can construct a local *adjacency matrix*. This symmetric matrix is defined so that all cells corresponding to nonzero entries in a single row (or column) are directly connected. In the two adjacency matrices shown in Figure 14.1, we have used different color for cells

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

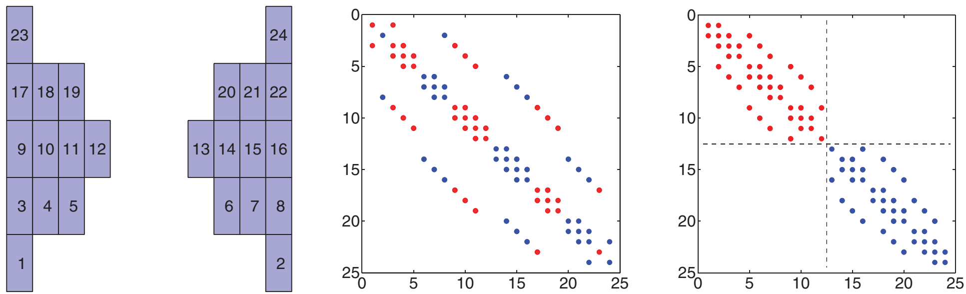


Figure 14.1 Partition of a disconnected block. The left plot shows all cells with the same p-value. The middle plot shows the adjacency matrix based on the original numbering, whereas the right plot shows the adjacency matrix after a Dulmage–Mendelsohn permutation has separated the connected components.

belonging to each of the two components for clarity. The middle plot shows that cell 1 is connected to cell 3; cell 2 is connected to cell 8; cell 3 is connected to cells 1, 4, and 9; and so on. To find the connected components, we use a Dulmage–Mendelsohn permutation. Disconnected components will then appear as diagonal blocks in the permuted adjacency matrix; see the right plot in Figure 14.1.

Graph operations like this can generally be used to adapt the partition to features in the geological model. As an example, the processing routine can also take an additional parameter `facelist` that specifies a set of faces across which the connections will be removed before processing:

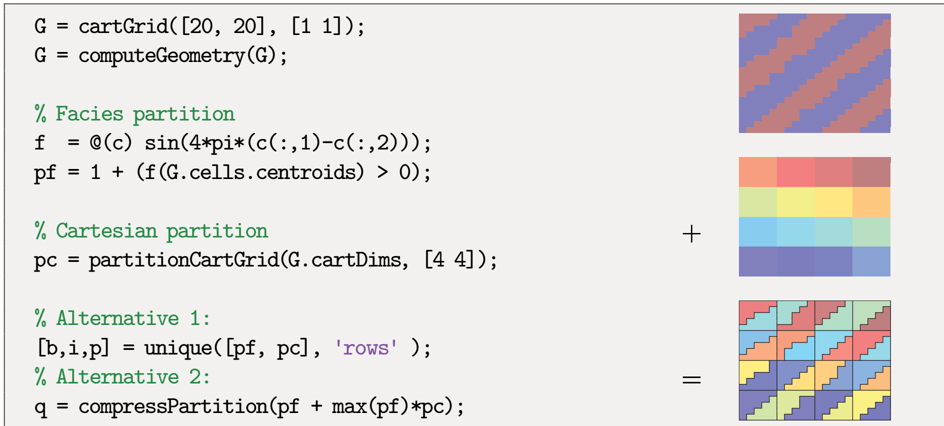
```
q = processPartition(G, p, facelist)
```

Using this functionality one can, for instance, prevent coarse blocks from crossing faults inside the model.

14.1.3 Composite Partitions

In many cases, it may be advantageous to create partition vectors by combining more than one partition principle. As an example, consider a heterogeneous medium consisting of two different facies (rock types), one with high permeability and one with low, that each form large contiguous regions. If we now let the coarse blocks respect the facies boundaries, we can assign each block a homogeneous property and avoid upscaling. Within each facies, we can further use a rectangular partition generated by `partitionCartGrid`, which is simpler and less computationally expensive than `partitionUI` but only works correctly for a grid having a fully intact logically Cartesian topology with no inactive cells, no cells that have been removed by `removeGrid`, and so on. The following code illustrates the principle:

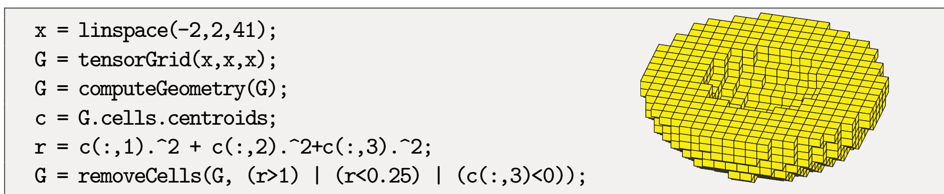
https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press



```
G = cartGrid([20, 20], [1 1]);
G = computeGeometry(G);

% Facies partition
f  = @(c) sin(4*pi*(c(:,1)-c(:,2)));
pf = 1 + (f(G.cells.centroids) > 0);

% Cartesian partition
pc = partitionCartGrid(G.cartDims, [4 4]);

% Alternative 1:
[b,i,p] = unique([pf, pc], 'rows' );
% Alternative 2:
q = compressPartition(pf + max(pf)*pc);
```

The example also shows two alternative techniques for combining different partitions. The first alternative collects the partitions as columns in an array `A`. The call `[b,i,p]=unique(A,'rows')` will return `b` as the unique rows of `A` so that `b=A(i)` and `A=b(p)`. Hence, `p` will be a partition vector that represents the unique intersection of all the partitions collected in `A`. In the second method, we treat the partition vectors as multiple subscripts, from which we compute a linear index. This index is not necessarily contiguous. Most routines in MRST require that partition vectors are contiguous to avoid having to treat special cases arising from noncontiguous partitions. To fulfill this requirement, we use the function `compressPartition` that renumbers a partition vector to remove any indices corresponding to empty grid blocks. The two alternatives have more or less the same computational complexity, and which alternative you choose in your implementation is largely a matter of what you think will be easiest to understand for others.

Altogether, the examples presented so far in this chapter explain the basic concepts of how you can create partition vectors. Before we go on to explain details of the coarse-grid structure and how to generate this structure from a given partition vector, we show one last and a bit more fancy example. To this end, we create a cup-formed grid, partition it, and then visualize the partition using a nice technique. To generate the cup-shaped grid, we use the fictitious-domain technique we previously used to generate the ellipsoidal grid in Figure 3.4 on page 61.



```
x = linspace(-2,2,41);
G = tensorGrid(x,x,x);
G = computeGeometry(G);
c = G.cells.centroids;
r = c(:,1).^2 + c(:,2).^2+c(:,3).^2;
G = removeCells(G, (r>1) | (r<0.25) | (c(:,3)<0));
```

Assume that we wish to partition this cup model into 100 coarse blocks. We could, for instance, try to use `partitionUI` to impose a regular $5 \times 5 \times 4$ partition. Because of the

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

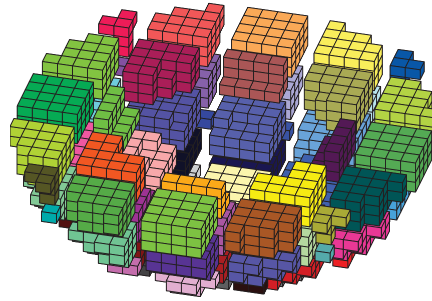
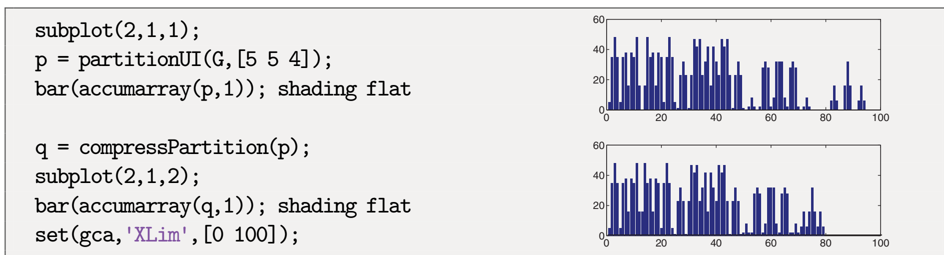


Figure 14.2 The partition of the cup-formed grid visualized with `explosionView`.

fictitious method, a large number of the ijk indices from the underlying Cartesian topology will correspond to cells that are not present in the actual grid. Imposing a regular Cartesian partition on such a grid typically gives block indices in the range $[1, \max(p)]$ that do not correspond to any cells in the underlying fine grid. In this particular case, only 79 out of the desired 100 blocks correspond to a volume within the grid model. To see this, we use the function `accumarray` to count the number of cells for each block index and plot the result as a bar chart:




```
subplot(2,1,1);
p = partitionUI(G,[5 5 4]);
bar(accumarray(p,1)); shading flat

q = compressPartition(p);
subplot(2,1,2);
bar(accumarray(q,1)); shading flat
set(gca,'XLim',[0 100]);
```

Figure 14.2 shows the partition obtained after we have compressed the partition vector. To clearly distinguish the different blocks, we have useed `explosionView` to create an *explosion view*, which is a useful technique for visualizing coarse partitions.

14.2 Coarse Grid Representation in MRST

When working with coarse grids, it is not always sufficient to only know the partition, i.e., which cells belong to which blocks. It can often be more advantageous to treat the coarse partition as a general polyhedral grid and have access to the inherent topology and possibly also the geometry. In MRST, we have chosen a compromise and use a coarse-grid structure that represents topology explicitly and geometry implicitly. This choice is motivated by flow solvers, which we previously have seen can be posed on an arbitrary grid graph provided each node has associated pore volume and depth value and each connection has an associated transmissibility.

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

Given a grid structure `G` and a partition vector `p`, we generate a structure `CG` representing the coarse grid by the following call:

```
CG = generateCoarseGrid(G, p)
```

The coarse-grid structure consists entirely of topological information stored in the same way as described in Section 3.4 for `G`: The fields `cells` and `faces` represent the coarse blocks and their connections. As a result, we can use `CG` seamlessly with many of the standard solvers in MRST. Unlike the original grid structure, however, `CG` does not represent the geometry of the coarse blocks and faces explicitly and does therefore not have a `nodes` field. The geometry information is instead obtained from the parent grid `G` and the partition vector `p`, copies of which are stored in the fields `parent` and `partition`, respectively.

The structure, `CG.cells`, that represents the coarse blocks consists of the following mandatory fields:

- **`num`:** the number N_b of blocks in the coarse grid.
- **`facePos`:** an indirection map of size `[num+1,1]` into the `faces` array, which is defined completely analogously as for the fine grid. Specifically, the connectivity information of block `i` is found in the submatrix

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

  You can now compute the number of connections of each block using the statement `diff(facePos)`.
- **`faces`:** an $N_c \times 2$ array of connections associated with a given block. Specifically, if `faces(i,1)==j`, then connection `faces(i,2)` is associated with block number j. To conserve memory, only the second column is actually stored in the grid structure. The first column can be reconstructed by a call to `rldecode`. Optionally, one may append a third column that contains a tag inherited from the parent grid.

In addition, the cell structure can contain the following optional fields, which typically are added by a call to `coarsenGeometry`, assuming that the corresponding information is available in the parent grid:

- **`volumes`:** an $N_b \times 1$ array of block volumes
- **`centroids`:** an $N_b \times d$ array of block centroids in $\mathbb{R}^d$

The face structure, `CG.faces`, consists of the following mandatory fields:

- **`num`:** the number N_c of global connections in the grid.
- **`neighbors`:** an $N_c \times 2$ array of neighboring information. Connection i is between blocks `neighbors(i,1)` and `neighbors(i,2)`. One of the entries in `neighbors(i,:)`, but not both, can be zero, to indicate that connection i is between a single block (the nonzero entry) and the exterior of the grid.

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

– **`connPos, fconn:`** packed data-array representation of the coarse → fine mapping. Specifically, the elements `fconn(connPos(i):connPos(i+1)-1)` are the connections in the parent grid (i.e., rows in `G.faces.neighbors`) that constitute coarse-grid connection `i`.

In addition to the mandatory fields, `CG.faces` has optional fields that contain geometry information and typically are added by a call to `coarsenGeometry`:

– **`areas:`** an $N_c \times 1$ array of face areas.
– **`normals:`** an $N_c \times d$ array of accumulated area-weighted, directed face normals in $\mathbb{R}^d$.
– **`centroids:`** an $N_c \times d$ array of face centroids in $\mathbb{R}^d$.

Like in `G`, the coarse grid structure also contains a field `CG.griddim` that is used to distinguish volumetric and surface grids, as well as a cell array `CG.type` of strings describing the history of grid constructor and modifier functions used to define the coarse grid.

As an illustrative example, let us partition a 4×4 Cartesian grid into a 2×2 coarse grid. This gives the following structure:

```
CG =
        cells: [1x1 struct]
        faces: [1x1 struct]
    partition: [16x1 double]
       parent: [1x1 struct]
      griddim: 2
         type: {'generateCoarseGrid'}
```

with the `cells` and `faces` fields given as

```
CG.cells =
        num: 4
    facePos: [5x1 double]
      faces: [16x2 double]
```

```
CG.faces =
          num: 12
    neighbors: [12x2 double]
      connPos: [13x1 double]
        fconn: [24x1 double]
```

Figure 14.3 shows relations between entities in the coarse grid and its parent grid. For instance, we see that block number one consists of cells one, two, five, and six because these are the rows in `CG.partition` that have value equal one. Likewise, we see that because `CG.faces.connPos(1:2)=[1 3]`, coarse connection number one is made up of two cell faces that correspond to faces number one and six in the parent grid because `CG.faces.fconn(1:2)=[1 6]`, and so on.

14.2.1 Subdivision of Coarse Faces

In the discussion so far, we have always assumed that there is only a single connection between two neighboring coarse blocks and that this connection is built up of a set of cell faces corresponding to all faces between pairs of cells in the fine grid that belong to the two different blocks. While this definition is useful for many workflows like in

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

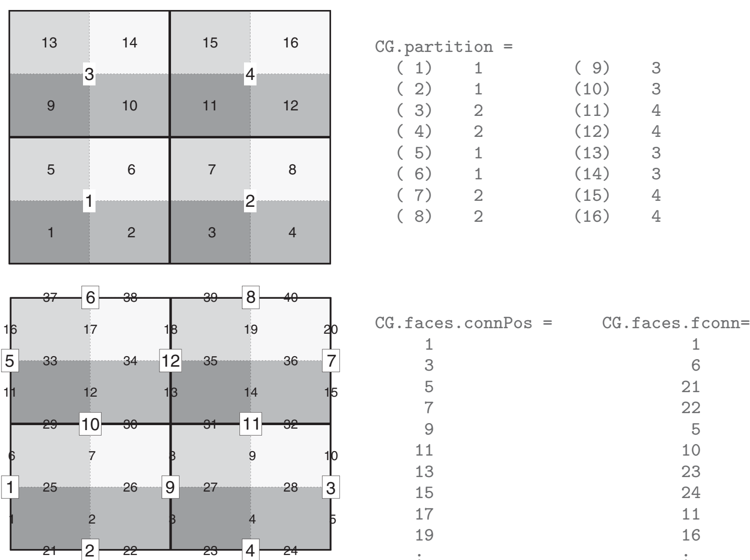


Figure 14.3 The relation between blocks in the coarse grid and cells in the parent grid (top) and between connections in the coarse grid and faces from the parent grid (bottom).

standard upscaling methods, there are also problems for which one may want to introduce more than one connection between neighboring blocks. To define a subdivision of coarse faces, we once again use a partition vector with one scalar value per face in the fine grid, defined completely analogous to vectors used for the volumetric partition. Assuming that we have two such partition vectors, pv describing the *volumetric* partition and pf describing the partition of cell faces, the corresponding coarse grid is built through the call:

```
CG = generateCoarseGrid(G, pv, pf);
```

In my experience, the simplest way to build a face partition is to compute it from an ancillary volumetric partition using the routine:

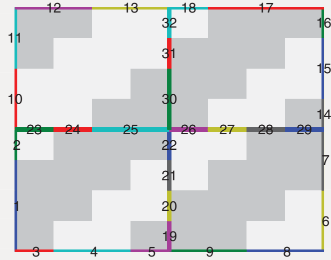
```
pf = cellPartitionToFacePartition(G, pv)
```

which assigns a unique, nonnegative integer for each pair of cell values occurring in the volumetric partition vector pv, and hence constructs a partitioning of all faces in the grid. Fine-scale faces that are not on the interface between coarse blocks are assigned zero value.

As an illustration, we continue the example from page 523. We first partition the 8×8 fine grid into a 2×2 coarse grid and then use facies information to subdivide faces of the coarse grid so that each coarse connection has a a given combination of facies values on opposite sides of the interface:

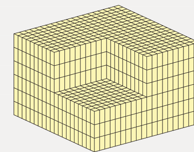
https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

```
G  = computeGeometry(cartGrid([8, 8], [1 1]));
f  = @(c) sin(3*pi*(c(:,1)-c(:,2)));
pf = 1 + (f(G.cells.centroids) > 0);
plotCellData(G, pf,'EdgeColor','none');

pv = partitionCartGrid(G.cartDims, [2 2]);
pf = cellPartitionToFacePartition(G,pf);
pf = processFacePartition(G, pv, pf);
CG = generateCoarseGrid(G, pv, pf);
CG = coarsenGeometry(CG);
cmap = lines(CG.faces.num);
for i=1:CG.faces.num
    plotFaces(CG,i,'LineWidth',6,'EdgeColor', cmap(i,:));
end
text(CG.faces.centroids(:,1), CG.faces.centroids(:,2), ...
      num2str((1:CG.faces.num)'),'FontSize',20,'HorizontalAlignment', 'center');
```



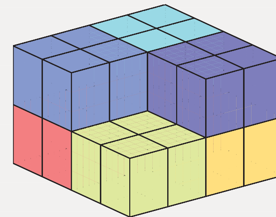
As for the volumetric partition, we require that each interface that defines a connection in the face partition consists of a connected set of cell faces. That is, it must be possible to connect any two cell faces belonging to given interface by a path that only crosses edges between cell faces that are part of the interface. To ensure that all coarse interfaces are connected collections of fine faces, we have used the routine `processFacePartition`, which splits disconnected interfaces into one or more connected interfaces.
The same principles apply also in 3D, here illustrated for a rectangular block with a rectangular cut-out:

```
G = computeGeometry(cartGrid([20 20 6]));
c = G.cells.centroids;
G = removeCells(G, ...
    (c(:,1)<10) & (c(:,2)<10) & (c(:,3)<3));
plotGrid(G); view(3); axis off
```



We introduce a volumetric partition and a face partition:

```
p = partitionUI(G,[2, 2, 2]);
q = partitionUI(G,[4, 4, 2]);
CG = generateCoarseGrid(G, p, ...
    cellPartitionToFacePartition(G,q));

plotCellData(CG,(1:max(p))','EdgeColor','none');
plotFaces(CG,1:CG.faces.num,...
    'FaceColor' , 'none' , 'LineWidth' ,2);
view(3); axis off
```

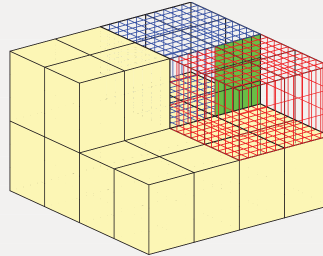


The structure `CG` also contains lookup tables for mapping blocks and interfaces in the coarse grid to cells and faces in the fine grid. To illustrate, we visualize one connection of

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

a subdivided coarse face that consists of several fine faces, along with the fine cells that belong to the neighboring blocks:

```
face  = 66;
sub   = CG.faces.connPos(face):CG.faces.connPos(face+1)-1;
ff    = CG.faces.fconn(sub);
neigh = CG.faces.neighbors(face,:);

show = false(1,CG.faces.num);
show(boundaryFaces(CG)) = true;
show(boundaryFaces(CG,neigh)) = false;
plotFaces(CG, show,'FaceColor',[1 1 .7]);
plotFaces(G, ff, 'FaceColor', 'g')
plotFaces(CG,boundaryFaces(CG,neigh), ...
   'FaceColor','none','LineWidth', 2);
plotGrid(G, p == neigh(1), 'FaceColor', 'none', 'EdgeColor', 'r')
plotGrid(G, p == neigh(2), 'FaceColor', 'none', 'EdgeColor', 'b')
```



14.3 Partitioning Stratigraphic Grids

You can also apply the principles outlined in the previous section to stratigraphic grids. To demonstrate this, we coarsen two corner-point models of industry-standard complexity: the sector model of the Johansen aquifer introduced in Section 2.5.4 and the SAIGUP model from Section 2.5.5. We also apply a few more advanced partition methods to the sector model with intersecting faults from Section 3.3.1. Full details are given in the scripts `coarsenJohansen`, `coarsenSAIGUP`, and `coarsenCaseB4`.

14.3.1 The Johansen Aquifer

The Johansen models were originally developed to study a potential site for geological storage of CO_2 injected as a supercritical fluid deep in the formation. In Section 2.5.4, we saw that the heterogeneous NPD5 sector model contains three formations: the Johansen sandstone delta bounded above by the Dunlin shale and below by the Amundsen shale. These three formations have distinctively different permeabilities (see Figure 2.15 on page 46) and play a very different roles in the sequestration process. The Johansen sandstone has relatively high porosity (and permeability) and is the container in which the CO_2 is to be kept. The low-permeability Dunlin shale acts as a seal that prevents the CO_2 from escaping back to the sea bottom, and we generally expect that the buoyant CO_2 phase will accumulate as a thin plume that migrates upward under the caprock in the up-dip direction.

To accurately simulate the up-dip migration under the top seal, it is highly important to preserve the correct interface between the Johansen sandstone and the Dunlin shale in the coarse model. (In general, it is a good advice to avoid creating coarse blocks containing large media contrasts, which would otherwise adversely affect upscaling accuracy.) We

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

Table 14.1 *Permeability values used to distinguish the different formations in the NPD5 sector model of the Johansen formation.*

Dunlin	Johansen	Amundsen
$K \leq 0.01$mD	0.1 mD $< K$	0.01 mD $< K \leq 0.1$mD

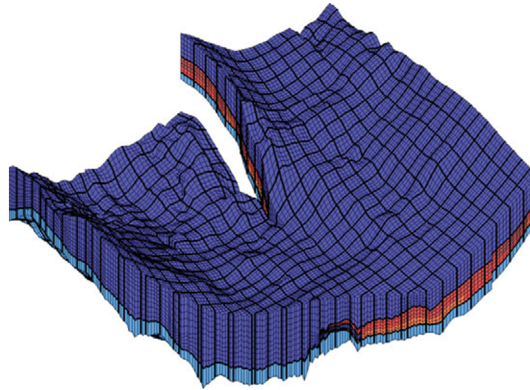


Figure 14.4 A 4 × 4 × 11 coarsening of the NPD5 model of the Johansen aquifer that preserves the Amundsen, Dunlin, and Johansen formations.

therefore coarsen the three formations separately, using permeability values K as indicator as shown in Table 14.1. Likewise, to correctly resolve the formation and migration of the thin plume, it is essential that the grid has as high vertical resolution as possible. Unless we use a vertically integrated model (as in MRST co2lab [19]), we would normally only reduce the lateral resolution, say by a factor of four in each lateral direction. Here, however, we first use only a single block in the vertical direction inside each formation to more clearly demonstrate how the coarsening can adapt to the individual formations.

Assuming that the grid G and the permeability field K have been initialized properly as described in Section 2.5.4, the coarsening procedure reads

```
pK = 2*ones(size(K)); pK(K<=0.1) = 3; pK(K<=0.01)= 1;
pC = partitionUI(G, [G.cartDims(1:2)/4 1]);
[b,i,p] = unique([pK, pC], 'rows');
p = processPartition(G,p);
CG = generateCoarseGrid(G, p);
plotCellData(G,log10(K),'EdgeColor','k','EdgeAlpha',.4); view(3)
outlineCoarseGrid(G,p,'FaceColor','none','EdgeColor','k','LineWidth',1.5);
```

Figure 14.4 shows the coarse grid obtained by intersecting the partition vector pC, which has only one block in the vertical direction, with the partition vector pK that represents the

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

different formations. In regions where all formations are present, we get three blocks in the vertical direction. In other regions, only the Dunlin and Amundsen shales are present and we hence have two blocks in the vertical direction.

The aquifer model contains one major and several minor faults. As a result, 1.35% of the cells in the original grid have more than six neighbors. Coarsening a model with irregular geometry (irregular perimeter, faults, degenerate cells, etc.) uniformly in index space will in most cases give many blocks with geometry that deviates quite a lot from being rectangular. The resulting coarse grid thus contains a larger percentage of unstructured connections than the original fine model. For this particular model, 20.3% of the blocks have more then six coarse faces. If we look at a more realistic coarsening that retains the vertical resolution of the original model, 16.5% of the blocks have more than six neighboring connections. This model is obtained if we repeat the earlier construction using

```
pC = partitionUI(G, G.cartDims./[4 4 1]);
```

Figure 14.5 shows six different coarse blocks sampled from the top grid layer of the Dunlin shale. Block number one is sampled from a part of the aquifer perimeter that does not follow the primary grid directions and thus has irregular geometry. The other five blocks contain (parts of) a fault and will therefore potentially have extra connections to blocks in grid layers below. Despite the irregular geometry of the blocks, the coarse grid can be used directly with most of the solvers discussed earlier in the book. In our experience, the quality of the coarse solution is generally more affected by the quality of the upscaling of the petrophysical parameters (see Chapter 15) than by the irregular block geometry. In fact, irregular blocks that preserve the geometry of the fine-scale model respect the layering and connections in the fine-scale geology and therefore often give more accurate results than a coarse model with more regular blocks if upscaled correctly.

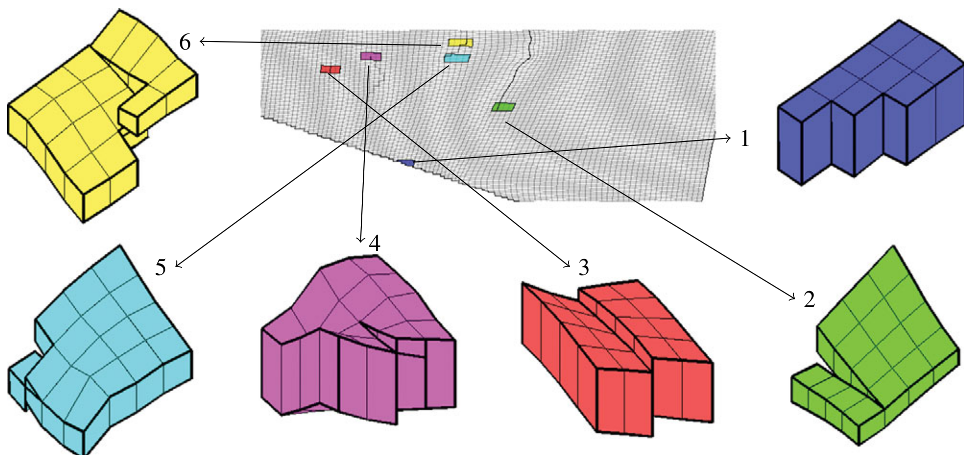


Figure 14.5 Six coarse blocks sampled from the top grid layer of the Dunlin formation in a $4 \times 4 \times 1$ coarsening of the NPD5 sector model of the Johansen formation.

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

14.3.2 The SAIGUP Model

The SAIGUP model, introduced in Section on page 97, has six user-defined rock types (also known as saturation regions; see Figure 2.20) that are used to specify different rock-fluid behavior (relative permeability and/or capillary pressure functions). Depending upon the purpose of the reduced model, we may want to preserve these rock types using the same type of technique as described in the previous example. This has the advantage that if each coarse block is made up of one rock type only, we would not have to upscale the rock-fluid properties. On the other hand, this typically leads to coarse grids with (highly) irregular block geometries and large variations in block volumes. To illustrate this point, we start by partitioning the grid uniformly into $6 \times 12 \times 3$ coarse blocks in index space:

```
p = partitionUI(G,[6 12 3]);
```

This introduces a partition of all cells in the logical $40 \times 120 \times 20$ grid, including cells that are inactive. To get a contiguous partition vector, we remove blocks that contain no active cells, and then renumber the vector. This reduces the total number of blocks from 216 to 201. Some of the blocks may contain disconnected cells because of faults and other nonconformities, and we must therefore postprocess the grid in physical space and split each disconnected block into a new set of connected sub-blocks:

```
p = compressPartition(p);
p = processPartition(G,p);
```

The result is a partition with 243 blocks that each consists of a set of connected cells in the fine grid. Figure 14.6 shows an explosion view of the individual coarse blocks. Whereas all cells in the original model are almost exactly the same size, the volumes of the coarse blocks span almost two orders of magnitude. In particular, the irregular boundary near the crest of the model introduces small blocks consisting of only a single fine cell in the lateral direction. Large variations in block volumes will adversely affect any flow solver if we later run a flow simulation on the coarsened model. To get coarse blocks with a more even size distribution, we therefore pick the smallest blocks and merge them with the neighbor that has the largest block volume. We repeat this process until the volumes of all blocks are above a prescribed lower threshold.

The merging algorithm is quite simple: we compute block volumes, select the block with the smallest volume and merge this block with one of its neighbors. Next, we update the partition vector by relabeling all cells in the block with the new block number and compress the partition vector to get rid of empty entries. Finally, we regenerate a coarse grid, recompute block volumes, pick the block with the smallest volume in the new grid, and repeat the process. In each iteration, we plot the selected block and its neighbors:

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

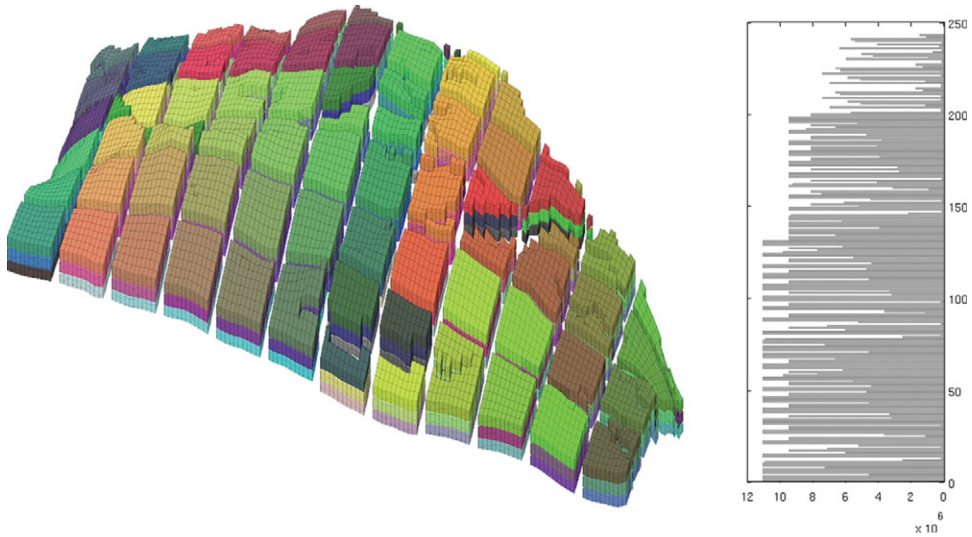
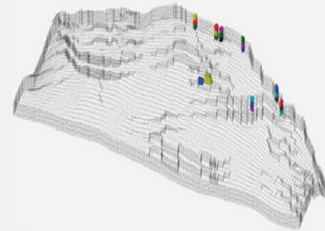


Figure 14.6 Logically Cartesian partition of the SAIGUP model. The plot to the left shows an explosion view of the individual blocks colored with `colorcube`. The bar graph to the right shows the volumes in units [m^3] for each of the blocks in the partition.



```
blockVols = CG.cells.volumes;
meanVol   = mean(blockVols);
[minVol, block] = min(blockVols);
while minVol<.1*meanVol

  % Find all neighbors of the block
  clist = any(CG.faces.neighbors==block,2);
  nlist = reshape(CG.faces.neighbors(clist,:),[],1);
  nlist = unique(nlist(nlist>0 & nlist~=block));
  plotBlockAndNeighbors(CG, block, ...
     'PlotFaults', [false, true], 'Alpha', [1 .8 .8 .8]);

  % Merge with neighbor having largest volume
  [~,merge] = max(blockVols(nlist));

  % Update partition vector
  p(p==block)  = nlist(merge);
  p = compressPartition(p);

  % Regenerate coarse grid and pick the block with the smallest volume
  CG = generateCoarseGrid(G, p);
  CG = coarsenGeometry(CG);
  blockVols = CG.cells.volumes;
  [minVol, block] = min(blockVols);
end
```

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

To find the neighbors of a given block, we first select all connections that involve block number `block`, which we store in the logical mask `clist`. We then extract the indices of the blocks involved in these connections by using `clist` to index the connection list `CG.faces.neighbors`. The block cannot be merged with the exterior or itself, so values 0 and `block` are filtered out. A pair of blocks may generally share more than one connection, so `unique` is used to remove multiple occurrences of the same block number.

When selecting which neighbor a block should be merged with, there are several points to consider from a numerical point of view. We typically want to keep blocks as regular and uniformly sized as possible, make sure that the cells inside each new block are well connected, and limit the number of new connections we introduce between blocks. Figure 14.7 shows several of the small blocks and their neighbors. In the algorithm shown in the most recent code box, we have chosen a simple merging criterion: each block is merged with the neighbor having the largest volume. In iterations one and three, the blue blocks are merged with the cyan blocks, which is probably fine in both cases. However, if the algorithm later wants to merge the yellow blocks, using the same approach may not give good results as shown in iteration ten. Here, it is natural to merge the blue block with the cyan block that lies in the same geological layer (same K index) rather than merging it with the magenta block that has the largest volume. The same problem is seen in iteration number four, where the blue block is merged with the yellow block, which is in another geological layer and only weakly connected to the blue block. As an alternative, we could choose to merge with the neighbor having the smallest volume, in which case the blue block would be merged with the yellow block in iterations one and three, with the magenta block in iteration four, and with the cyan block in iteration six. This would tend to create blocks consisting of cells from a single column in the fine grid, which may not be what we want. Another solution would be to merge across the coarse face having the largest transmissibility (sum of fine-scale transmissibilities).

Altogether, we see that there are several issues we need to take into consideration and these could potentially lead to mutually conflicting criteria. This obviously makes it difficult to devise a good algorithm that merges blocks in the grid in a simple and robust manner. A more advanced strategy could, for instance, include additional constraints that penalize merging blocks belonging to different layers in the coarse grid. Likewise, you may want to avoid creating connections that only involve small surface areas. However, such connections may already be present in the coarsening we start from, as illustrated by the yellow and cyan blocks in the lower-right plot in Figure 14.7, which were created when partitioning the grid uniformly in index space.

The basic `processPartition` routine only checks that there is a connection between all cells inside a block. A more sophisticated routine could obviously also check the face areas associated with each connection, or the magnitude of the associated transmissibility, and consider different parts of the block to be disconnected if the area or transmissibility that connects them is too small. As a step in this direction, we could consider the face area when we postprocess the first uniform partition, e.g., do something like the following,

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

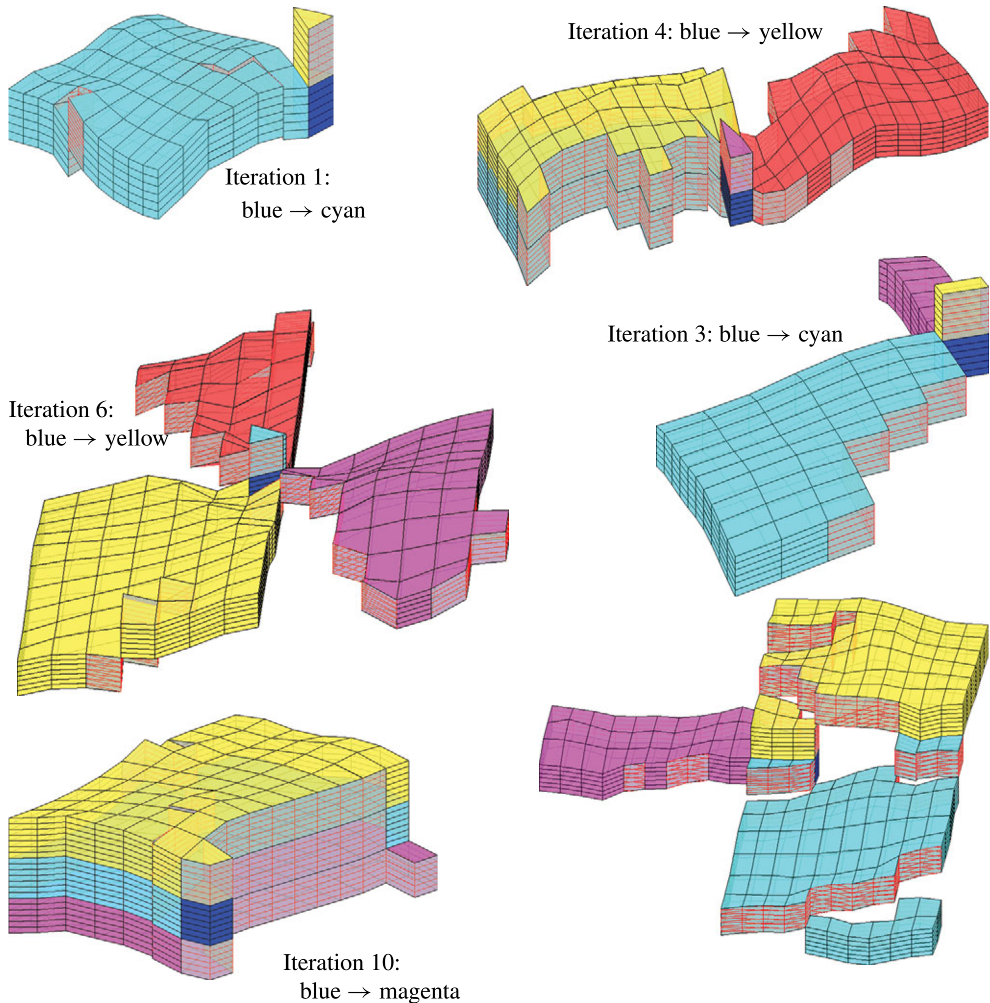


Figure 14.7 Examples of small blocks (in blue) being merged with one of their neighbors shown in semitransparent color with fault faces in gray.

```
p = partitionUI(G,[6 12 3]);
p = compressPartition(p);
p = processPartition(G, p, G.faces.areas<250);
```

This increases the number of blocks in the final grid obtained after merging small blocks from 220 to 273, but avoids constructing blocks looking like the yellow and cyan block in the lower-right plot in Figure 14.7. On the other hand, this approach obviously involves a threshold parameter that will vary from case to case and has to be set by an expert. The

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

result may also be very sensitive to the choice of this parameter. To see this, you can try to redo the initial partition with threshold values 300 and 301 for the face areas.

14.3.3 Near Well Refinement for CaseB4

In this example[1], we consider the smallest $36\times48\times12$ pillar grid from CaseB4 (see Section 3.3.1) describing a reservoir section with two intersecting dip-slip faults. The reservoir is known to have four distinct geological layers with k indices 1, 2–7, 8–11, and 12. We start by making a standard load-balanced partition in index space:

```
p0 = partitionUI(G, [7 9 1]);
pf = processPartition(G, p0, find(G.faces.tag==1));
```

In the second line, we identify all fault faces and ask the processing routine to remove any connections across these faces when examining whether any of the 63 original blocks are disconnected or not. Blocks that are fully penetrated by a fault will contain at least two connected components and will hence be split. After splitting, the partition has 79 blocks. Although not needed for this model, it is quite simple to also split blocks that are only partially penetrated by faults; see [187] for more details. The disadvantage of imposing hard constraints like fault faces on the partitioning process is that it tends to increase the variation in block sizes, as shown in Figure 14.8.

Pressure gradients and flow rates are usually much larger near wells than inside the reservoir. How accurate we are able to capture the flow solution in the near-well region strongly influences the accuracy of the overall simulation. It may therefore be desirable to have higher grid resolution in the near-well zone than inside the reservoir. To refine blocks near the two wells, we use `refineNearWell` from the `coarsegrid` module, which takes a set of points and partitions these into cylindrical sectors around a single point in the xy-plane. The first well is placed in the corner of the reservoir section and we partition the perforated well block into five radial sections. The width of the radial sections is set to decay as the logarithm of the radial distance:

```
[wc,p] = deal( W(i).cells(1), pf);       % Pick well cell in top layer
wpt    = G.cells.centroids(wc,:);        % Center point of refinement
cells  = (p == p(wc));                   % All cells in block
pts    = G.cells.centroids(cells,:);     % Points to be repartitioned
out    = refineNearWell(G.cells.centroids(cells,:), wpt, ...
              'angleBins', 1, 'radiusBins', 4, ...
              'logbins', true, 'maxRadius', inf);
p(cells) = max(p) + out;                 % Insert new partition in block
p  = compressPartition(p);               %   and compress vector
```

The second well is perforated in the middle of a coarse block. It is therefore natural to refine both in the radial and the angular directions. The number of blocks in the angular

[1] Complete code: `coarsenCaseB4.m` in the `book` module.

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

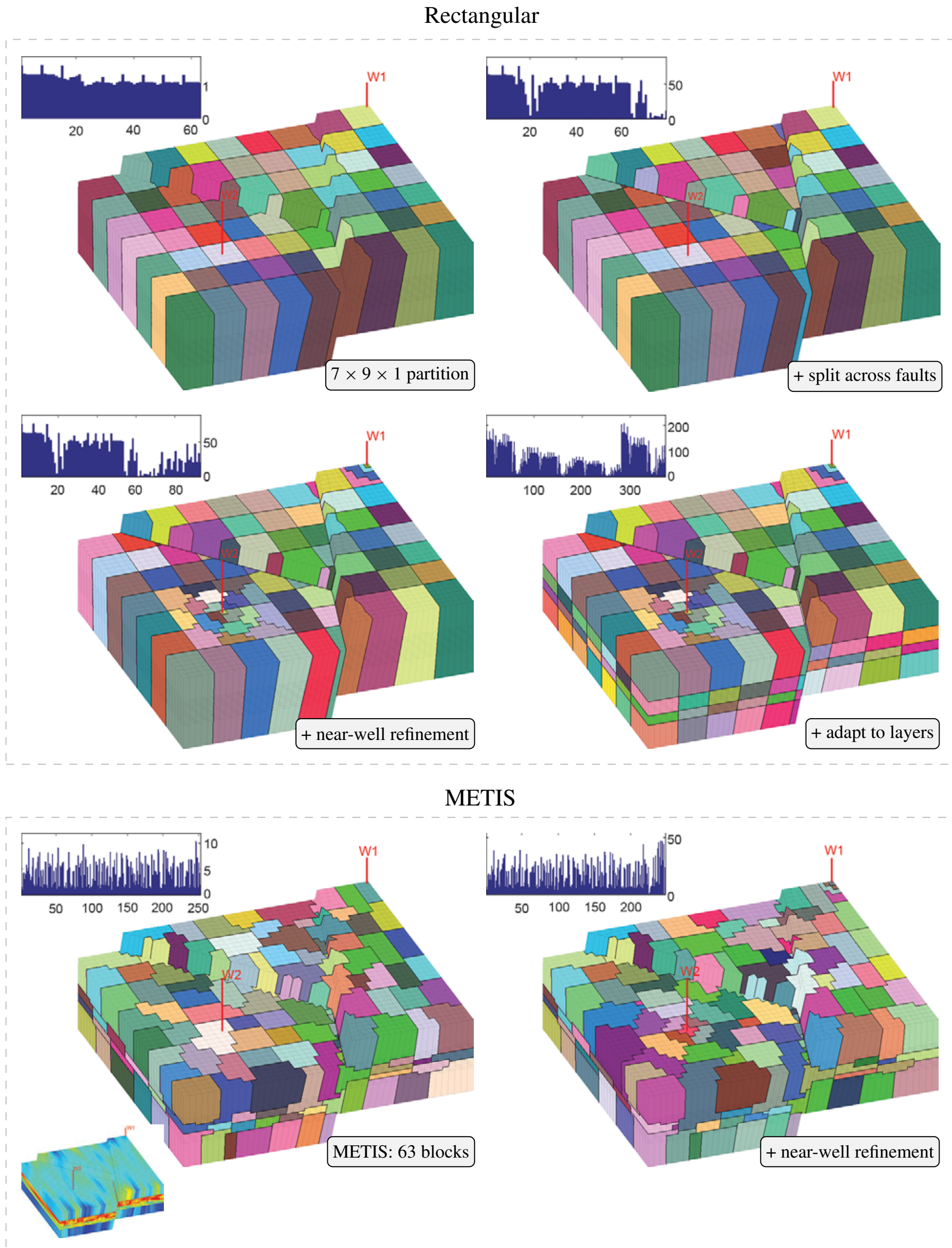


Figure 14.8 Partitioning of the CaseB4 pillar grid. The rectangular partition starts by a call to `partitionUI`, followed by the splitting of blocks across faults, near-well refinement by `refineNearWell`, and finally, vertical partition following geological layers. The METIS partition starts by a graph partitioning with transmissibility as edge weights followed by a near-well refinement. The large plots show how the partition is gradually refined by including new partition principles. The small plots show the variation in block volumes relative to the volume of the smallest block.

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

direction can be set for each radial section. Here, we therefore let the number of blocks increase outward, starting with a single block in the innermost layer, and then four, six, and nine blocks in the next three layers. To be able to place such a refinement, we repartition not only the well block but also the eight surrounding neighbors. To find these blocks, we construct a temporary coarse grid structure and use the topology from `CG` to construct an adjacency matrix similar to the one we discussed on page 522.

```
% Adjacency matrix
CG = generateCoarseGrid(G, pw);
N  = getNeighbourship(CG);
A  = getConnectivityMatrix(N,true,CG.cells.num);

% Find all axial/diagonal neighbors
rblk  = zeros(CG.cells.num,1);
rblk(pwv)=1;
rblk((A*A*rblk)>1) = 1;
cells = rblk(pw)>0;
```

$$r = \begin{array}{|c|c|c|c|c|}\hline & & & & \\ \hline & & & & \\ \hline & & 1 & & \\ \hline & & & & \\ \hline & & & & \\ \hline\end{array}$$

$$Ar = \begin{array}{|c|c|c|c|c|}\hline & & & & \\ \hline & & 1 & & \\ \hline & 1 & 1 & 1 & \\ \hline & & 1 & & \\ \hline & & & & \\ \hline\end{array}$$

$$A^2r = \begin{array}{|c|c|c|c|c|}\hline & & 1 & & \\ \hline & 2 & 2 & 2 & \\ \hline 1 & 2 & 5 & 2 & 1 \\ \hline & 2 & 2 & 2 & \\ \hline & & 1 & & \\ \hline\end{array}$$

We start by defining an indicator vector `rblk` equal one in the well block and zero in all other blocks. Multiplying by the adjacency matrix `A` will set the value in each block equal the sum over the block and its face-neighbors. After one multiplication, `rblk` therefore has ones in all the face-neighbors, and after two multiplications all blocks surrounding the initial block should have value larger than one. Next, we reset `rblk` to one in all blocks with `A*A*rblk > 1` and index the resulting vector by the partition vector `pw` to extract all cells inside these blocks. These cells can be repartitioned as for the first well:

```
out   = refineNearWell(pts, wpt, 'angleBins', [1,4,6,9],...
        'radiusBins', 4, 'logbins', true, 'maxRadius', inf);
```

Altogether, this increases the number of coarse blocks to 93. Finally, we can impose vertical refinement that follows the four geological layers. Assuming the indices of these layers are given as an indirection map, this partition is constructed as follows:

```
% Here: layers = [1 2 8 12 13]
pK = rldecode((1:numel(layers)-1)',diff(layers));
[~,~,k]=gridLogicalIndices(G);
pk = compressPartition((pK(k)-1)*max(pw)+pw);
pk = processPartition(G, pk);
```

The resulting partition shown in Figure 14.8 has 372 blocks and significant variation in block sizes. This does not necessarily make the coarse grid ill-suited for flow simulations, but in a more careful implementation it would have been natural to postprocess the grid to get rid of small blocks away from the near-well regions that can be merged with their neighbors.

As an alternative to the regular partition in index space, we can use a standard software like METIS [154], which offers state-of-the-art partitioning of graphs and meshes based on multilevel recursive-bisection, multilevel k-way, and multi-constraint partitioning

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

algorithms. METIS generally tries to make blocks that are as uniform as possible and at the same time have as few connections as possible. If we let fine-scale transmissibilities measure connection strengths between cells for the edge-cut minimization algorithm, the software tries to construct blocks without crossing large permeability contrasts. The `coarsegrid` module implements a wrapper that greatly simplifies the task of calling METIS. The following call gives a partition with approximately the same number of blocks as in the structural partition with four vertical layers, just discussed:

```
mrstModule add incomp
hT = computeTrans(G, rock);
p0 = partitionMETIS(G, hT, 7*9*4, 'useLog', true);
```

The partitioning function uses the standard `incompTPFA` solver from the `incomp` module to set up a fine-scale discretization matrix that describes how cells are connected. Here, we take the logarithm of the transmissibilities before the connection strength is constructed. Experience shows that this gives better partitions when transmissibilities vary several orders of magnitude locally. The resulting partition is shown in the lower-left plot in Figure 14.8. Compared with the rectangular partition, the grid blocks are more irregular and have more connections and larger variation in block sizes. Also for this partition, we can introduce near-well refinement exactly as described in this subsection. The first well falls within a smaller grid block and hence gives slightly different local blocks. For the second well, our detection of neighboring blocks selects a larger region to repartition and hence `refineNearWell` is able to generate a refinement that looks more circular compared our first partition method.

A technical note: To inform the system where to find METIS, we use a global variable `METISPATH` that can be set in your `startup_user` function. If you, like me, use Linux and have installed METIS in, e.g., `/usr/local/bin`, you add the following line to your `startup_user.m` file:

```
global METISPATH; METISPATH = fullfile('/usr','local','bin');
```

Computer exercises

14.3.1 Rerun CaseB4 with the stair-stepped model instead. Why does the code not work? (Hint: check the number of active cells.) Can you make a partition that splits blocks properly on opposite sides of faults?

14.3.2 The code example also contains a set up that tries to force METIS to split blocks across faults, which does not work well. Can you develop an alternative strategy to ensure that fault faces are also preserved in the METIS partition? (Hint: try to partition each fault block separately.)

14.3.3 Try to implement the merging technique from the SAIGUP example to reduce the large difference in cell volumes. (Hint: make sure that you do not merge any blocks in the near-well regions.)

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

14.4 More Advanced Coarsening Methods

The previous section outlined two main types of partitioning methods. Systematic graph partitioning algorithms like those implemented in METIS are widely used and very robust, but do not necessarily give you the desired control over the coarsening process. In particular, it can be quite difficult to rigorously formulate the desired coarsening as a set of cost functionals and constraints that are well posed and computationally tractable. More ad hoc and straightforward approaches that explicitly impose a structured partition in index space or use features from the model as partition vectors (Section 14.1.3) leave you in full control, but can easily lead to undesired artifacts like large variations in block volumes as we saw for SAIGUP and CaseB4. For large and complex models one cannot rely on visual quality control and manual repair of individual blocks. A significant body of research has therefore been devoted to develop automated or semiautomated coarsening procedures that incorporate certain features of the geology or flow physics.

Most methods reported in the literature try to generate a coarser and more optimal grid consisting of standard grid blocks with spatially varying resolution. The size of each grid block is usually determined by equilibrating a density measure or by introducing a background indicator whose variation is minimized within blocks and maximized between blocks. In particular, it is common to impose constraints on the geometrical shape of the new grid blocks (Delaunay/Voronoi) and the degree to which they align with the cells in the original grid [257, 47, 100]. Density measures and indicators can be defined using geological quantities like permeability [116, 160]; flow-based quantities like fluid velocity [101], vorticity [201], or streamlines [301, 72, 59, 310, 126]; local a priori error measures [162]; or statistical or a posteriori goal-oriented error indicators that measure how a particular point influences the error in production responses or other predefined quantities. Flow-based coarsening has been shown to be a powerful approach in combination with upscaling, and has been developed both for structured and unstructured grids. The basic goal of flow-based griding [93] is to introduce higher resolution in regions of high flow and coarser resolution in regions of lower flow.

An alternative approach is to generate coarse blocks by *agglomerating* cells that are similar in a sense prescribed by the user. This gives coarse blocks with complex polyhedral forms that follow the geometries of the original fine grid. An early approach in this direction [3] was aimed at coarse-scale discretization of transport equations and suggested to group cells according to the magnitude of the velocity (or flux) field so that each coarse block consists of a collection of cells through which the flow has approximately the same magnitude. Later research has shown that the original *nonuniform coarsening method* is just a special case of a much wider set of heuristic methods for producing coarse partitions that adapt to various types of geological or flow constraints. These methods have been developed for three different purposes: (i) to increase the accuracy of multiscale pressure solvers by adapting to fine-scale geological features [5, 6, 222, 194, 17, 191]; (ii) to define partitions that preserve rock types or relperm regions and thereby reduces the need for upscaling; and (iii) to find flow-adapted partitions suitable for coarse-scale transport solvers [3, 125, 124].

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

These methods can, in turn, be incorporated into a more general framework that tries to impose various types of geological/flow constraints in a hierarchical manner [187]. Similar aggregation-based methods have been applied to fractured media [153, 138]. The `agglom` module implements a set of modular components that can be combined in different ways to create various types of agglomerated partitions as outlined in the next section.

14.5 A General Framework for Agglomerating Cells

This section outlines the general agglomeration framework implemented in the `agglom` module. The agglomeration process can be viewed as governed by three rules:

- a neighbor definition that gives the *permissible directions* one can search for new cells to agglomerate;
- a set of indicator functions that determine the *feasible directions* among all the permissible directions;
- one or more indicators to determine how far the agglomeration should proceed, i.e., indicators that can be used to locally determine the size of each grid block.

Permissible directions are given by the grid topology and will in the default setup consist of all face neighbors. However, the permissible directions can also be extended to include cells sharing a common edge/node or cells whose centroids lie within a prescribed geometric distance. Likewise, the permissible directions can be restricted locally so that blocks cannot be agglomerated across faults or boundaries between different facies types, initialization regions, saturation regions, etc. The *feasible directions* are defined through a set of cell-based indicator functions that may contain geological or petrophysical features, flow-based quantities, or general user-set expert knowledge. One can also impose additional rules that prevent the total indicator function from growing too large within blocks, try to regularize the outline of the blocks, minimize aspect ratios. Altogether, this gives a very flexible framework that can be used to construct many different types of grids.

To realize this framework, the `agglom` module implements a set of source functions that create new partition vectors, and a set of filter functions that take one or more partitions as input and create a new partition as output. Examples of filter operations include combining/intersecting multiple partition vectors; performing sanity checks and modifications to ensure that no blocks are disconnected or contained within other blocks; or modifying the partition vector by merging small blocks or splitting large blocks.

14.5.1 Creating Initial Partitions

You have already seen several examples of topological partitions, e.g., generated by `partitionUI` from the `coarsegrid` module. Another way to create initial partitions is to segment indicator functions representing the heterogeneity of the medium. Examples of such indicator functions include the logarithm of permeability, velocity magnitude,

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

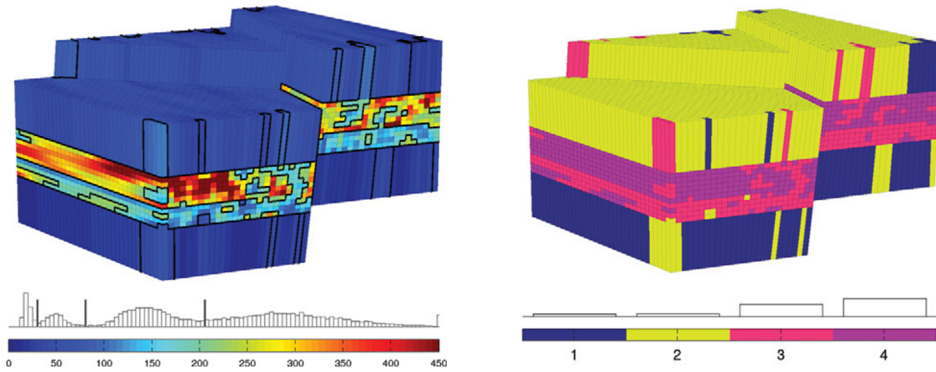


Figure 14.9 Segmentation with permeability as indicator for CaseB4 with four lognormally distributed layers (left). The right plot shows the resulting bins.

time-of-flight or residence time, vorticity, etc. Assuming we have an indicator function `I` that takes a single value in each cell of a grid `G`, the indicator can be segmented into bins by calling

```
p = segmentIndicator(G, I, bins)
```

Constructing indicator functions and making initial partitions will typically involve some kind of expert knowledge, and hence the MRST module does not offer specific routines. However, the tutorials contain several examples that should be instructive. To illustrate this function, let us revisit the CaseB4 example from Section 14.3.3 and see if we can use this function with permeability as an indicator to automatically detect the four geological layers in the model. The permeability is lognormal within each layer. The histogram of permeability values in Figure 14.9 has four distinct peaks, but the different layers are not clearly separated. We thus pick limits for the bins in the local minima of the distribution (`K` is given in md):

```
p = segmentIndicator(G, K, [0 30 80 205 inf], 'split', false);
```

Here, we have asked the function to only output the segmented bins. These bins, shown to the right in Figure 14.9, do not follow the layers exactly since permeability values are not unique to a single layer. If it is important to preserve the layering exactly, we would be better off by using the k index for the initial partition. In other cases like the SPE 10 model, all we have to bin different rock types are the permeability and/or porosity values.

14.5.2 Connectivity Checks and Repair Algorithms

Partitions defined by segmenting indicator functions usually give bins that consist of multiple connected components. These can be split by the function `processPartition` as discussed in Section 14.1.2. This function is called by default from within

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

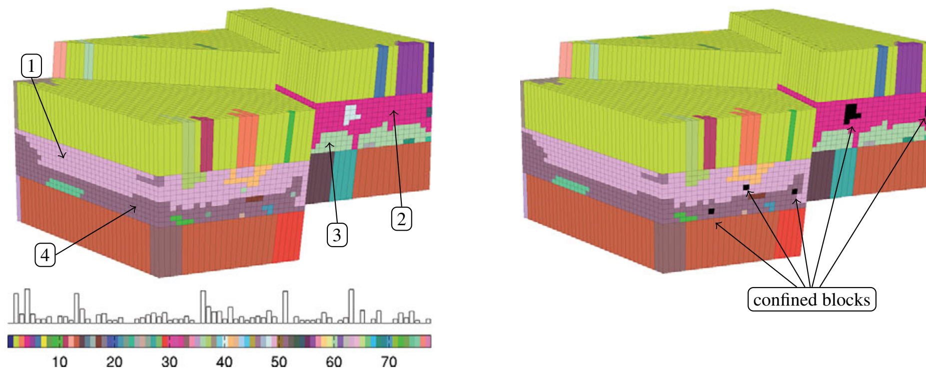


Figure 14.10 The left plot shows coarse blocks from the initial segmentation of CaseB4; the histogram of cells per block has logarithmic y-scale. Blocks 1,2 and 3,4 are disconnected components from layers two and three, respectively. The right figure shows isolated blocks that are completely confined within another block.

`segmentIndicator`, and if we drop the optional `'split'` flag from the call, the initial bins are split so that they form blocks satisfying MRST's basic requirements for a coarse grid. Figure 14.10 shows that the resulting partition consists of six large coarse blocks and a number of smaller ones. Layers two and three are not fully connected across all faults and the corresponding bins have therefore each been split in two large blocks (1 and 2, and 3 and 4 in the plot). The remaining small blocks are the result of permeability variations inside the layers.

Blocks that are entirely confined within other blocks only have a single interface. When the resulting grid is used for flow simulation, the only flow that can pass the interface between the block and its surrounding neighbor must be caused by source terms or compressible effects. This tends to create numerical artifacts unless the block contains a well. For incompressible flow, in particular, the corresponding block interface will act as an internal no-flow boundary. An additional sanity check should therefore be performed to detect such blocks:

```
cb = findConfinedBlocks(G, p);
[cb,p] = findConfinedBlocks(G,p);
```

The first call just outputs a list of confined blocks, whereas the second also merges these blocks with their surrounding neighbor. The right plot in Figure 14.10 illustrates confined blocks for CaseB4.

The detection algorithm is somewhat simplified and the function cannot handle nested cases properly. In the case of recursively confined blocks, only the outermost and the innermost blocks are listed in `cb`. To find a more general solution, we can look to algorithms for detecting *biconnected components* from graph theory. Let me explain the idea by a simple example: Figure 14.11 shows a 7×7 grid partitioned into six coarse blocks together

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

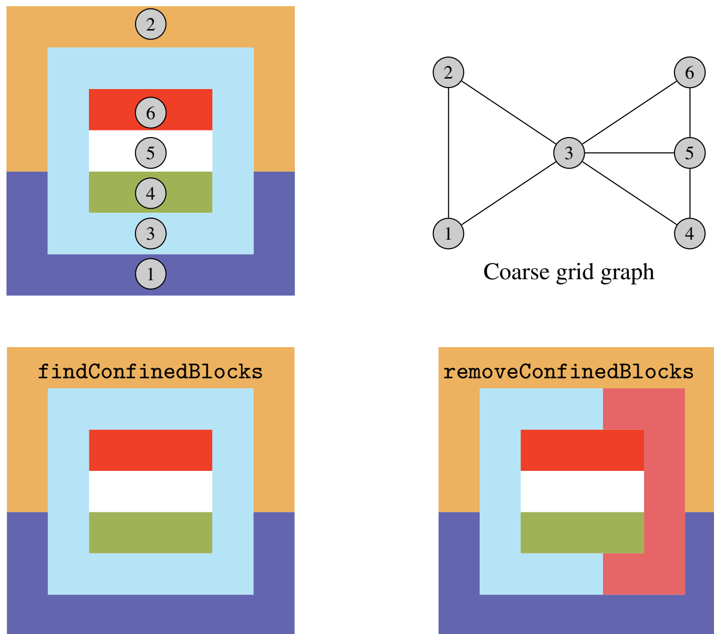


Figure 14.11 Detection and repair of recursively confined blocks by use of functionality from the MATLAB Boost Graph Library. The simple `findConfinedBlocks` function fails to detect and repair cases where a set of blocks are confined within another block.

with the corresponding coarse-grid graph. This graph is not biconnected: node 3 belongs to two connected components (1–2–3 and 3–4–5–6), and if this node is removed, the graph becomes disconnected. The function

```
pn = removeConfinedBlocks(G, p)
```

merges single confined blocks with their surrounding blocks. In addition, it uses functionality from the MATLAB Boost Graph Library[2] to determine whether a graph is biconnected or not and to find the corresponding components of the graph. If the graph is not biconnected, any grid block that completely surrounds a set of other blocks is split in two (along a plane orthogonal to the x-axis through the block center). The resulting partition is not necessarily singly connected and may have to be processed by a further call to `processPartition`.

14.5.3 Indicator Functions

The agglomeration framework relies on two indicator functions, which we for simplicity refer to as a *volume indicator* and a *flow indicator*. We assume that each indicator function

[2] This software is freely available under a BSD License. The `matlab_bgl` module has a function `downloadMBGL` that downloads and installs this library. This is one of the few examples of external dependencies in MRST.

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

takes value $I(c_i)$ in cell c_i and that these values can be interpreted as densities, i.e., that they are positive, additive, and normalized by the cell volume $|c_i|$. As for the cells, we associate indicator values $I(B_j)$ to each block, defined as the volumetric average of the cell indicator values within the block (or as the arithmetic average if I is not a density).

The volume indicator is only used to determine block volumes and is either set to be unity for bulk volumes and ϕ for pore volumes. The flow indicator can in principle be any user-defined quantity, but typical examples include quantities derived from permeability, time-of-flight or residence time, velocity or flux, vorticity, and so on (see discussion in [124]). Error indicators or sensitivities can also be used as flow indicators. Specific examples are given later.

To understand the role of these two indicators, let us look at four general principles you can use to control how cells are grouped together:

1. The variation of the flow indicator should be minimized inside each block. That is, each block should be as homogeneous as possible so that cells with high and low permeabilities, large and small fluxes, high and low travel times, and so on, are separated in different blocks. Likewise, cells from different facies, deposition environments, flow units, initialization regions, relperm/capillary regions, and so on, should preferably not be agglomerated into the same block. Algorithmically, this means that the flow indicator is used to pick the most feasible among the permissible neighbors of a block when adding a new cell to a growing block.
2. The integral of the flow indicator should be equilibrated over blocks. You can use this principle refine resolution in regions with high flow, low residence time, high error or sensitivity indicator, and so on. Full equilibration will in general require rigorous optimization. Algorithms in the `agglom` framework instead try to ensure that the integral of the flow indicator is within prescribed upper bounds. These bounds determine when to stop agglomerating more cells into a block, or alternatively, determine when blocks are too big and must be split.
3. The size of each block should be with certain upper and lower bounds. Algorithmically, upper limits on block sizes are imposed by the upper bound on the flow indicator, whereas lower bounds are imposed through the volume indicator when determining the blocks that should potentially be merged with their neighbors.
4. Blocks should not have too-large aspect ratios and be as regular as possible. Regularity is determined in part by the source functions generating initial partitions, and in part by filter functions that split or merge individual blocks. Controlling regularity may be difficult in ad hoc algorithms, and if regularity is important, you may be better off using METIS.

14.5.4 Merge Blocks

We have already seen how manually constructed partitions can give grids with very large variations in block sizes. Variations in block sizes can be even more pronounced if the initial partition is created by segmenting an indicator function. One method to remedy the

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

problem would be to loop through all blocks in the partition and try to merge each small blocks with one of its neighbors. Our simple implementation in the SAIGUP example from Section 14.3.2 was not very efficient as it required generation of a new coarse grid structure in each iteration. Our choice of merging with the neighbor having largest volume was also somewhat ad hoc.

In the agglomeration framework, we use the volume indicator I_v to measure the size of cells and blocks with $I_v \equiv 1$ for bulk volume and $I_v = \phi$ for pore volume. (The latter definition can also be modified to include net-to-gross or other factors that affect the pore volume.) Let $G = \{c_i\}_{i=1}^n$ denote the whole grid consisting of n cells that each has a bulk volume $|c_i|$. Then, block $B_j = \{c_i \mid p(c_i) = j\}$ should be merged with a neighbor if it violates the following lower bound

$$I_v(B_j)\,|B_j| \geq n_l \bar{I}_v, \qquad \bar{I}_v = \frac{1}{n}\sum_{i=1}^{n} I_v(c_i)\,|c_i|, \tag{14.1}$$

where $n_l \geq 1$ is a prescribed constant and $I_v(B_j)|B_j| = \sum_{c_i \in B_j} I_v(c_i)\,|c_i|$. In other words, block B_j should be merged if its integrated indicator value is less than n_l times the average volume indicator value for all cells. To determine which block to merge with, we use the flow indicator I_f and merge block B_j with the block among block B_j's permissible neighbors $\mathcal{N}(B_j)$ that has the closest indicator value, i.e.,

$$\tilde{B} = \text{argmin}_{B \subset \mathcal{N}(B_j)} |I_f(B) - I_f(B_j)|. \tag{14.2}$$

This is implemented in the function

```
q = mergeBlocks(p, G, Iv, Ifl, NL, 'static_partition', ps)
```

which takes a partition `p`, a volume indicator `Iv`, and a positive flow indicator `Ifl` and merges all blocks that violate (14.1). In doing so, each merged block is agglomerated into the neighboring block that has the closest I_f value. Optionally, the merging operation can be set to respect a static partition `p2` so that no block ends up having different `ps` values.

The problem with this method is that it does not impose any control on the upper size of the merged blocks. In particular, merging with the nearest flow indicator tends to work against the principle that flow indicators should be equilibrated across all the grid blocks. The idea[3] of the flow indicator is that it should enable us to aggressively coarsen the grid in regions of low flow or regions that has limited impact on the overall simulation result, while keeping higher resolution in regions of high flow.

To provide upper control on the size of the blocks, we introduce a second requirement, this time on the flow indicator:

$$I_f(B)\,|B| \leq n_u \bar{I}_f, \tag{14.3}$$

[3] In principle, the flow indicator need not have anything to do with flow; I still think it is important that you know the underlying philosophy so that you can design indicators so that the merging/refinement process works as you desire.

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

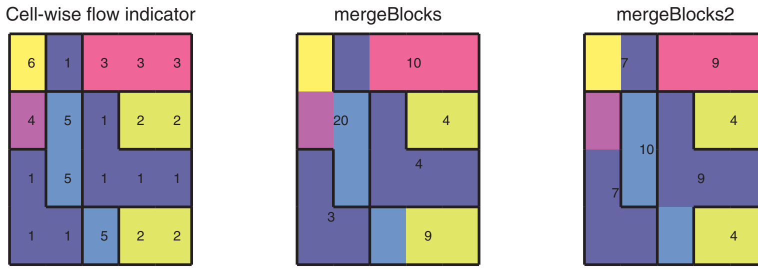


Figure 14.12 Comparison of the two merging algorithms for a simple 5 × 4 grid. The left plot shows the initial partition and the flow indicator I_f per cell. The middle and right plot shows partitions after merging, with numbers giving the integrated flow indicator, which according to (14.3) should not exceed 7.5.

where n_u is a prescribed parameter and $\bar{I}_f$ is the flow indicator averaged over all cells, defined exactly the same way as $\bar{I}_v$. The second merging function

```
q = mergeBlocks2(p, G, Iv, Ifl, NL, NU)
q = mergeBlocks2(p, G, Iv, Ifl, NL, NU, 'nblock', NB, 'cfac', cfac)
```

not only provides a more efficient implementation of the merging process, but also imposes (14.3) as a *soft constraint*, meaning that the routine tries its best to avoid violating this condition, but if no other options are available, it merges blocks even if the flow indicator of the new block becomes too high. It is also possible to specify an optional relative factor `cfac` at which (14.3) is turned into a *hard constraint*. Likewise, the parameter `NB` gives an upper bound on the number of cells that can be agglomerated into a single block. (Notice also that this routine does *not* support static partitions.)

Figure 14.12 compares the two methods for a small 5 × 4 grid, called with the parameters

```
p1 = mergeBlocks(p, G, ones(n,1), I, 2);
p2 = mergeBlocks2(p, G, ones(G.cells.num,1), I, 2, 3);
```

Here, `mergeBlocks` identifies four single-cell blocks that are merged with their neighbor with closest value: 4→6, 5→(2,2), (4,6)→(5,5), and 1→(3,3,3). The result is three new blocks that all violate (14.3). The alternative routine merges 4→(1,1,1) and 6→1, which both are acceptable. For cell with value 5, the only option is to merge with (1,1,1,1), even if this violates (14.3).

14.5.5 Refine Blocks

The next natural step would now be to refine blocks that violate the upper bound (14.3). You can do this by invoking any of the two calls

```
p = refineBlocks(p, G, Ifl, NU, @refineUniform, 'cartDims',[Nx Ny Nz])
p = refineBlocks(p, G, Ifl, NU, @refineGreedy)
```

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

The first call refines blocks by encapsulating the corresponding cell indices in a rectangular bounding box and then performing a load-balanced partition inside this box. The second line uses a greedy algorithm that starts by picking an arbitrary cell c_0 on the block boundary and then agglomerates a new block from the cell on the block boundary that is furthest away from the first cell. Once this new block is large enough, the routine once again picks the cell among the remaining ones that is furthest away from c_0 and starts to agglomerating a second block, and so on. The greedy algorithm comes in four different versions:

- `refineGreedy` – grows a new block inward from the block boundary by adding one entire ring of permissible neighbors in each iteration [3]. The permissible level-one neighbors are all cells sharing a face with existing cells in the growing block (optional parameter `'nlevel'=1`). The default behavior (`'nlevel'=2`) is to also include cells that share (at least) two faces with existing cells in the block or level-one cells in the ring.
- `refineGreedy2` – improved algorithm that only adds parts of the ring of level-one or level-two neighbors to honor the upper bound (14.3) on the flow indicator.
- `refineGreedy3` – sorts cells in the neighbor ring according to number of faces shared with the growing block. Permissible neighbors are defined by the function `neighboursByNodes` and includes all cells that share a face or a node with cells in the growing block. Computing this neighborship is expensive for large grids.
- `refineGreedy4` – sorts cells in the ring of neighbors according to difference in `Ifl` value, using the same permissible neighbors as `refineGreedy3`.

To illustrate how these functions operate for permissible level-one neighbors only, we consider a 4×4 grid with uniform flow indicator $I_f \equiv 1$. The functions are run with the following parameters:

```
p = refineGreedy(I,G,I,4,'nlevel',n);
  :
p = refineUniform(I,G,I,4,'CartDims',[2,2]);
```

where `I` is a vector of all ones. Figure 14.13 shows that the uniform partition gives four blocks, as expected. All greedy algorithms pick $c_0 = 1$ and start agglomerating blocks from cell 16. The simplest greedy algorithm, `refineGreedy`, only adds face neighbors and hence tends to create diamond-shaped blocks like the second block,

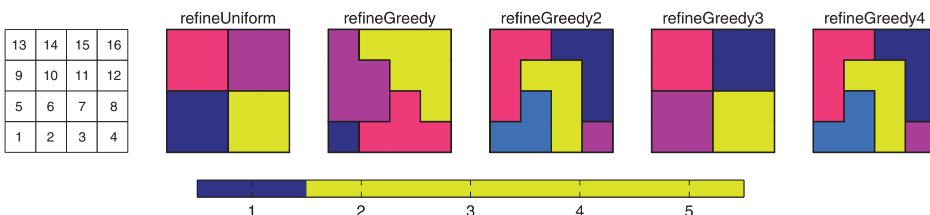


Figure 14.13 Refinement algorithms using level-one permissible neighbors on a 4×4 grid with uniform flow indicator.

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

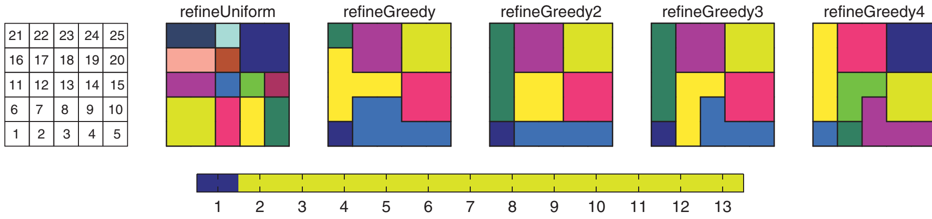


Figure 14.14 Refinement algorithms using level-two permissible neighbors (which is the default setting) on a 5 × 5 grid with uniform flow indicator.

which has six cells (8,11,12,14,15,16) and thus violates (14.3). After agglomerating three blocks, the algorithm is only left with cell c_0 and terminates. For `refineGreedy2` and `refineGreedy4`, the first step agglomerates cells (12,15,16). In the next step, cells (8,11,14) are possible merge candidates, out of which 8 is chosen since it has the smallest index. After the algorithm has agglomerated five blocks, there are no remaining cells in the original block and the algorithm terminates. The empty block is removed by a call to `compressPartition`, and blocks two to six are renumbered as blocks one to five. The third algorithm, `refineGreedy3`, selects cell 11 in the second step since this cell shares one more face with cells (12,15,16) than the other two candidates. Once the first block is formed, the greedy algorithms selects the remaining cell with maximum distance to c_0 as the next seed (cell 4 for the first algorithm, cell 11 for the second and fourth, and cell 8 for the third.)

Figure 14.14 shows a similar test case with level-two neighbors and a 5 × 5 grid. Here, the uniform refinement splits the block into four blocks of size 3×3, 2×3, 3×2, and 2×2. The first three violate (14.3) and are thus further split twice in each direction. To understand the irregular block shapes for the greedy algorithm, we can consider agglomeration of the fourth block, which starts by cells (4,5). Cell 3 is the only permissible neighbor in step two, whereas the third step has three candidates (2,7,8). All three are added by the first algorithm, whereas `refineGreedy2` chooses the cell with lowest index, which coincidentally gives a regular block, albeit with a larger aspect ratio. Why the other two algorithms choose cell 8 is a random effect caused by how MATLAB picks extremal values for a vector with identical entries.

The greedy refinement algorithms apply equally well to general polygonal/polyhedral grids with unstructured topologies. Uniform Cartesian refinement, on the other hand, relies on an ijk numbering and thus primarily works for grids with rectangular topology. However, the same idea can be generalized to unstructured grids if we instead of relying on topology use the cell centroids to sample from a rectangular subdivision of each block's geometric bounding box (see function `sampleFromBox`). This type of refinement is implemented in

```
p = refineRecursiveCart(p, G, I, NU, cartDims)
```

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

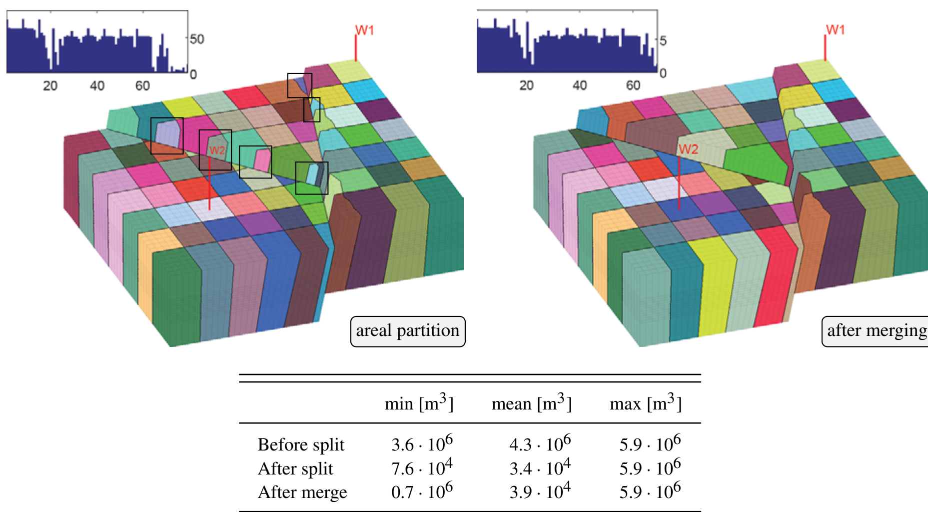
The function does not always produce the exact same results as `refineBlocks` with uniform refinement, but usually runs faster.

Refinement and merging are complementary operations that in principle can be applied in any order. A large number of tests nonetheless show that it is typically better to first merge small blocks, and then refine large ones, and improved grids can often be obtained if the merging-refinement process is iterated a few times. Notice also that the greedy algorithms tend to accentuate irregularity of blocks and should not be used uncritically.

14.5.6 Examples

We end by a few examples that show how the algorithmic components discussed in this section can be used to generate various types of coarse grids.

Example 14.5.1 (CaseB4) *As you may recall from Section 14.3.3, we first partitioned the CaseB4 grid in the ij coordinates and then split the resulting coarse blocks across the three faults. Figure 14.15 reports block sizes before and after splitting. To get rid of the smallest blocks, we can merge all blocks with a pore volume less than 20% of the mean block volume, which is approximately 50 times the mean cell volume. Applying one of the merging algorithms directly to the partition risks merging blocks across faults. We avoid this if we create a new vector that partitions the model into three fault blocks and use this to constrain the merging algorithm:*



	min [m^3]	mean [m^3]	max [m^3]
Before split	$3.6 \cdot 10^6$	$4.3 \cdot 10^6$	$5.9 \cdot 10^6$
After split	$7.6 \cdot 10^4$	$3.4 \cdot 10^4$	$5.9 \cdot 10^6$
After merge	$0.7 \cdot 10^6$	$3.9 \cdot 10^4$	$5.9 \cdot 10^6$

Figure 14.15 Areal partition for CaseB4 before and after merging of blocks whose pore volume is less than 50 times the average cell volume ($1.3 \cdot 10^4$ m^3). The tabular shows pore volumes in various stages of the algorithm.

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

```
Iv  = poreVolume(G, rock)./G.cells.volumes;
pm = mergeBlocks(pf, G, Iv, Iv, 50, 'static_partition', ...
     processPartition(G,ones(G.cells.num,1),find(G.faces.tag==1)));
```

Notice, that we here use the same indicator for both volume and flow, which generally would cause the algorithm to merge with blocks having similar average porosity values. After merging, the ratio between the smallest and the largest block is decreased by a factor ten, and we see that the algorithm has merged small blocks with neighbors on the correct side of the faults. It is important to merge small blocks before *we refine around wells to avoid merging any of the refined blocks in the near-well region.*

There are many different ways one can define flow indicators, for instance:

- From Darcy's law, we know that if a region is subject to a constant pressure gradient, the local magnitude of flow is proportional to permeability. Pressure gradients are rarely constant throughout a reservoir, but permeability can still be used as a simple a priori indicator to separate potential local regions of high and low flow before any flow solution is computed.
- Given a flow solution, it is natural to use velocity magnitude in each cell as a flow indicator. For a finite-volume method, this quantity must be reconstructed from the intercell fluxes (see discussion of non-Newtonian flow in Section 7.4), and alternatively one can use the sum of the absolute values of the fluxes as a measure that is simpler to compute. In a certain sense, flux/velocity can be considered as *local* indicators, since they only measure relative variations between individual cells and do not account for the actual flow paths.
- To get a *global* flow indicator that also takes representative flow paths into account, we can use time-of-flight or residence time.

As all three quantities tend to vary several orders of magnitude, we use their logarithm instead as suitable indicators:

```
iK = log10(rock.perm(:,1)); iK = iK - min(iK) + 1;
v  = sqrt(sum(faceFlux2cellVelocity(G, state.flux).^2, 2));
iV = log10(v); iV = iV - min(iV) + 1;
iT = -log10(Tf+Tb); iT = iT - min(iT) + 1;
```

assuming that `state` holds a representative flow solution and that `Tf,Tb` are the corresponding forward and backward time-of-flight values. Instead of residence time, we could have used the product of the two time-of-flight values. Let us illustrate these indicators by an example.

Example 14.5.2 (Adaptive coarsening) *Consider a five-spot pattern posed on Layer 25 from the SPE 10 model. For improved readability of plots, we only consider a* 120×60 *excerpt, initially partitioned into a* 6×3 *coarse grid. With a uniform flow indicator, we would get a coarse grid with* 5×5 *cells per block if we apply a* 2×2 *refinement twice.*

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

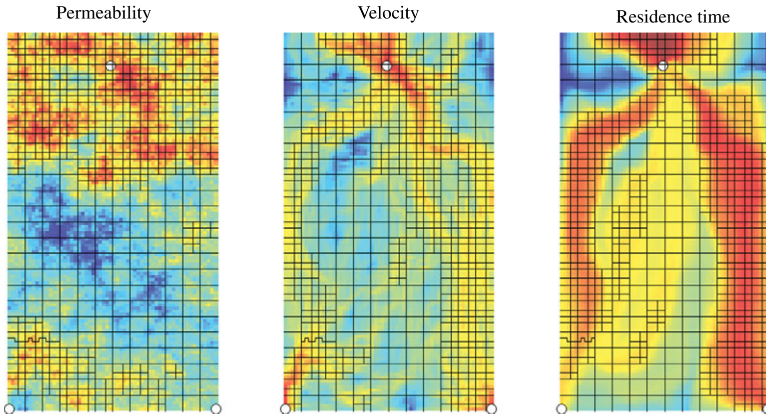


Figure 14.16 Recursive coarsening of a 120 × 60 excerpt of a five-spot well setup on SPE 10, Layer 25. The region is first partitioned uniformly into 6 × 3 coarse grids, and then refined recursively by a factor 2 × 2 by different flow indicators with $n_u = 25$ in (14.3).

With a heterogeneous flow indicator, we should expect to see local variations in the grid resolution. Figure 14.16 confirms this. All the initial blocks are refined twice, so that no grid block has more than 5 × 5 cells. Indicator `iK` *adds one extra refinement to all blocks having high permeabilities, since these would experience high flow if all cells in the model were subject to the same pressure drop. However, not all these blocks experience high flow for the* specific *five-spot flow pattern, and similarly there will be significant flow through many blocks with low(er) permeability because the injector and the two producers in the southeast/southwest corners lie on opposite sides of the low-permeability belt in the middle of the model.*

High-flow connections between the injector and the two producers are detected much better by the velocity indicator `iV`*. Unlike the indicator* `iT` *based on time-of-flight, the velocity indicator does not take into account the cumulative resistance to flow along flow paths and may therefore fail to refine blocks that have low local flow velocities but still lie along a flow path with low residence time. On the other hand, if a partition is built using a lot of specific flow information, it might be less suitable when applied to simulate radically different flow patterns.*

Example 14.5.3 (NUC algorithm) *The original nonuniform coarsening (NUC) algorithm [3] starts by segmenting a velocity indicator and then uses a sequence of merge-refine-merge operations as illustrated in Figure 14.17. Here, we have used the upper half of the model from the previous example, and constructed the NUC partition by the following sequence of operations:*

```
[volI,flwI] = deal( poreVolume(G, rock)./G.cells.volumes, iV);
ps = segmentIndicator(G, flwI, 5);
pm1 = mergeBlocks(ps, G, volI, flwI, 20);
pr  = refineGreedy(pm1, G, flwI, 30);
pm2 = mergeBlocks(pr, G, volI, flwI, 20);
```

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

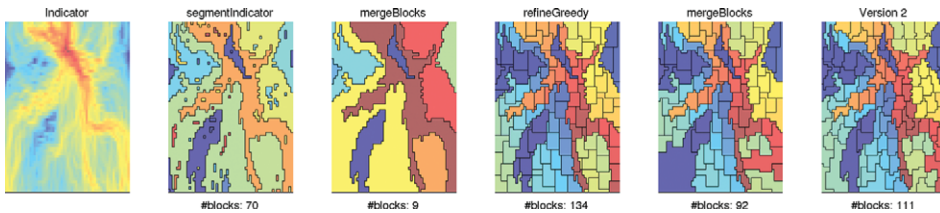


Figure 14.17 Application of the original nonuniform coarsening method of [3] for a 60×60 excerpts of a five-spot setup on Layer 25 of SPE 10.

The rightmost plot in the figure is obtained by replacing the merge and refinement operations by `mergeBlocks2` *and* `refineGreedy2`*, respectively. Although this type of partition originally was developed to coarsen transport equations, it is currently implemented as part of an industry-standard tool for single-phase upscaling [187] with time-of-flight as flow indicator.*

The `agglom` module contains several tutorial examples that give a more in-depth discussion of the NUC family of coarsening methods, and also includes an example of how the techniques in this section can be used to generate *dynamic* grid coarsening that adapts to evolving displacement fronts.

COMPUTER EXERCISES

14.5.1 How would you merge small blocks *after* the near-well refinement is introduced in CaseB4 to avoid merging these refined blocks?

14.5.2 Replace the merging in the SAIGUP example (Section 14.3.2) by `mergeBlocks`. Which flow indicator would you choose? Can you change the merging algorithm so that it can use the `SATNUM` field as flow indicator? (Hint: to this end you should use a majority vote rather than a volume-weighted average to get representative flow indicator per block.)

14.5.3 Rerun the NUC example using the other greedy algorithms with level-1 and level-2 neighbors and permeability and time-of-flight indicators. Try to use the NUC algorithm on other examples, e.g., CaseB4.

14.5.4 A greedy agglomeration strategy may not always be optimal: Assume that you have a coarse block with integrated indicator 410, and that the upper bound is 400. The algorithm then tries to agglomerate two blocks with indicators 400 and 10, respectively. Can you make a function that instead tries to split the indicator more evenly among the new blocks?

14.6 Multilevel Hierarchical Coarsening

Geocellular models are realizations of a deeper geological understanding that include structural, stratigraphic, sedimentologic, and diagenetic aspects. On the way to generate a grid model and populating it with petrophysical properties, the reservoir may have been divided

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

into different units, flow zones, environments of deposition, layering, and lithographic facies that each represents characteristics of the rock and how it was formed. The geocellular models we have encountered so far in the book have mainly consisted of a grid populated with a set of petrophysical properties like permeability, porosity, and net-to-gross. Some geocellular model may also contain geological indicators such as facies or rock type used to generate petrophysical properties. We have also seen that reservoirs can be subdivided into various types of regions to model spatial dependence in relative permeability models (PVTNUM), PTV behavior (PVTNUM), rock compressibility (ROCKTAB), and equilibration regions (EQLNUM).

The purpose of coarsening a grid is usually to develop one or more reduced models that only contain the most essential properties that affect fluid flow. It is well known that structural and stratigraphic frameworks have the largest impact on flow patterns, and preserving key concepts of these frameworks all the way to flow simulation is crucial to reliably predict flow patterns in the reservoir [48]. Likewise, it is important to preserve initial fluid contacts. Unfortunately, the volumetric partitions imposed by basic geological characterization are oftentimes lost in geostatistical algorithms and rarely made available as cell or face properties in the geocellular models. As an example, consider the distinction between high-permeability sand channels and low-permeability shales and coal in the Upper Ness formation from SPE 10. Likewise, think of CaseB4, where we in Figure 14.9 somewhat unsuccessfully tried to delineate four different geological layers based on permeability. To simplify future creation of coarse models, I therefore recommend that as much as possible of the basic geological characterization is preserved in terms of cellular indicators even after the geocellular model has been populated with petrophysical properties.

The idea of *hierarchical multilevel coarsening* is to define the various geological characterizations and regions just discussed as partition vectors, give them individual priority, and apply them recursively. By varying the number of features included, one can create a hierarchy of coarse partitions of increasing resolution. This approach can be combined with the agglomeration framework from the previous section to separate models into flow-dependent compartments representing high-flow and low-flow zones, or zones that are close to or far away from wells. A detailed discussion of this approach is outside the scope of this book. Instead, I give two illustrative examples.

Example 14.6.1 *The first example is a conceptual 40 × 40 model that has three different geological features: flow unit, lithofacies assemblage (LFA), and two intersecting fractures. The two first are represented by cell indicators* `pu` *and* `pl`*, whereas faults are represented as an explicit list of faces,* `faults`*. When combined, the three geological properties subdivide the reservoir into 15 different regions; see plots in the upper row of Figure 14.18. Flow unit and LFA are used to populate the geocellular model with stochastic realizations of the petrophysical properties. The lower-left plot in Figure 14.18 shows a flow indicator* `I` *derived from permeability, which has mean value approximately equal two in LFA 1 and*

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

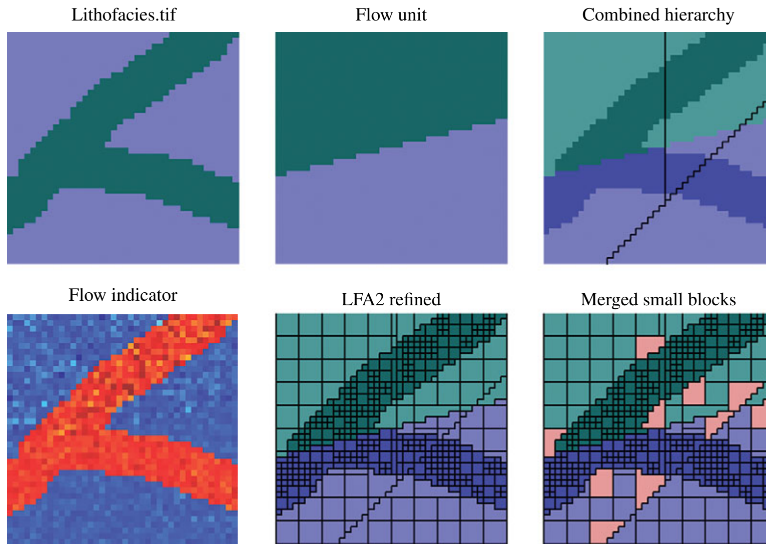


Figure 14.18 Multilevel hierarchical partition for a conceptual geological model with three geological properties, flow unit, lithofacies assemblage, and faults, which combined subdivide the reservoir into fifteen regions. The second row shows flow indicator, hierarchical 2×2 refinement, and the same refinement after blocks with less than four cells in LFA 1 have been merged (shown in a red shade).

ten in LFA 2. We use this flow indicator to introduce a hierarchical Cartesian subdivision with higher resolution in the second LFA[4] *:*

```
pc = ones(G.cells.num,4);
for i=1:4
   pc(:,i) = partitionCartGrid(G.cartDims,[5 5].*2^(i-1));
end
p = applySuccessivePart(processPartition(G,pu,faults),G, I, 8, [pl pc]);
```

That is, we first introduce a basic partition that splits the flow units and fault blocks, and then use the flow indicator to further partition the grid according to lithofacies and Cartesian blocks when the accumulated block indicator exceeds eight times the average cell value. This subdivision will create small blocks also inside the first lithofacies. To merge blocks with less than four cells, we can use `mergeBlocksByConnections`*, which merges blocks according to connection strength between fine-scale cells. The function does not merge blocks that have negative connection strength, and hence we set negative connection strength inside LFA 2 and between cells that belong to different regions in the geological hierarchy:*

[4] Complete code: `showHierCoarsen.m` in `book`.

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

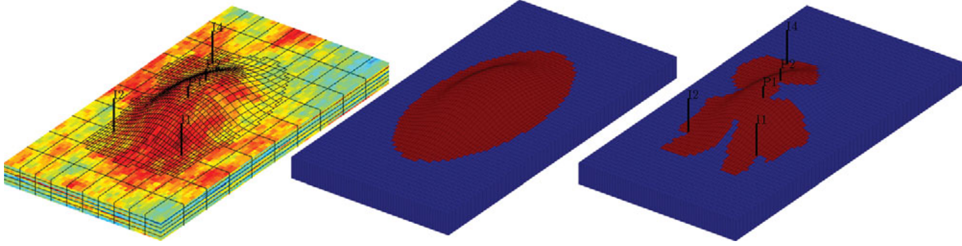


Figure 14.19 Conceptual illustration of a reservoir model with aggressive coarsening in the aquifer zone, modest coarsening above the initial water contact, and original resolution for cells with low residence time.

```
qb = [0; processPartition(G,pl+2*(pu-1), faults)];
ql = [0; pl];
T = 2*(qb(G.faces.neighbors(:,1)+1)==qb(G.faces.neighbors(:,2)+1))-1;
T(ql(G.faces.neighbors(:,1)+1)==2) = -1;
pm = mergeBlocksByConnections(G, p, T, 4);
```

The lower-right plot in Figure 14.18 shows the result. This example is admittedly simple, but similar principles can be applied to industry-standard models [187].

Example 14.6.2 *In the second example, we consider a conceptual model of an anticline reservoir that overlies an aquifer. We use a hierarchical approach to aggressively coarsen cells below the initial water contact, which are assumed to be water-filled throughout the whole simulation. For the cells above the water contact, we apply a more modest coarsening. To capture the initial displacement between injectors and producers, we pick all cells with residence time less than twice the median residence time of all cells perforated by wells. Cells with low residence time are part of the high-flow zones and are thus kept at their original resolution. In addition, all blocks in the coarsest partition that are perforated by a well are refined back to their original resolution to ensure that we accurately capture flow in the near-well zone. Figure 14.19 shows the resulting model. You can find complete source code in the script* `showAnticlineModel.m` *in the* `book` *module.*

14.7 General Advice and Simple Guidelines

Over the years, I have made the general and perhaps somewhat disappointing observation that making a good coarse grid is more an art than an exact science. In particular, I do not believe black-box approaches are able to automatically select the optimal coarsening. As a result, MRST does not provide well-defined workflows for coarsening, but rather offers tools that (hopefully) are sufficiently flexible to support you in combining your creativity, physical intuition, and experience to generate good coarse grids.

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press

Coarse partition are often used as input to a flow simulation. I conclude the chapter by suggesting a few guidelines that may contribute to decrease numerical errors introduced by coarsening:

1. Keep the grid as simple as possible: Use regular partitions for mild heterogeneities and cases where saturation/concentration profiles are expected to be regular.
2. Try to keep the number of connections between coarse-grid blocks as low as possible to minimize the bandwidth of the discretized systems, and avoid having too many small blocks, which otherwise would increase the dimension of the discrete system and adversely affect its stability without necessarily improving the accuracy significantly.
3. Use your understanding of the reservoir to judge what are the important features to preserve when coarsening the model. In doing so, you should think of what are primary geological features that control pressure communication and the main flow directions.
4. Make sure you have sufficient *vertical resolution* if you study systems with large density differences between the fluid phases (e.g., gas injection or CO_2 sequestration) to be able to capture the vertical segregation and the lighter fluid's tendency to override the other fluid(s).
5. Explore alternative partitioning strategies and compare the results.
6. Use relatively robust methods like METIS with transmissibility weights or regular partitions if you have limited understanding of which features affect the flow patterns.
7. Use basic geological features when available, e.g., facies likely yields more robust results than segmented permeability values.
8. Adjust your partitioning strategy to the purpose of the coarsening. If you want to run multiple simulations with more or less the same flow pattern, you can use features of the specific flow patterns to aggressively coarsen parts of the reservoir with low flow. Conversely, a fully flow-adapted coarsening may not work well in workflows where well positions and flow patterns change significantly from one simulation to the next.
9. Use simple indicator functions that have an intuitive interpretation, e.g., permeability, velocity magnitude, time-of-flight/travel time, etc.
10. Adapt coarse blocks to avoid upscaling large permeability contrasts, constrain blocks to contiguous saturation regions to avoid upscaling relative permeability and capillary pressure curves, etc.
11. Check the quality of a coarse grid using single-phase flow, e.g., by computing various types of flow diagnostics to check that you preserve the volumetric communication from the fine-scale model.

The next chapter introduces you to methods you can use to compute effective petrophysical properties on a coarsened grid.

https://doi.org/10.1017/9781108591416.019 Published online by Cambridge University Press