5
Incompressible Solvers for Single-Phase Flow

A simulation model can be considered to consist of three main parts; the first part describes the reservoir rock, the second part describes the mathematical laws that govern fluid behavior, and the last represents wells and other drive mechanisms. We have already discussed how to model the reservoir rock and its petrophysical properties in Chapters 2 and 3, and shown how the resulting models are represented in MRST using a grid object, usually called `G`, that describes the geometry of the reservoir, and a rock object, usually called `rock`, that describes petrophysical parameters. Likewise, Chapter 4 discussed the fundamental flow equations for single-phase flow and presented basic numerical discretizations for elliptic Poisson-type equations and the gradient/divergence operators that appear in flow models.

This chapter discusses the additional parts you need to make a full model and implement a simulator. We show how to represent fluid behavior in terms of fluid objects that contain basic properties such as density, viscosity, and compressibility. We will later extend these fluid objects to model more complex behavior by including properties like relative permeability and capillary pressure that describe interaction among multiple fluid phase and the porous rock. Likewise, we discuss necessary data structures to represent forcing terms such as boundary conditions, (volumetric) source terms, and models of injection and production wells. It is also convenient to introduce a state object holding the primary unknowns and derived quantities like pressure, fluxes, and face pressures.

There are two different ways the data objects just outlined can be combined to form a full simulator. In Section 4.4.2, we saw how to use discrete differential operators to write the flow equations in residual form and then employ automatic differentiation to linearize and form a linear system. Whereas this technique is elegant and will prove highly versatile for compressible flow models later in the book, it is an overkill for incompressible single-phase flow, since the flow equations already are linear. In this chapter, we therefore outline how to use a classic procedural approach to implement the discretized flow equations. We start by outlining the data structures and constructors needed to set up fluid properties and forcing terms, and once this is done, we move on to discuss in detail how to build two-point discretizations and assemble and solve corresponding linear systems. For pedagogical purposes, we present a somewhat simplified version of the basic flow solvers for incompressible flow that are implemented in the add-on modules `incomp` and `diagnostics` of

https://doi.org/10.1017/9781108591416.008 Published online by Cambridge University Press

MRST. At the end of the chapter we go through several simulation cases and give all code lines necessary for full simulation setups with various drive mechanisms.

5.1 Basic Data Structures in a Simulation Model

The simple flow solvers we discussed in the previous chapter did not contain any fluid properties and assumed no-flow boundary conditions and point sources as the only forcing term. In this section we outline basic data structures you can use to set up more comprehensive single-phase simulation cases.

5.1.1 Fluid Properties

The only fluid properties we need in the basic single-phase flow equations are the viscosity and the fluid density for incompressible models and the fluid compressibility for compressible models. More complex single-phase and multiphase models require additional fluid and rock-fluid properties. To simplify the communication of fluid properties between flow and transport solvers, it is good practice to introduce a common API. To this end, MRST uses so-called *fluid objects* that contain a predefined set of basic fluid properties as well as function handles used to evaluate rock-fluid properties that are only relevant for multiphase flow. This basic structure can be expanded by optional parameters and functions to represent more advanced fluid models. The following shows how to initialize the most basic fluid object that only requires viscosity and density as input

```
fluid = initSingleFluid('mu' , 1*centi*poise, ...
                        'rho', 1014*kilogram/meter^3);
```

After initialization, the fluid object contains pointers to functions that can be used to evaluate petrophysical properties of the fluid:

```
fluid =
    properties: @(varargin)properties(opt,varargin{:})
    saturation: @(x,varargin)x.s
       relperm: @(s,varargin)relperm(s,opt,varargin{:})
```

Only the first function is relevant for single-phase flow, and returns the viscosity when called with a single output argument, and the viscosity and the density when called with two output arguments. The other two functions can be considered as dummy functions that ensure that the single-phase fluid object is compatible with solvers written for more advanced fluid models. The `saturation` function accepts a reservoir state as argument (see Section 5.1.2) and returns the corresponding saturation (volume fraction of the fluid phase), which will either be empty or set to unity, depending upon how the reservoir state has been initialized. The `relperm` function accepts a fluid saturation as argument and returns the relative permeability, i.e., the reduction in permeability due to the presence of other fluid phases. This function should always be identical to one for single-phase models.

5.1.2 Reservoir States

To hold the dynamic state of the reservoir, MRST uses a special data structure. We refer to realizations of this structure as the *state objects*. In its basic form, the structure contains three elements: a vector `pressure` with one pressure per cell in the grid, a vector `flux` with one value per face in the grid, and a vector `s` that should either be empty or be a vector with a unit entry for each cell, since we only have a single fluid. The state object is initialized by a call to the function

```
state = initResSol(G, p0, s0);
```

where `p0` is the initial pressure and `s0` is an optional parameter, giving the initial saturation (which should be identical to one for single-phase models). Contrary to what the name may imply, this function *does not* initialize the fluid pressure to be in hydrostatic equilibrium. If such a condition is needed, it must be enforced explicitly by the user. In the case of wells in the reservoir, you should use the alternative function:

```
state = initState(G, W, p0, s0);
```

This gives a state object with an additional field `wellSol`, which is a vector with one entry per well. Each element in the vector is a structure that contains two fields, `wellSol.pressure` and `wellSol.flux`. These two fields are vectors of length equal the number of completions in the well and contain the bottom-hole pressure and flux for each completion.

5.1.3 Fluid Sources

The simplest way to describe flow in or out from interior points in the reservoir is to use volumetric source terms. You can create source terms as follows

```
src = addSource([],  cells, rates);
src = addSource(src, cells, rates, 'sat', sat);
```

Here, the input/output values are:

- `src`: array of MATLAB structures describing separate sources. If the first input argument is empty, the routine will output a single structure. Otherwise, it will append the new structure to the array of existing sources sent as input. Each source structure contains the following fields:
  - `cell`: cells containing explicit sources,
  - `rate`: rates for these explicit sources,
  - `value`: pressure or flux value for the given condition,
  - `sat`: fluid composition of injected fluids in cells with rate>0.
- `cells`: indices to the cells in the grid model in which this source term should be applied.

- **rates:** vector of volumetric flow rates, one scalar value for each cell in `cells`. Note that these values are interpreted as flux rates (typically in units of [m^3/day]) rather than as flux density rates (which must be integrated over the cell volumes to obtain flux rates).
- **sat:** optional parameter that specifies the composition of the fluid injected from this source. In this $n \times m$ array of fluid compositions, n is the number of elements in `cells` and m is the number of fluid phases. For $m = 3$, the columns are interpreted as: 1="aqua," 2="liquid," and 3="vapor." This field is for the benefit of multiphase transport solvers, and is ignored for all sinks (at which fluids flow *out* of the reservoir). The default value is `sat = []`, which corresponds to single-phase flow. If `size(sat,1)==1`, this saturation value will be repeated for all cells specified by `cells`.

For convenience, `rates` and `sat` *may* contain a single value; this value is then used for all faces specified in the call.

There can only be a single net source term per cell in the grid. Moreover, for incompressible flow with no-flow boundary conditions, the source terms *must* sum to zero if the model is to be well posed, or alternatively sum to the flux across the boundary. If not, we would either inject more fluids than we extract, or vice versa, and hence implicitly violate the assumption of incompressibility.

5.1.4 Boundary Conditions

As discussed in Section 4.3.1, all outer faces of a grid are assumed as no-flow boundaries unless other conditions are specified explicitly. The basic mechanism for specifying Dirichlet and Neumann boundary conditions is to use the function:

```
bc = addBC(bc, faces, type, values);
bc = addBC(bc, faces, type, values, 'sat', sat);
```

Here, the input values are:

- **bc:** array of MATLAB structures describing separate boundary conditions. If the first input argument is empty (`bc==[]`), the routine will output a single structure. Otherwise, it will append the new structure to the array of existing boundary conditions sent as input. Each structure contains the following fields:
  - **face:** external faces for which explicit conditions are set,
  - **type:** cell array of strings denoting type of condition,
  - **value:** pressure or flux value for the given condition,
  - **sat:** composition of fluids passing through inflow faces, not used for single-phase models.
- **faces:** array of external faces at which this boundary condition is applied.
- **type:** type of boundary condition. Supported values are `'pressure'` and `'flux'`, or a cell array of such strings.

- **`values`:** vector of boundary conditions, one scalar value for each face in `faces`. Interpreted as a pressure value in units [Pa] when `type` equals `'pressure'` and as a flux value in units [m^3/s] when `type` is `'flux'`. In the latter case, positive values in `values` are interpreted as injection fluxes *into* the reservoir, while negative values signify extraction fluxes, i.e., fluxes *out of* the reservoir.
- **`sat`:** optional parameter that specifies the composition of the fluid injected across inflow faces. Similar setup as explained for source terms in Section 5.1.3.

There can only be a single boundary condition per face in the grid. Solvers assume that boundary conditions are given on the boundary; conditions in the interior of the domain yield unpredictable results. Moreover, for incompressible flow and only Neumann conditions, the boundary fluxes *must* sum to zero if the model is to be well posed. If not, we would either inject more fluids than we extract, or vice versa, and hence implicitly violate the assumption of incompressibility.

For convenience, MRST also offers two additional routines for setting Dirichlet and Neumann conditions at all outer faces in a certain direction for grids having a logical IJK numbering:

```
bc = pside(bc, G, side, p);
bc = fluxside(bc, G, side, flux)
```

The `side` argument is a string that must match one out of the following six alias groups:

1:	'West',	'XMin',	'Left'
2:	'East',	'XMax',	'Right'
3:	'South',	'YMin',	'Back'
4:	'North',	'YMax',	'Front'
5:	'Upper',	'ZMin',	'Top'
6:	'Lower',	'ZMax',	'Bottom'

These groups correspond to the cardinal directions mentioned as the first alternative in each group. You should also be aware of an important difference in how fluxes are specified in `addBC` and `fluxside`. Specifying a scalar value in `addBC` means that this value will be copied to all faces the boundary condition is applied to, whereas a scalar value in `fluxside` sets the cumulative flux for all faces that make up the global side to be equal the specified value.

5.1.5 Wells

Wells are similar to source terms in the sense that they describe injection or extraction of fluids from the reservoir, but differ in the sense that they not only provide a volumetric flow rate, but also contain a model that couples this flow rate to the difference between the average reservoir in the grid cell and the pressure inside the wellbore. As discussed in Section 4.3.2, this relation can be written for each perforation as

$$v_p = J(p_i - p_f), \tag{5.1}$$

where J is the well index, p_i is the pressure in the perforated grid cell, and p_f is the flowing pressure in the wellbore. The wellbore is assumed to be in hydrostatic equilibrium so that p_f in each completion can be found from the pressure at the top of the well and the density along the wellbore. For single-phase, incompressible flow, this hydrostatic balance reads $p_f = p_{wh} + \rho \Delta z_f$, where p_{wh} is the pressure at the well head and Δz_f is the vertical distance from this point and to the perforation. By convention, the structure used to represent wells in MRST is called `W`, and consists of the following fields:

- **`cells`:** an array index to cells perforated by this well.
- **`type`:** string describing which variable is controlled (i.e., assumed to be fixed), either `'bhp'` or `'rate'`.
- **`val`:** the target value of the well control; pressure value for `type='bhp'` or rate for `type='rate'`.
- **`r`:** the wellbore radius (double).
- **`dir`:** a char describing the direction of the perforation (`'x'`, `'y'` or `'z'`).
- **`WI`:** the well index: either the productivity index or the well injectivity index depending on whether the well is producing or injecting.
- **`dZ`:** the height differences from the well head, which is defined as the topmost contact (i.e., the contact with the minimum z-value counted amongst all cells perforated by this well).
- **`name`:** string giving the name of the well.
- **`compi`:** fluid composition, only used for injectors.
- **`refDepth`:** reference depth of control mode.
- **`sign`:** defines whether the well is intended to be producer or injector.

Well structures are created by a call to the function

```
W = addWell(W, G, rock, cellInx);
W = addWell(W, G, rock, cellInx, 'pn', pv, ..);
```

Here, `cellInx` is a vector of indices to the cells perforated by the well, and `'pn'`/`pv` denote one or more keyword/value pairs that can be used to specify optional parameters in the well model:

- **`type`:** string specifying well control, `'bhp'` (default) means that the well is controlled by bottom-hole pressure, whereas `'rate'` means that the well is rate controlled.
- **`val`:** target for well control. Interpretation of this values depends upon `type`. For `'bhp'` the value is assumed to be in unit Pascal, and for `'rate'` the value is given in unit [m^3/sec]. Default value is `0`.
- **`radius`:** wellbore radius in meters. Either a single, scalar value that applies to all perforations, or a vector of radii, with one value for each perforation. The default radius is 0.1 m.
- **`dir`:** well direction. A single CHAR applies to all perforations, while a CHAR array defines the direction of the corresponding perforation.

- **innerProduct:** used for consistent discretizations discussed in Chapter 6.
- **WI:** well index. Vector of length equal the number of perforations in the well. The default value is -1 in all perforations, whence the well index will be computed from available data (cell geometry, petrophysical data, etc.) in grid cells containing well completions.
- **Kh:** permeability times thickness. Vector of length equal the number of perforations in the well. The default value is -1 in all perforations, whence the thickness will be computed from the geometry of each perforated cell.
- **skin:** skin factor for computing effective well bore radius. Scalar value or vector with one value per perforation. Default value: 0.0 (no skin effect).
- **Comp_i:** fluid composition for injection wells. Vector of saturations. Default value: Comp_i=[1,0,0] (water injection).
- **Sign:** well type: production (sign=-1) or injection (sign=1). Default value: [] (no type specified).
- **name:** string giving the name of the well. Default value is 'Wn', where *n* is the number of this well, i.e., n=numel(W)+1.

For convenience, MRST also provides the function

```
W = verticalWell(W, G, rock, I, J, K)
W = verticalWell(W, G, rock, I,    K)
```

for specifying vertical wells in models described by Cartesian grids or grids that have some kind of extruded structure. Here,

- **I,J:** gives the horizontal location of the well heel. In the first mode, both I and J are given and then signify logically Cartesian indices so that I is the index along the first logical direction, whereas J is the index along the second logical direction. This mode is only supported for grids having an underlying Cartesian (logical) structure such as purely Cartesian grids or corner-point grids.
  In the second mode, only I is described and gives the *cell index* of the topmost cell in the column through which the vertical well is completed. This mode is supported for logically Cartesian grids containing a three-component field G.cartDims or for otherwise layered grids that contain the fields G.numLayers and G.layerSize.
- K: a vector of layers in which this well should be completed. If isempty(K) is true, then the well is assumed to be completed in all layers in this grid column and the vector is replaced by 1:num_layers.

5.2 Incompressible Two-Point Pressure Solver

The two-point flux-approximation (TPFA) scheme introduced in Section 4.4.1 is implemented as two different routines. The first routine,

```
hT = computeTrans(G,rock)
```

computes the half-face transmissibilities and does not depend on the fluid model, the reservoir state, or the driving mechanisms and is hence part of is part of MRST's core functionality. The second routine

```
state = incompTPFA(state, G, hT, fluid, 'mech1', obj1, ..)
```

takes the complete model description as input and assembles and solves the two-point system. The routine is specific to incompressible flow and is thus placed in the `incomp` module. Here, `'mech'` specifies the drive mechanisms (`'src'`, `'bc'`, and/or `'wells'`) using correctly defined objects `obj`, as discussed in Sections 5.1.3–5.1.5.

Notice that `computeTrans` may fail to compute sensible transmissibilities if `rock.perm` is not given in SI units. Likewise, `incompTPFA` may produce strange results if the inflow and outflow specified by the boundary conditions, source terms, and wells do not sum to zero and hence violate the assumption of incompressibility. However, if fixed pressure is specified in wells or on parts of the outer boundary, there will be an outflow or inflow that will balance the net rate specified elsewhere.

The remainder of this section presents details of the inner workings of the incompressible solver. By going through the essential code lines needed to compute half-transmissibilities and solve and assemble the global system, we demonstrate how simple it is to implement the TPFA method on general polyhedral grid. If you are not interested in these details, you can jump directly to Section 5.4, which contains several examples demonstrating the use of the incompressible solver for single-phase flow.

To focus on the discretization and keep the discussion simple, we will not look at the full implementation of the two-point solver in `incomp`. Instead, we discuss excerpts from two simplified functions, `simpleComputeTrans` and `simpleIncompTPFA`, located in the `1phase` directory of the `mrst-book` module. Together, these form a simplified single-phase solver, which has been created for pedagogical purposes.

Assume we have a standard grid `G` containing cell and face centroids, e.g., as computed by the `computeGeometry` function discussed in Section 3.4. Then, the essential code lines of `simpleComputeTrans` are as follows: First, we define the vectors $\vec{c}_{i,k}$ from cell centroids to face centroids; see Figure 5.1. To this end, we first need to determine the map

$$T_{i,k} = A_{i,k}\boldsymbol{K}_i \frac{\vec{c}_{i,k}\cdot\vec{n}_{i,k}}{|\vec{c}_{i,k}|^2}$$

$$T_{ik} = [T_{i,k}^{-1} + T_{k,i}^{-1}]^{-1}$$

$$\sum_k T_{ik}(p_i - p_k) = q_i$$

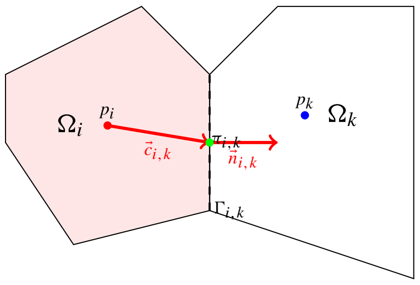


Figure 5.1 Two-point discretization on general polyhedral cells

from faces to cell number so that the correct cell centroid is subtracted from each face centroid.

```
hf = G.cells.faces(:,1);
hf2cn = gridCellNo(G);
C = G.faces.centroids(hf,:) - G.cells.centroids(hf2cn,:);
```

Face normals in MRST are assumed to have length equal the corresponding face areas, and hence correspond to $A_{i,k}\vec{n}_{i,k}$ in (4.51). To get the correct sign, we look at the neighboring information specifying which cells share the face: if the current cell number is in the first column, the face normal has positive sign. If not, it gets a negative sign:

```
sgn = 2*(hf2cn == G.faces.neighbors(hf, 1)) - 1;
N   = bsxfun(@times, sgn, G.faces.normals(hf,:));
```

The permeability tensor may be stored in different formats, as discussed in Section 2.5, and we therefore use an utility function to extract it:

```
[K, i, j] = permTensor(rock, G.griddim);
```

Finally, we compute the half transmissibilities, $C^T KN/C^T C$. To limit memory use, this is done in a for-loop (which is rarely used in MRST):

```
hT = zeros(size(hf2cn));
for k=1:size(i,2),
   hT = hT + C(:,i(k)) .* K(hf2cn, k) .* N(:,j(k));
end
hT = hT./ sum(C.*C,2);
```

The actual code has a few additional lines that perform various safeguards and consistency checks.

Once the half transmissibilities have been computed, they can be passed to the `simpleIncompTPFA` solver. The first thing this solver needs to do is adjust the half transmissibilities to account for fluid viscosity, since they were derived for a fluid with unit viscosity:

```
mob = 1./fluid.properties(state);
hT  = hT .* mob(hf2cn);
```

Then, we loop through all faces and compute the face transmissibility as the harmonic average of the half-transmissibilities

```
T = 1 ./ accumarray(hf, 1 ./ hT, [G.faces.num, 1]);
```

The MATLAB function `accumarray` constructs an array by accumulation (see Appendix A.4). A call to `a = accumarray(subs,val)` will use the subscripts in `subs` to create an array `a` based on the values `val`. Each element in `val` has a corresponding row in `subs`. The function collects all elements that correspond to identical subscripts in `subs` and stores their sum in the element of `a` corresponding to the subscript. In our case, `G.cells.faces(:,1)` gives the global face number for each half face, and hence the call to `accumarray` will sum the transmissibilities of the half-faces that correspond to a given global face and store the result in the correct place in a vector of `G.faces.num` elements. The function `accumarray` is very powerful and is used a lot in MRST in place of nested for-loops. In fact, we also employ this function to loop over all cells in the grid and collect and sum the transmissibilities of the faces of each cell to define the diagonal of the TPFA matrix:

```
nc = G.cells.num;
i  = all(G.faces.neighbors ~= 0, 2);
n1 = G.faces.neighbors(i,1);
n2 = G.faces.neighbors(i,2);
d  = accumarray([n1; n2], repmat(T(i),[2,1]),[nc, 1]);
```

Now that we have computed both the diagonal and the off-diagonal elements of A, the discretization matrix itself can be constructed by a straightforward call to MATLAB's `sparse` function:

```
I  = [n1; n2; (1:nc)'];
J  = [n2; n1; (1:nc)'];
V  = [-T(i); -T(i); d]; clear d;
A  = sparse(double(I), double(J), V, nc, nc);
```

Finally, we check if Dirichlet boundary conditions are imposed on the system, and if not, we modify the first element of the system matrix to fix the pressure in the first cell to zero, before solving the system:

```
A(1) = 2*A(1);
p = mldivide(A, rhs);
```

To solve the system, we rely on MATLAB's default solver `mldivide`, which uses a complex flow chart to check the structure of the matrix to see whether it is square, diagonal, (permuted) triangular, tridiagonal, banded, or Hermitian, and then chooses a specialized and efficient solver accordingly. By running the command `spparms('spumoni',2)` before you call `mldivide`, you can tell MATLAB to output information about which linear solver it chooses and the tests leading up the specific choice. For the type of sparse matrices we consider here, the end result is a call to a direct solver from UMFPACK implementing unsymmetric, sparse, multifrontal LU factorization [89, 88]. Such a direct solver is efficient for small-to-medium-sized systems, but for larger systems it is more efficient to use sparse iterative solvers such as a (preconditioned) multilevel method. The linear solver can be passed as a function-pointer argument to both `incompTPFA` and `simpleIncompTPFA`,

```
mrstModule add agmg
state = incompTPFA(state, G, hT, fluid, 'wells', W, 'LinSolve', @(A,b) agmg(A,b,1));
```

Here, we have used the aggregation-based algebraic multigrid solver AGMG [238, 15], which integrates well with MRST and is vailable at no cost for academic research and teaching. Section 12.3.4 discusses various specialized linear solver for compressible multiphase flow simulations.

Once the cell pressures have been computed, we can compute pressure values at the face centroids using the half-face transmissibilities

```
fp = accumarray(G.cells.faces(:,1), p(hf2cn).*hT, [G.faces.num,1])./ ...
     accumarray(G.cells.faces(:,1), hT, [G.faces.num,1]);
```

and then construct fluxes across the interior faces

```
ni   = G.faces.neighbors(i,:);
flux = -accumarray(find(i), T(i).*(p(ni(:,2))-p(ni(:,1))), [nf, 1]);
```

In the code excerpts given above, we did not account for gravity forces and general Dirichlet or Neumann boundary conditions, which both will complicate the code beyond the scope of the current presentation. The interested reader should consult the actual code to work out these details. The standard `computeTrans` function can also be used for different representations of petrophysical parameters, and includes functionality to modify the discretization by overriding the definition of cell and face centers and/or including multipliers that modify the values of the half-transmissibilities; see e.g., Sections 2.4.3 and 2.5.5. Likewise, the `incompTPFA` solver from the `incomp` module is implemented for a general, incompressible flow model with multiple fluid phases with flow driven by a general combination of boundary conditions, fluid sources, and well models.

We will shortly present several examples of how this solver can be used for flow problems on structured and unstructured grids. However, before doing so, we outline another flow solver from the `diagnostics` module, which will prove useful to visualize flow patterns.

5.3 Upwind Solver for Time-of-Flight and Tracer

The `diagnostics` module, discussed in more detail in Chapter 13, provides various functionality to probe a reservoir model to establish communication patterns between inflow and outflow regions, timelines for fluid movement, and various measures of reservoir heterogeneity. At the hart of this module, lies the function

```
tof = computeTimeOfFlight(state, G, rock, 'mech1', obj1, ..)
```

which implements the upwind, finite-volume discretization introduced in Section 4.4.3 for solving the time-of-flight equation $\vec{v} \cdot \nabla\tau = \phi$. As you probably recall, time-of-flight is the time it takes a neutral particle to travel from the nearest fluid source or inflow boundary to each point in the reservoir. Here, the `'mech'` arguments represent drive mechanisms (`'src'`, `'bc'`, and/or `'wells'`) specified in terms of specific objects `obj`, as discussed in Sections 5.1.3 to 5.1.5. You can also compute the backward time-of-flight – the time it takes to travel from any point in the reservoir to the nearest fluid sink or outflow boundary – with the same equation if we change sign of the flow field and modify the boundary conditions and/or source terms accordingly. In the following, we will go through the main parts of how this discretization is implemented.

We start by identifying all volumetric sources of inflow and outflow, which may be described as source/sink terms in `src` and/or as wells in `W`, and collect the results in a vector `q` of source terms having one value per cell

```
[qi,qs] = deal([]);
if  ~isempty(W),
   qi = [qi; vertcat(W.cells)];
   qs = [qs; vertcat(state.wellSol.flux)];
end
if ~isempty(src),
   qi = [qi; src.cell];
   qs = [qs; src.rate];
end
q = sparse(qi, 1, qs, G.cells.num, 1);
```

We also need to compute the accumulated inflow and outflow from boundary fluxes for each cell. This will be done in three steps. First, we create an empty vector `ff` with one entry per global face, find all faces that have Neumann conditions, and insert the corresponding value in the correct row

```
ff    = zeros(G.faces.num, 1);
isNeu = strcmp('flux', bc.type);
ff(bc.face(isNeu)) = bc.value(isNeu);
```

The flux is not specified on faces with Dirichlet boundary conditions and must be extracted from the solution computed by the pressure solver, i.e., from the `state` object that holds the reservoir state. We also need to set the correct sign so that fluxes *into* a cell are positive and fluxes *out of* a cell are negative. The sign of the flux across an outer face is correct if `neighbors(i,1)==0`, but if `neighbors(i,2)==0` we need to reverse the sign (we should also check that `i` is not empty)

```
i = bc.face(strcmp('pressure', bc.type));
ff(i) = state.flux(i) .* (2*(G.faces.neighbors(i,1)==0) - 1);
```

The last step is to sum all the fluxes across outer faces and collect the result in a vector `qb` that has one value per cell

```
outer = ~all(double(G.faces.neighbors) > 0, 2);
qb = sparse(sum(G.faces.neighbors(outer,:), 2), 1, ff(is_outer), G.cells.num, 1);
```

Here, `sum(G.faces.neighbors(outer,:), 2)` gives an array containing the indices of each cell attached to an outer face.

Once the contributions to inflow and outflow are collected, we can start building the upwind flux discretization matrix $\boldsymbol{A}$. The off-diagonal entries are defined such that $A_{ji} = \max(v_{ij}, 0)$ and $A_{ij} = -\min(v_{ij}, 0)$, where v_{ij} is the flux computed by the TPFA scheme discussed in the previous section.

```
i   = ~any(G.faces.neighbors==0, 2);
out = min(state.flux(i), 0);
in  = max(state.flux(i), 0);
```

The diagonal entry equals the outflux minus the divergence of the velocity, which can be obtained by summing the off-diagonal rows. This will give the correct equation in all cell except for those with a positive fluid source. Here, the net outflux equals the divergence of the velocity and we hence end up with an undetermined equation. In these cells, we can as a reasonable approximation[1] set the time-of-flight to be equal the time it takes to fill the cell, which means that the diagonal entry should be equal the fluid rate inside the cell.

```
n   = double(G.faces.neighbors(i,:));
inflow  = accumarray([n(:, 2); n(:, 1)], [in; -out]);
d = inflow + max(q+qb, 0);
```

Having obtained diagonal and all the nonzero off-diagonal elements, we can assemble the full matrix

```
nc = G.cells.num;
A  = sparse(n(:,2), n(:,1), in, nc, nc) + sparse(n(:,1), n(:,2), -out, nc, nc);
A = -A + spdiags(d, 0, nc, nc);
```

We have now established the complete discretization matrix, and time-of-flight can be computed by a simple matrix inversion

```
tof  = A \ poreVolume(G,rock);
```

If there are no gravity forces and the flux has been computed by the two-point method (or some other monotone scheme), one can show that the discretization matrix $\boldsymbol{A}$ can be permuted to a lower-triangular form [221, 220]. In the general case, the permuted matrix will be block triangular with irreducible diagonal blocks. Such systems can be inverted very

[1] Notice, however, that to get the correct values for 1D cases, it is more natural to set time-of-flight equal *half* the time it takes to fill the cell.

efficiently using a permuted back-substitution algorithm as long as the irreducible diagonal blocks are small. MATLAB is quite good at detecting such structures, and using the simple backslash (\) operator is therefore efficient, even for quite large models. However, for models of real petroleum assets described on stratigraphic grids (see Section 3.3), it is often necessary to preprocess flux fields to get rid of numerical clutter that would otherwise introduce large irreducible blocks inside stagnant regions. By specifying optional parameters to `computeTimeOfFlight`, the function will get rid of such small cycles in the flux field and set the time-of-flight to a prescribed upper value in all cells that have sufficiently small influx. This tends to reduce the computational cost significantly for large models with complex geology and/or significant compressibility effects.

The same routine can also compute *stationary tracers*, as discussed in Section 4.3.4. This is done by passing an optional parameter,

```
tof = computeTimeOfFlight(state, G, rock, .., 'tracer',tr)
```

where `tr` is a cell-array of vectors that each gives the indices of cells that emit a unique tracer. For incompressible flow, the discretization matrix of the tracer equation is the same as that for time-of-flight, and all we need to do to extend the solver is to assemble the right-hand side

```
numTrRHS = numel(tr);
TrRHS = zeros(nc,numTrRHS);
for i=1:numTrRHS,
   TrRHS(tr{i},i) = 2*qp(tr{i});
end
```

Since we have doubled the rate in any cells with a positive source when constructing the matrix $\boldsymbol{A}$, the rate on the right-hand side must also be doubled.

With the extra right-hand sides assembled, we can solve the combined time-of-flight/tracer problem as a linear system with multiple right-hand sides,

```
T  = A \ [poreVolume(G,rock) TrRHS];
```

which means that we essentially get the tracer for free as long as the number of tracers does not exceed the number of right-hand columns MATLAB can handle in one solve. We will return to a more thorough discussion of the tracer partitions Chapter 13 and show how these can be used to delineate connectivities within the reservoir. In the rest of this chapter, we will consider time-of-flight and streamlines as a means to study flow patterns in reservoir models.

5.4 Simulation Examples

We have now introduced you to all the functionality from the `incomp` module necessary to solve a single-phase flow problem, as well as the time-of-flight solver from the `diagnostics` module, which can be used to compute time lines in the reservoir. In the following, we discuss several examples and demonstrate step by step how to set up a flow model, solve it, and visualize and analyze the resulting flow field. Complete codes can be found in the `1phase` directory of the `book` module.

5.4.1 Quarter Five-Spot

As our first example, we show how to solve $-\nabla \cdot (\mathbf{K}\nabla p) = q$ with no-flow boundary conditions and two source terms at diagonally opposite corners of a 2D Cartesian grid covering a 500×500 m^2 area. This setup mimics a standard quarter five-spot well pattern, which we encountered in Figure 4.7 on page 126 when discussing well models. The full code is available in the script `quarterFiveSpot.m`. We use a rectangular grid with homogeneous petrophysical data ($K = 100$ mD and $\phi = 0.2$):

```
[nx,ny] = deal(32);
G = cartGrid([nx,ny],[500,500]);
G = computeGeometry(G);
rock = makeRock(G, 100*milli*darcy, .2);
```

As we saw above, all we need to develop the spatial discretization is the reservoir geometry and the petrophysical properties. This means that we can compute the half transmissibilities without knowing any details about the fluid properties and the boundary conditions and/or sources/sinks that will drive the global flow:

```
hT = simpleComputeTrans(G, rock);
```

The result of this computation is a vector with one value per local face of each cell in the grid, i.e., a vector with `G.cells.faces` entries.

The reservoir is horizontal and gravity forces are therefore not active. We create a fluid with properties that are typical for water:

```
gravity reset off
fluid = initSingleFluid('mu' , 1*centi*poise, ...
                        'rho', 1014*kilogram/meter^3);
```

To drive flow, we use a fluid source in the southwest corner and a fluid sink in the northeast corner. The time scale of the problem is defined by the strength of the source terms. Here, we set these terms so that a unit time corresponds to the injection of one pore volume of fluids. By convention, all flow solvers in MRST automatically assume no-flow conditions on all outer (and inner) boundaries if no other conditions are specified explicitly.

```
pv  = sum(poreVolume(G,rock));
src = addSource([], 1, pv);
src = addSource(src, G.cells.num, -pv);
```

The data structure used to represent the fluid sources contains three elements:

```
cell: [2x1 double]
rate: [2x1 double]
 sat: []
```

We recall that `src.cell` gives cell numbers where source terms are nonzero, and the vector `src.rate` specifies the fluid rates, which by convention are positive for inflow into the reservoir and negative for outflow from the reservoir. The last data element `src.sat` specifies fluid saturations, which only has meaning for multiphase flow models and hence is set to be empty here.

Strictly speaking, state structure need not be initialized for an incompressible model in which none of the fluid properties depend on the reservoir state. However, to avoid treatment of special cases, MRST requires that the structure is initialized and passed as argument to the pressure solver. We therefore initialize it with a dummy pressure value of zero and a unit fluid saturation (fraction of void volume filled by fluid), since we only have a single fluid

```
state = initResSol(G, 0.0, 1.0);
display(state)
```

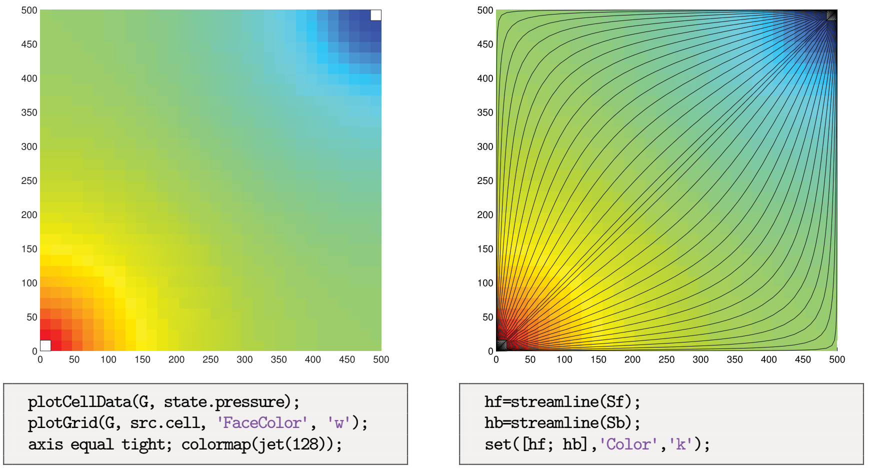
```
state =
    pressure: [1024x1 double]
        flux: [2112x1 double]
           s: [1024x1 double]
```

This completes the setup of the model. To solve for pressure, we simply pass reservoir state, grid, half transmissibilities, fluid model, and driving forces to the flow solver, which assembles and solves the incompressible equation.

```
state = simpleIncompTPFA(state, G, hT, fluid, 'src', src);
display(state)
```

```
state =
        pressure: [1024x1 double]
            flux: [2112x1 double]
               s: [1024x1 double]
    facePressure: [2112x1 double]
```

As explained here, `simpleIncompTPFA` solves for pressure as the primary variable and then uses transmissibilities to reconstruct the face pressure and intercell fluxes. After a call to the pressure solver, the `state` object is therefore expanded by a new field `facePressure` that contains pressures reconstructed at the face centroids. Figure 5.2 shows the resulting pressure distribution. To improve the visualization of the flow field, we show streamlines. The `streamlines` add-on module to MRST implements Pollock's method [253] for semi-analytical tracing of streamlines. Here, we use this functionality to trace streamlines forward and backward, starting from the midpoint of all cells along the northwest–southeast diagonal in the grid



```
plotCellData(G, state.pressure);
plotGrid(G, src.cell, 'FaceColor', 'w');
axis equal tight; colormap(jet(128));
```

```
hf=streamline(Sf);
hb=streamline(Sb);
set([hf; hb],'Color','k');
```

Figure 5.2 Solution of the quarter five-spot problem on a 32 × 32 uniform grid. The left plot shows the pressure distribution and in the right plot we have imposed streamlines passing through centers of the cells on the northwest–southeast diagonal.

```
mrstModule add streamlines;
seed = (nx:nx-1:nx*ny).';
Sf = pollock(G, state, seed, 'substeps', 1);
Sb = pollock(G, state, seed, 'substeps', 1, 'reverse', true);
```

The `pollock` routine produces a cell array of individual streamlines, which we pass onto MATLAB's `streamline` routine for plotting, as shown to the right in Figure 5.2.

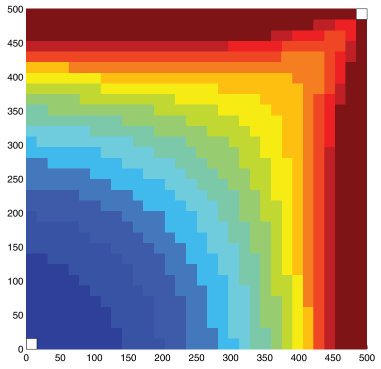
To get a better picture of how fast the fluid flows through our domain, we solve the time-of-flight equation (4.40) subject to the condition that $\tau = 0$ at the inflow, i.e., at all points where $q > 0$. For this purpose, we use the `computeTimeOfFlight` solver discussed in Section 5.3, which can compute time-of-flight both forward from inflow points and into the reservoir,

```
toff = computeTimeOfFlight(state, G, rock, 'src', src);
```

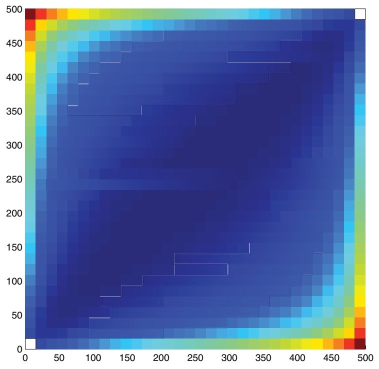
and from outflow points and backwards into the reservoir

```
tofb = computeTimeOfFlight(state, G, rock, 'src', src, 'reverse', true);
```

Isocontours of time-of-flight define natural time lines in the reservoir. To emphasize this, the left plot in Figure 5.3 shows time-of-flight plotted using only a few colors to make a rough contouring effect. The sum of forward and backward time-of-flight gives the total time it takes a fluid particle to pass from an inflow point to an outflow point. We can thus use this total residence time to visualize high-flow and stagnant regions, as demonstrated in the right plot of Figure 5.3.



```
plotCellData(G, toff);
plotGrid(G,src.cell,'FaceColor','w');
axis equal tight;
colormap(jet(16)); caxis([0,1]);
```



```
plotCellData(G, toff+tofb);
plotGrid(G,src.cell,'FaceColor','w');
axis equal tight;
colormap(jet(128));
```

Figure 5.3 Quarter five-spot problem on a 32 × 32 grid. The left plot shows time-of-flight plotted with a few color levels to create a crude contouring effect. The right plots total travel time clearly distinguishing high-flow and stagnant regions.

Computer exercises

5.4.1 Run the quarter five-spot example with the following modifications:

a. Replace the Cartesian grid by a curvilinear grid, e.g., using `twister` or a random perturbation of internal nodes as shown in Figure 3.3.
b. Replace the grid by the locally refined grid from Exercise 3.2.6.
c. Replace the homogeneous permeability by a heterogeneous permeability derived from the Carman–Kozeny relation (2.6).
d. Set the domain to be a single layer of the SPE 10 model. Hint: use `getSPE10rock()` to sample the petrophysical parameters and remember to convert to SI units.

Notice that the `pollock` function may not work for non-Cartesian grids.

5.4.2 Construct a grid similar to the one in Exercise 3.1.1, except that the domain is given a 90-degree flip so that axis of the cylindrical cutouts align with the z-direction. Modify the code presented in this section so that you can compute a five-spot setup with one injector near each corner and a producer in the narrow middle section between the cylindrical cutouts.

5.4.2 Boundary Conditions

To demonstrate how to specify boundary conditions, we go through essential code lines of three different examples. In all three examples, the reservoir is 50 m thick, is located at a depth of approximately 500 m, and is restricted to a 1×1 km^2 area. The permeability is uniform and anisotropic, with a diagonal (1,000, 300, 10) mD tensor, and the porosity is uniform and equal 0.2. In the first two examples, the reservoir is represented as a $20 \times 20 \times 5$ rectangular grid, and in the third example the reservoir is given as a corner-point grid of the same Cartesian dimension, but with an uneven uplift and four intersecting faults (as shown in the left plot of Figure 3.32):

```
[nx,ny,nz] = deal(20, 20, 5);
[Lx,Ly,Lz] = deal(1000, 1000, 50);
switch setup
   case 1,
      G = cartGrid([nx ny nz], [Lx Ly Lz]);
   case 2,
      G = cartGrid([nx ny nz], [Lx Ly Lz]);
   case 3,
      G = processGRDECL(makeModel3([nx ny nz], [Lx Ly Lz/5]));
      G.nodes.coords(:,3) = 5*(G.nodes.coords(:,3)-min(G.nodes.coords(:,3)));
end
G.nodes.coords(:,3) = G.nodes.coords(:,3) + 500;
```

Setting rock and fluid parameters, computing transmissibilities, and initializing the reservoir state can be done as explained in the previous section, and details are not included for brevity; you find the complete scripts in `boundaryConditions`.

Linear Pressure Drop

In the first example (`setup=1`), we specify Neumann conditions with total inflow of 5,000 m^3/day on the east boundary and Dirichlet conditions with fixed pressure of 50 bar on the west boundary:

```
bc = fluxside(bc, G, 'EAST', 5e3*meter^3/day);
bc = pside   (bc, G, 'WEST', 50*barsa);
```

This completes the definition of the model, and we can pass the resulting objects to the `simpleIncompTFPA` solver to compute the pressure distribution shown to the right in Figure 5.4. In the absence of gravity, these boundary conditions will result in a linear pressure drop from east to west inside the reservoir.

Hydrostatic Boundary Conditions

In the next example, we use the same model, except that we now include the effects of gravity and assume hydrostatic equilibrium at the outer vertical boundaries of the model. First, we initialize the reservoir state according to hydrostatic equilibrium, which is straightforward to compute if we for simplicity assume that the overburden pressure is caused by a column of fluids with the exact same density as in the reservoir:

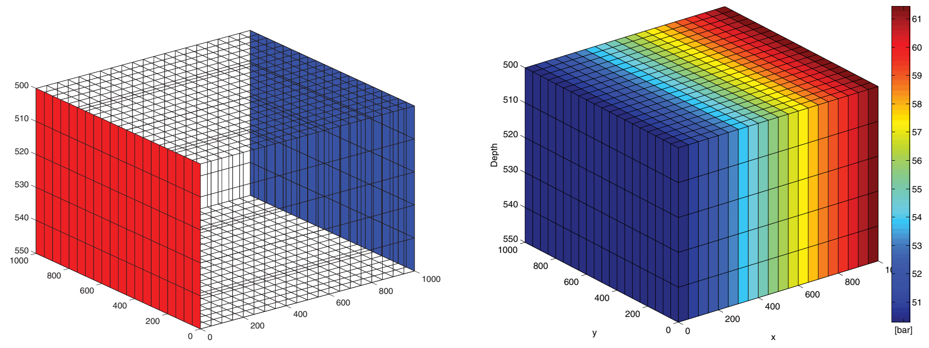


Figure 5.4 First example of a flow driven by boundary conditions. In the left plot, faces with Neumann conditions are marked in blue and faces with Dirichlet conditions are marked in red. The right plot shows the resulting pressure distribution.

```
state = initResSol(G, G.cells.centroids(:,3)*rho*norm(gravity), 1.0);
```

There are at least two different ways to specify hydrostatic boundary conditions. The simplest approach is to use the function `psideh`, i.e.,

```
bc = psideh([], G, 'EAST', fluid);   bc = psideh(bc, G, 'WEST', fluid);
bc = psideh(bc, G, 'SOUTH', fluid);  bc = psideh(bc, G, 'NORTH', fluid);
```

Alternatively, we can do it manually ourselves. To this end, we need to extract the reservoir perimeter defined as all exterior faces are vertical, i.e., whose normal vector has no z-component,

```
f = boundaryFaces(G);
f = f(abs(G.faces.normals(f,3))<eps);
```

To get the hydrostatic pressure at each face, we can either compute it directly by using the face centroids,

```
fp = G.faces.centroids(f,3)*rho*norm(gravity);
```

or we use the initial equilibrium that has already been established in the reservoir by sampling from the cells adjacent to the boundary

```
cif = sum(G.faces.neighbors(f,:),2);
fp  = state.pressure(cif);
```

The latter may be useful if the initial pressure distribution has been computed by a more elaborate procedure than what is currently implemented in `psideh`. In either case, the boundary conditions can now be set by the call

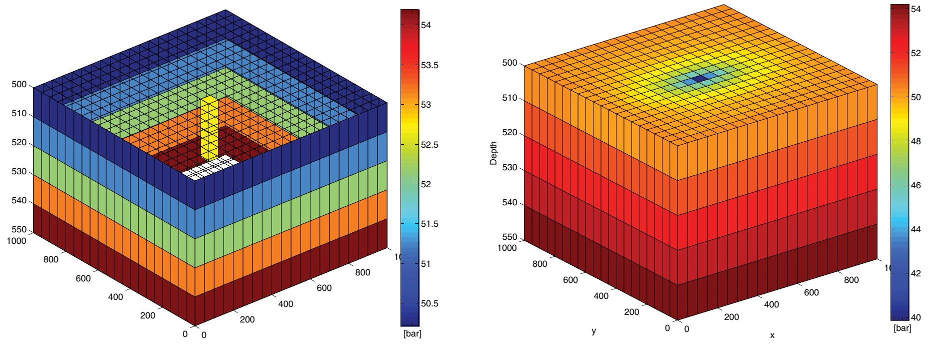


Figure 5.5 A reservoir with hydrostatic boundary condition and fluid extracted from a sink penetrating two cells in the upper two layers of the model. The left plot shows the boundary and the fluid sink, while the right plot shows the resulting pressure distribution.

```
bc = addBC(bc, f, 'pressure', fp);
```

To make the problem a bit more interesting, we also include a fluid sink at the midpoint of the upper two layers in the model,

```
ci = round(.5*(nx*ny-nx));
ci = [ci; ci+nx*ny];
src = addSource(src, ci, repmat(-1e3*meter^3/day,numel(ci),1));
```

Figure 5.5 shows the boundary conditions and source terms to the left and the resulting pressure distribution to the right. The fluid sink causes a pressure drawdown, which has ellipsoidal shape because of the anisotropic permeability field.

Conditions on Non-Rectangular Domain

For the reservoir shown in Figure 5.6, we cannot simply use tags for to select faces on the east and west perimeter. The problem is that `fluxside` and `pside` define cardinal sides to consist of all exterior faces whose normal vector point in the correct cardinal direction. This is illustrated in the upper-left plot of Figure 5.6, where we have tried to specify boundary conditions using the same procedure as in Figure 5.4. We can easily get rid of faces that really lie on the north and south boundary if we use the subrange feature of `fluxside` and `pside` to restrict the boundary conditions to a subset of the global side, as shown in the upper-right plot of Figure 5.6

```
bc = fluxside([], G, 'EAST', 5e3*meter^3/day, 4:15, 1:5);
bc = pside   (bc, G, 'WEST', 50*barsa,        7:17, []);
```

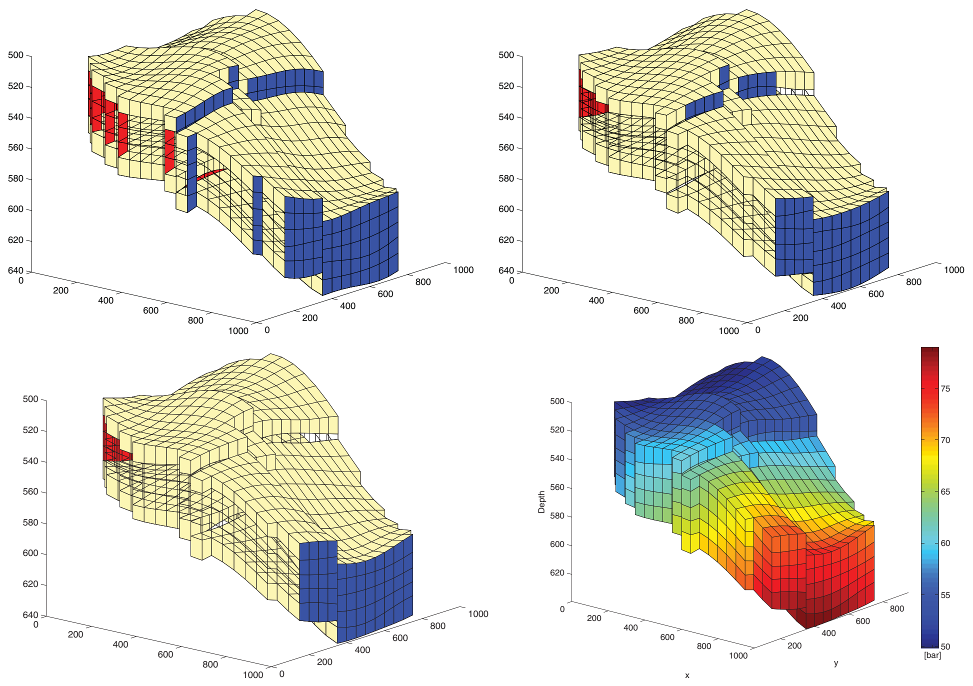


Figure 5.6 Specifying boundary conditions along the outer perimeter of a corner-point model. The upper-left plot shows the use of `fluxside/pside` (blue/red color) and to set boundary conditions on the east and west global boundaries. In the upper-right point, the same functions have been used with a specification of subranges in the global sides. In the lower-left plot, we have utilized user-supplied information to correctly set the conditions only along the perimeter. The lower-right plot shows the resulting pressure solution.

Unfortunately, `fluxside/pside` still picks exterior faces at the faults, classified as pointing east/west because of their normal. Similar problems may arise in other models because of pinched, eroded, or inactive cells.

To find the east- and west-most faces only, we need to use advanced tactics. In our case, the natural perimeter is defined as those faces that lie on the bounding box of the model, on which we distribute the total flux to individual faces according to the face area. For the Neumann condition we therefore get

```
x = G.faces.centroids(f,1);
[xm,xM] = deal(min(x), max(x));
ff = f(x>xM-1e-5);
bc = addBC(bc, ff, 'flux', (5e3*meter^3/day) ...
          * G.faces.areas(ff)/ sum(G.faces.areas(ff)));
```

We can specify the Dirichlet condition in a similar manner. The lower-right plot shows the correct linear pressure drop.

COMPUTER EXERCISES

5.4.3 Consider a 2D box with a sink at the midpoint and inflow across the perimeter specified either in terms of a constant pressure or a constant flux. Are there differences in the two solutions, and if so, can you explain why? Hint: use time-of-flight, total travel time, and/or streamlines to investigate.

5.4.4 Apply the production setup from Figure 5.5, with hydrostatic boundary conditions and fluids extracted from two cells at the midpoint of the model, to the model depicted in Figure 5.6.

5.4.5 Compute flow for all models in data sets `BedModels1` and `BedModel2` subject to linear pressure drop in the x and then in the y-direction. These models of small-scale heterogeneity are developed to compute representative properties in simulation models on a larger scale. A linear pressure drop is the most widespread computational setup used for flow-based upscaling. What happens if you try to specify flux conditions?

5.4.6 Consider models from the `CaseB4` data set. Use appropriate boundary conditions to drive flow across the faults and compare flow patterns computed on the pillar and the stair-stepped grid for the two different model resolutions. Can you explain any differences you observe?

5.4.3 Structured versus Unstructured Stencils

We have so far only discussed hexahedral grids having structured cell numbering. The two-point schemes can also be applied to fully unstructured and polyhedral grids. To demonstrate this, we define a non-rectangular reservoir by scaling the grid from Figure 3.8 to cover a 1×1 km^2 area.

```
load seamount;
T = triangleGrid([x(:) y(:)], delaunay(x,y));
[Tmin,Tmax] = deal(min(T.nodes.coords), max(T.nodes.coords));
T.nodes.coords = bsxfun(@times, ...
     bsxfun(@minus, T.nodes.coords, Tmin), 1000./(Tmax - Tmin));
T = computeGeometry(T);
```

We assume a homogeneous and isotropic permeability of 100 mD and use the same fluid properties as in the previous examples. Constant pressure of 50 bar is set at the outer perimeter and fluid is drained at a constant rate of one pore volume over 50 years from a well located at (450,500). (Script: `stencilComparison.m`.)

For comparison, we generate two Cartesian grids covering the same domain, one with approximately the same number of cells as the triangular grid and a 10×10 refinement of this grid to provide a reference solution,

```
G = computeGeometry(cartGrid([25 25], [1000 1000]));
inside = isPointInsideGrid(T, G.cells.centroids);
G = removeCells(G, ~inside);
```

The function `isPointInsideGrid` implements a simple algorithm for finding whether points lie inside the circumference of a grid. First, all boundary faces are extracted and then the corresponding nodes are sorted so that they form a closed polygon. Then, we use MATLAB's built-in function `inpolygon` to check if the points are inside this polygon. To also construct a radial grid, we reuse the code from page 86 to set up points inside $[-1,1] \times [-1,1]$ graded radially towards the origin

```
P = [];
for r = exp([-3.5:.2:0, 0, .1]),
   [x,y] = cylinder(r,25); P = [P [x(1,:); y(1,:)]];
end
P = unique([P'; 0 0],'rows');
```

We scale the points and translate so that their origin coincides with the fluid sink

```
[Pmin,Pmax] = deal(min(P), max(P));
P = bsxfun(@minus, bsxfun(@times, ...
           bsxfun(@minus, P, Pmin), 1200./(Pmax-Pmin)), [150 100]);
```

We remove all points outside of the triangular grid before we use the point set to first generate a triangular and then a Voronoi grid:

```
inside = isPointInsideGrid(T, P);
V = computeGeometry( pebi( triangleGrid(P(inside,:)) ));
```

Once we have constructed the grids, the setup of the remaining part of the model is the same in all cases. To avoid unnecessary replication of code, we collect the grids in a cell array and use a simple for-loop to set up and simulate each model realization:

```
g = {G, T, V, Gr};
for i=1:4
   rock = makeRock(g{i}, 100*milli*darcy, 0.2);
   hT = simpleComputeTrans(g{i}, rock);
   pv = sum(poreVolume(g{i}, rock));

   tmp = (g{i}.cells.centroids - repmat([450, 500],g{i}.cells.num,[])).^2;
   [~,ind] = min(sum(tmp,2));
   src{i} = addSource(src{i}, ind, -.02*pv/year);

   bc{i} = addBC([], boundaryFaces(g{i}), 'pressure', 50*barsa);

   state{i} = incompTPFA(initResSol(g{i},0,1), ...
       g{i}, hT, fluid, 'src', src{i}, 'bc', bc{i}, 'MatrixOutput', true);

   [tof{i},A{i}] = computeTimeOfFlight(state{i}, g{i}, rock,...
       'src', src{i},'bc',bc{i}, 'reverse', true);
end
```

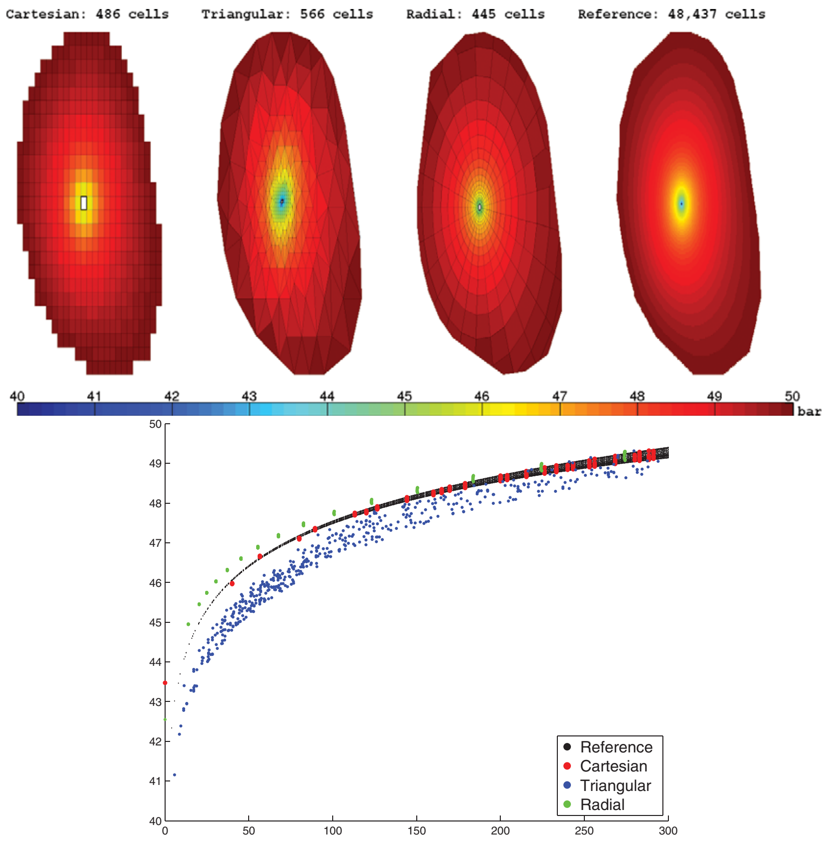


Figure 5.7 Comparison of the pressure solution for three different grid types: uniform Cartesian, triangular, and a graded radial grid. The scattered points used to generate the triangulated domain and limit the reservoir are sampled from the `seamount` data set and scaled to cover a 1×1 km^2 area. Fluids are drained from the center of the domain, assuming a constant pressure of 50 bar at the perimeter.

Figure 5.7 shows the pressure solutions computed on the four different grids, whereas Figure 5.8 reports the sparsity patterns of the corresponding linear systems for the three coarse grids. As expected, the Cartesian grid gives a banded matrix consisting of five diagonals corresponding to each cell and its four neighbors in the cardinal directions. Even though this discretization is not able to predict the complete drawdown at the center (the reference solution predicts a pressure slightly below 40 bar), it captures the shape of the drawdown region quite accurately; the region appears ellipsoidal because of the non-unit aspect ratio in the plot. In particular, we see that the points in the radial plot follow those of the fine-scale reference closely. The spread in the points as $r \to 300$ is not a grid orientation effect, but the result of variations in the radial distance to the fixed pressure at the outer boundary on all four grids.

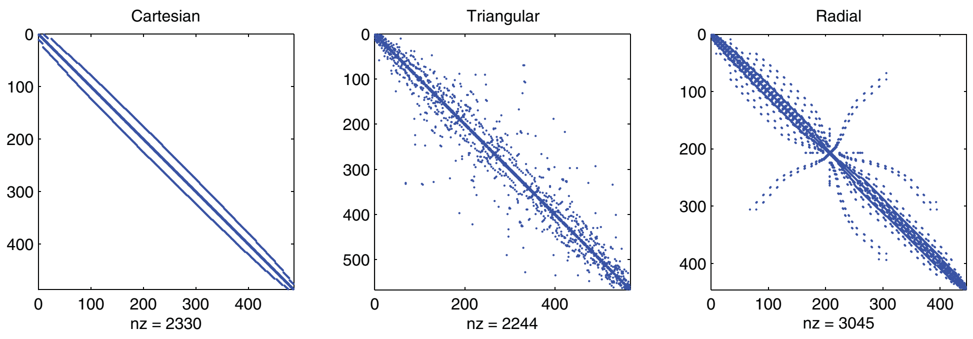


Figure 5.8 Sparsity patterns for the TPFA stencils on the three different grid types shown in Figure 5.7.

The unstructured triangular grid is more refined near the well and is hence able to predict the pressure drawdown in the near-well region more accurately. However, the overall structure of this grid is quite irregular, as you can see from the sparsity pattern of the linear system shown in Figure 5.8, and the irregularity gives significant grid orientation effects. You can see this from the irregular shape of the color contours in the upper part of Figure 5.7, as well as from the spread in the scatter plot. In summary, this grid is not well suited for resolving the radial symmetry of the pressure drawdown in the near-well region. But to be fair, the grid was not generated for this purpose either.

Except for close to the well and close to the exterior boundary, the topology of the radial grid is structured in the sense that each cell has four neighbors, two in the radial direction and two in the angular direction, and the cells are regular trapezoids. This should, in principle, give a banded sparsity pattern if the cells are ordered starting at the natural center point and moving outward, one ring at the time. To verify this claim, you can execute the following code:

```
[~,q] = sort(state{3}.pressure);
spy(state{3}.A(q,q));
```

However, as a result of how the grid was generated by first triangulating and then forming the dual, the cells are numbered from west to east, which explains why the sparsity pattern is so far from being a simple banded structure. This may potentially affect the efficiency of a linear solver, but has no impact on the accuracy of the numerical approximation, which is good because of the grading towards the well and the symmetry inherent in the grid. Slight differences in the radial profile compared with the Cartesian grid(s) are mainly the result of the fact that the source term and the fixed pressure conditions are not located at the exact same positions in the simulations.

In Figure 5.9, we also show the sparsity pattern of the linear system used to compute the reverse time-of-flight from the well and back into the reservoir. Using the default cell ordering, the sparsity pattern of each upwind matrix will appear as a less dense version

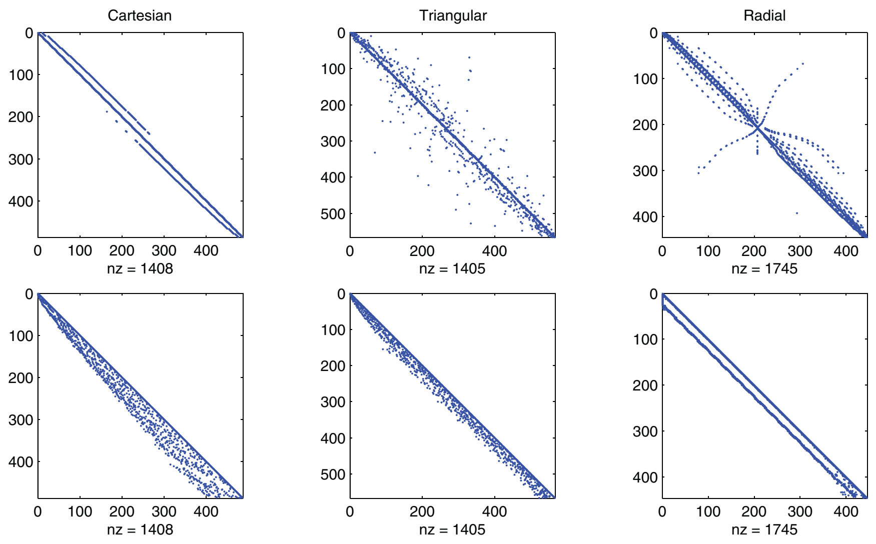


Figure 5.9 Sparsity patterns for the upwind stencils used to compute time-of-flight on the three different grid types shown in Figure 5.7. In the lower row, the matrices have been permuted to lower-triangular form by sorting the cell pressures in ascending order.

of the pattern for the corresponding TPFA matrix. However, whereas the TPFA matrices represent an elliptic equation in which information propagates in both directions across cell interfaces, the upwind matrices are based on one-way connections arising from fluxes between pairs of cells that are connected in the TPFA discretization. To reveal the true nature of the system, we can permute the system by either sorting the cell pressures in ascending order (potential ordering) or using the function `dmperm` to compute a Dulmage–Mendelsohn decomposition. As pointed out in Section 5.3, the result is a lower triangular matrix, from which it is simple to see the unidirectional propagation of information you should expect for a hyperbolic equation having only positive characteristics.

Computer exercises

5.4.7 Compare the sparsity patterns resulting from potential ordering and use of `dmperm` for both the upwind and the TPFA matrices.

5.4.8 Investigate the flow patterns in more details using forward time-of-flight, travel time, and streamlines.

5.4.9 Replace the boundary conditions by a constant influx, or set pressure values sampled from a radially symmetric pressure solution in an infinite domain.

5.4.4 Using Peaceman Well Models

Flow in and out of a wellbore takes place on a scale much smaller than a single grid cell in typical sector and field models and is therefore commonly modeled using a semi-analytical model of the form (4.35). This section presents two examples to demonstrate how such models can be included in the simulation setup using data objects and utility functions introduced in Section 5.1.5.

Box Reservoir

We consider a reservoir consisting of a homogeneous 500 × 500 × 25 m^3 sand box with an isotropic permeability of 100 mD, represented on a regular 20 × 20 × 5 Cartesian grid. The fluid is the same as in the examples as in the earlier examples in this section. You can find all code lines necessary to set up the model, solve the flow equations, and visualize the results in the script `firstWellExample.m`. Setting up the model is quickly done once you have gotten familiar with MRST:

```
[nx,ny,nz] = deal(20,20,5);
G = computeGeometry( cartGrid([nx,ny,nz], [500 500 25]) );
rock = makeRock(G, 100*milli*darcy, .2);
fluid = initSingleFluid('mu',1*centi*poise,'rho',1014*kilogram/meter^3);
hT    = computeTrans(G, rock);
```

The reservoir is produced by a well pattern consisting of a vertical injector and a horizontal producer. The injector is located in the southwest corner of the model and operates at a constant rate of 3,000 m^3 per day. The producer is completed in all cells along the upper east rim and operates at a constant bottom-hole pressure of 1 bar (i.e., 10^5 Pascal in SI units):

```
W = verticalWell([], G, rock, 1, 1, 1:nz, 'Type', 'rate', 'Comp_i', 1,...
              'Val', 3e3/day(), 'Radius', .12*meter, 'name', 'I');
W = addWell(W, G, rock, nx : ny : nx*ny, 'Type', 'bhp',  'Comp_i', 1, ...
          'Val', 1.0e5, 'Radius', .12*meter, 'Dir', 'y', 'name', 'P');
```

In addition to specifying the type of control on the well (`'bhp'` or `'rate'`), we must specify wellbore radius and fluid composition, which is `'1'` for a single phase. After initialization, the array `W` contains two data objects, one for each well:

```
Well #1:                         |       Well #2:
       cells: [5x1 double]       |              cells: [20x1 double]
        type: 'rate'             |               type: 'bhp'
         val: 0.0347             |                val: 100000
           r: 0.1000             |                  r: 0.1000
         dir: [5x1 char]         |                dir: [20x1 char]
          WI: [5x1 double]       |                 WI: [20x1 double]
          dZ: [5x1 double]       |                 dZ: [20x1 double]
        name: 'I'                |               name: 'P'
       compi: 1                  |              compi: 1
    refDepth: 0                  |           refDepth: 0
        sign: 1                  |               sign: []
```

This concludes the model specification, and we now have all the information we need to initialize the reservoir state, and assemble and solve the system

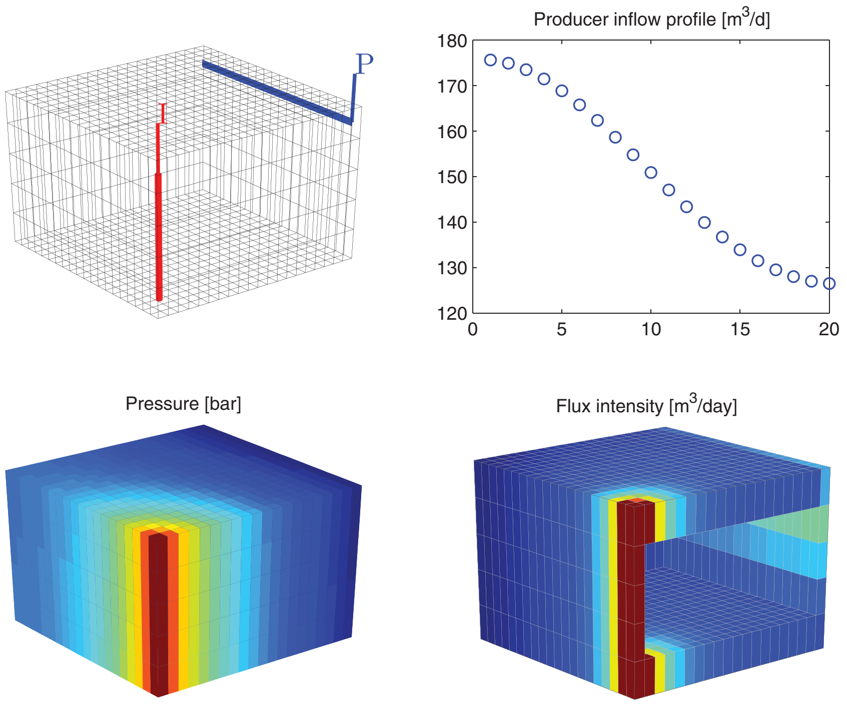


Figure 5.10 Solution of a single-phase, incompressible flow problem inside a box reservoir with a vertical injector and a horizontal producer.

```
gravity reset on;
resSol = initState(G, W, 0);
state  = incompTPFA(state, G, hT, fluid, 'wells', W);
```

Figure 5.10 shows how the inflow rate decays with the distance to the injector as expected. We can compute the flux intensity depicted in the lower-right plot using the following command, which first maps the vector of face fluxes to a vector with one flux per half face and then sums the absolute value of these fluxes to get a flux intensity per cell:

```
cf = accumarray(getCellNoFaces(G), abs(faceFlux2cellFlux(G, state.flux)));
```

Shallow-Marine Reservoir

In the final example, we return to the SAIGUP model from Section 3.5.1. This model does not represent a real reservoir, but is one out of a large number of models built to be plausible realizations that contain the types of structural and stratigraphic features one could encounter in models of real clastic reservoirs. Continuing from Section 3.5.1, we simply assume that the grid and the petrophysical model has been loaded and processed.

The script `saigupWithWells.m` reports all details of the setup and also explains how to speed up the grid processing by using two C-accelerated routines for constructing a grid from ECLIPSE input and computing areas, centroids, normals, volumes, etc.

The permeability input is an anisotropic tensor with zero vertical permeability in a number of cells. As a result, some parts of the reservoir may be completely sealed off from the wells. This will cause problems for the time-of-flight solver, which requires that all cells in the model must be flooded after some finite time that can be arbitrarily large. To avoid this potential problem, we assign a small constant times the minimum positive vertical permeability to the cells that have zero cross-layer permeability.

```
is_pos = rock.perm(:, 3) > 0;
rock.perm(~is_pos, 3) = 1e-6*min(rock.perm(is_pos, 3));
```

Similar safeguards are implemented in most commercial simulators.

We recover fluid from the reservoir using six producers spread throughout the middle of the reservoir; each producer operates at a fixed bottom-hole pressure of 200 bar. Eight injectors located around the perimeter provide pressure support, each operating at a prescribed and fixed rate. The wells are described by a Peaceman model as in the previous example. To make the code as compact as possible, all wells are vertical with location specified in the logical ij subindex available in the corner-point format. The following code specifies the injectors:

```
nz = G.cartDims(3);
I = [ 3, 20,  3, 25,  3, 30,  5, 29];
J = [ 4,  3, 35, 35, 70, 70,113,113];
R = [ 1,  3,  3,  3,  2,  4,  2,  3]*500*meter^3/day;
W = [];
for i = 1 : numel(I),
  W = verticalWell(W, G, rock, I(i), J(i), 1:nz, 'Type', 'rate', ...
                   'Val', R(i), 'Radius', .1*meter, 'Comp_i', 1, ...
                   'name', ['I$_{', int2str(i), '}$']);
end
```

The producers are specified in the same way. Figure 5.11 shows the well positions and the pressure distribution. We see a clear pressure buildup along the east, south, and west rims of the model. Similarly, there is a pressure drawdown in the middle of the model around producers P2, P3, and P4. The total injection rate is set so that one pore volume will be injected in a little less than 40 years.

This is a single-phase simulation, but let us for a while think of our setup in terms of injection and production of different fluids (since the fluids have identical properties, we can think of a blue fluid being injected into a black fluid). In an ideal situation, one would wish that the blue fluid would sweep the whole reservoir before it breaks through to the production wells, as this would maximize the displacement of the black fluid. Even in the simple quarter five-spot examples in Section 5.4.1 (see Figure 5.3), we saw that this was not the case, and you cannot expect that this will happen here, either. The lower plot in Figure 5.11 shows all cells in which the total travel time (sum of forward and backward

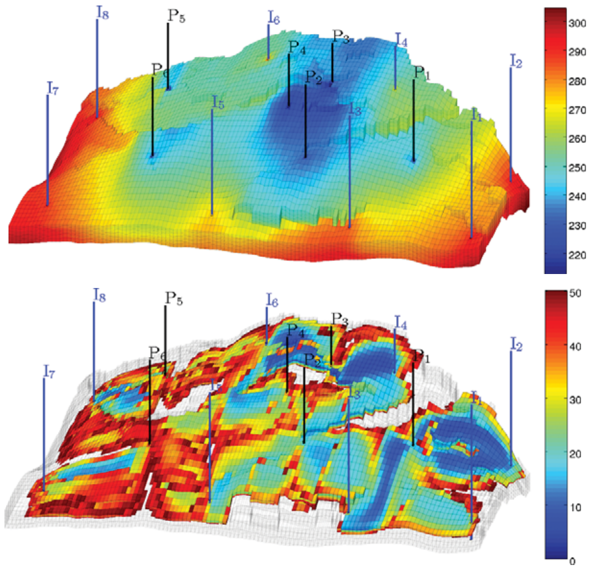


Figure 5.11 Incompressible, single-phase simulation of the SAIGUP model. The upper plot shows pressure distribution, and the lower plot shows cells with total travel time of less than 50 years.

time-of-flight) is less than 50 years. By looking at such a plot, you can get a quite a good idea of regions where there is very limited communication between the injectors and producers (i.e., areas without colors). If this was a multiphase flow problem, these areas would typically contain bypassed or unswept oil and be candidates for infill drilling or other mechanisms that would improve the volumetric sweep. We will come back to a more detailed discussion of flow patterns and volumetric connections in Section 13.5.2.

Computer exercises

5.4.10 Change the parameter `'Dir'` from `'y'` to `'z'` in the box example and rerun the case. Can you explain why you get a different result?

5.4.11 Switch the injector in the box example to be controlled by a bottom-hole pressure of 200 bar. Where would you place the injector to maximize production rate if you can only perforate (complete) it in five cells?

5.4.12 Consider the SAIGUP model: can you improve the well placement and/or the distribution of fluid rates. Hint: is it possible to utilize time-of-flight information?

5.4.13 Use the function `getSPE10setup` to set up an incompressible, single-phase version of the full SPE 10 benchmark. Compute pressure, time-of-flight and tracer concentrations associated with each well. Hint: You may need to replace MATLAB's standard backslash-solver by a highly-efficient iterative solver like AGMG [238, 15] to get reasonable computational performance. Also, beware that you may run out of memory if your computer is not sufficiently powerful.