# 7

# Compressible Flow and Rapid Prototyping

In previous chapters we have outlined and explained in detail how to discretize and solve incompressible flow problems. This chapter will teach you how to discretize the basic equations for single-phase, compressible flow by use of the discrete differential and averaging operators that were introduced in Section 4.4.2. As briefly shown in Examples 4.4.2 and 4.4.3 in the same section, these discrete operators enable you to implement discretized flow equations in a compact form similar to the continuous mathematical description. Use of automatic differentiation (see Appendix A.5 for more details) then ensures that no analytical derivatives have to be programmed explicitly as long as the discrete flow equations and constitutive relationships are implemented as a sequence of algebraic operations. MRST makes it possible to combine discrete operators and automatic differentiation with a flexible grid structure, a highly vectorized and interactive scripting language, and a powerful graphical environment. This is in my opinion the main reason why the software has proved to be an efficient tool for developing new computational methods and workflow tools. In this chapter, I try to substantiate this claim by showing several examples of rapid prototyping. We first develop a compact and transparent solver for compressible flow and then extend the basic single-phase model to include pressure-dependent viscosity, non-Newtonian fluid behavior, and temperature effects. As usual, you can find complete scripts for all examples in a subdirectory (`ad-1ph`) of the `book` module.

## 7.1 Implicit Discretization

As our basic model, we consider the single-phase continuity equation,

$$\frac{\partial}{\partial t}(\phi\rho) + \nabla\cdot(\rho\vec{v}) = q, \qquad \vec{v} = -\frac{\mathbf{K}}{\mu}\left(\nabla p - g\rho\nabla z\right). \tag{7.1}$$

The primary unknown is usually the fluid pressure $p$. Additional equations are supplied to provide relations between $p$ and the other quantities in the equation, e.g., by using an equation of state to relate fluid density to pressure $\rho = \rho(p)$, specifying porosity as



https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

function of pressure $\phi(p)$ through a compressibility factor, and so on; see the discussion in Section 4.2. Notice also that $q$ is defined slightly differently in (7.1) than in (4.5).

Using the discrete operators introduced in Section 4.4.2, the basic implicit discretization of (7.1) reads

$$\frac{(\boldsymbol{\phi}\boldsymbol{\rho})^{n+1} - (\boldsymbol{\phi}\boldsymbol{\rho})^{n}}{\Delta t^n} + \texttt{div}(\boldsymbol{\rho}\boldsymbol{v})^{n+1} = \boldsymbol{q}^{n+1}, \tag{7.2a}$$

$$\boldsymbol{v}^{n+1} = -\frac{\boldsymbol{K}}{\boldsymbol{\mu}^{n+1}}\big[\texttt{grad}(\boldsymbol{p}^{n+1}) - g\boldsymbol{\rho}^{n+1}\texttt{grad}(z)\big]. \tag{7.2b}$$

Here, $\boldsymbol{\phi} \in \mathbb{R}^{n_c}$ denotes the vector with one porosity value per cell, $\boldsymbol{v}$ is the vector of fluxes per face, and so on. The superscript refers to discrete times at which one wishes to compute the unknown reservoir states and $\Delta t$ denotes the distance between two such consecutive points in time.

In many cases of practical interest it is possible to simplify (7.2). For instance, if the fluid is only slightly compressible, several terms can be neglected so that the nonlinear equation reduces to a linear equation in the unknown pressure $p^{n+1}$, which we can write on residual form as

$$\frac{\boldsymbol{p}^{n+1} - \boldsymbol{p}^{n}}{\Delta t^n} - \frac{1}{c_t \mu \boldsymbol{\phi}}\texttt{div}\big(\boldsymbol{K}\,\texttt{grad}(\boldsymbol{p}^{n+1})\big) - \boldsymbol{q}^{n} = 0. \tag{7.3}$$

The assumption of slight compressibility is not always applicable and for generality we assume that $\phi$ and $\rho$ depend nonlinearly on $p$ so that (7.2) gives rise to a nonlinear system of equations that needs to be solved in each time step. As we will see later in this chapter, viscosity may also depend on pressure, flow velocity, and/or temperature, which adds further nonlinearity to the system. If we now collect all the discrete equations, we can write the resulting system of nonlinear equations in short vector form as

$$\boldsymbol{F}(\boldsymbol{x}^{n+1}; \boldsymbol{x}^{n}) = \boldsymbol{0}. \tag{7.4}$$

Here, $\boldsymbol{x}^{n+1}$ is the vector of unknown state variables at the next time step and the vector of current states $\boldsymbol{x}^n$ can be seen as a parameter.

We will use the Newton–Raphson method to solve the nonlinear system (7.4): assume that we have a guess $\boldsymbol{x}_0$ and want to move this towards the correct solution, $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{0}$. To this end, we write $\boldsymbol{x} = \boldsymbol{x}_0 + \boldsymbol{\Delta x}$, use a Taylor expansion for linearization, and solve for the approximate increment $\boldsymbol{\delta x}$

$$\boldsymbol{0} = \boldsymbol{F}(\boldsymbol{x}_0 + \boldsymbol{\Delta x}) \approx \boldsymbol{F}(\boldsymbol{x}_0) + \nabla \boldsymbol{F}(\boldsymbol{x}_0)\boldsymbol{\delta x}.$$

This gives rise to an iterative scheme in which the approximate solution $\boldsymbol{x}^{i+1}$ in the $(i+1)$-th iteration is obtained from

$$\frac{d\boldsymbol{F}}{d\boldsymbol{x}}(\boldsymbol{x}^i)\delta\boldsymbol{x}^{i+1} = -\boldsymbol{F}(\boldsymbol{x}^i), \qquad \boldsymbol{x}^{i+1} \leftarrow \boldsymbol{x}^i + \boldsymbol{\delta x}^{i+1}. \tag{7.5}$$

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

Here, $\boldsymbol{J} = d\boldsymbol{F}/d\boldsymbol{x}$ is the Jacobian matrix, while $\boldsymbol{\delta x}^{i+1}$ is referred to as the *Newton update* at iteration number $i + 1$. Theoretically, the Newton process exhibits quadratic convergence under certain smoothness and differentiability requirements on $\boldsymbol{F}$. Obtaining such convergence in practice, however, will crucially depend on having a sufficiently accurate Jacobian matrix. For complex flow models, the computation of residual equations typically requires evaluation of many constitutive laws that altogether make up complex nonlinear dependencies. Analytical derivation and subsequent coding of the Jacobian can therefore be very time-consuming and prone to human errors. Fortunately, the computation of the Jacobian matrix can in almost all cases be broken down to nested differentiation of elementary operations and functions and is therefore a good candidate for automation using automatic differentiation. This will add an extra computational overhead to your code, but in most cases the increased CPU time is completely offset by the shorter time it takes you to develop a proof-of-concept code. Likewise, unless your model problem is very small, the dominant computational cost of solving a nonlinear PDE comes from the linear solver called within each Newton iteration.

The idea of using automatic differentiation to develop reservoir simulators is not new. This technique was introduced in an early version of the commercial Intersect simulator [80], but has mainly been pioneered through a reimplementation of the GPRS research simulator [58]. The new simulator, called AD-GPRS, is primarily based on fully implicit formulations [303, 325, 302], in which independent variables and residual equations are AD structures implemented using ADETL, a library for forward-mode AD realized by expression templates in C++ [323, 322]. This way, the Jacobi matrices needed in the nonlinear Newton-type iterations can be constructed from implicitly computed derivatives when evaluating the residual equations. In [185], the authors discuss how to use the alternative backward-mode differentiation to improve computational efficiency. Automatic differentiation is also used in the open-source `Flow` simulator from the Open Porous Media (OPM) initiative. `OPM Flow` can be considered as a C++ sibling of MRST, which originally used a similar vector-oriented AD library. This has later been replaced by a localized, cell-based AD library for improved efficiency.

## 7.2 A Simulator Based on Automatic Differentiation

We will now present step-by-step how you can use the AD library in MRST to implement an implicit solver for the compressible, single-phase continuity equation (7.1). In particular, we revisit the discrete spatial differentiation operators from Section 4.4.2 and introduce additional discrete averaging operators that together enable us to write the discretized equations in an abstract residual form that resembles the semi-continuous form of the implicit discretization in (7.2). Starting from this residual form, it is relatively simple to obtain a linearization using automatic differentiation and set up a Newton iteration.

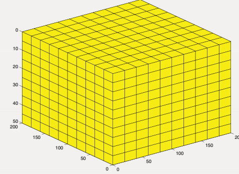
https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

### *7.2.1 Model Setup and Initial State*

For simplicity, we consider a homogeneous box model:

```
[nx,ny,nz] = deal( 10,  10, 10);
[Lx,Ly,Lz] = deal(200, 200, 50);
G = cartGrid([nx, ny, nz], [Lx, Ly, Lz]);
G = computeGeometry(G);

rock = makeRock(G, 30*milli*darcy, 0.3);
```



Beyond this point, our implementation is agnostic to details about the grid, except when we specify well positions on page 206, which would typically involve more code lines for a complex corner-point model like the Norne and SAIGUP models discussed in Sections 3.3.1 and 3.5.1.

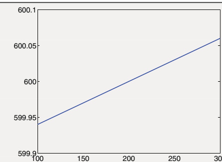
We assume constant rock compressibility $c_r$. Accordingly, the pore volume pv obeys the differential equation[1] $c_r \text{pv} = d\text{pv}/dp$ or

$$\text{pv}(p) = \text{pv}_r \, e^{c_r(p-p_r)}, \tag{7.6}$$

where $\text{pv}_r$ is the pore volume at reference pressure $p_r$. To define the relation between pore volume and pressure, we use an anonymous function:

```
cr   = 1e-6/barsa;
p_r  = 200*barsa;
pv_r = poreVolume(G, rock);

pv   = @(p) pv_r .* exp( cr * (p - p_r) );
```

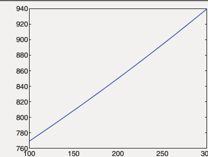



The fluid is assumed to have constant viscosity $\mu = 5$ cP, and as for the rock, we assume constant fluid compressibility $c$, resulting in the differential equation $c\rho = d\rho/dp$ for fluid density. Accordingly,

$$\rho(p) = \rho_r e^{c(p-p_r)}, \tag{7.7}$$

where $\rho_r$ is the density at reference pressure $p_r$. With this set, we can define the equation of state for the fluid:

```
mu    = 5*centi*poise;
c     = 1e-3/barsa;
rho_r = 850*kilogram/meter^3;
rhoS  = 750*kilogram/meter^3;
rho   = @(p) rho_r .* exp( c * (p - p_r) );
```




The assumption of constant compressibility will only hold for a limited range of temperatures. Moreover, surface conditions are not inside the validity range of the constant

[1] To highlight the close correspondence between the computer code and the mathematical equation, we here deliberately violate the advice to never use a compound symbol to denote a single mathematical quantity.

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

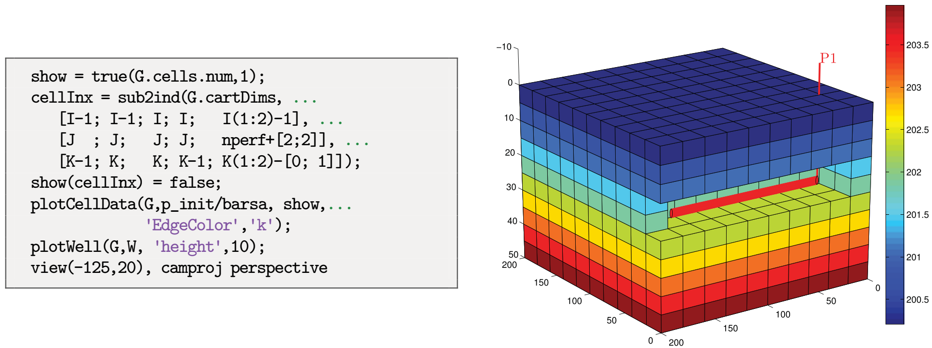



Figure 7.1 Model with initial pressure and single horizontal well.

compressibility assumption. We therefore set the fluid density $\rho_S$ at surface conditions separately because we will need it later to evaluate surface volume rate in our model of the well, which consists of a horizontal wellbore perforated in eight cells:

```
nperf = 8;
I = repmat(2, [nperf, 1]);
J = (1:nperf).'+1;
K = repmat(5, [nperf, 1]);
cellInx = sub2ind(G.cartDims, I, J, K);
W = addWell([ ], G, rock, cellInx, 'Name', 'producer', 'Dir', 'x');
```

Assuming the reservoir is initially at equilibrium implies that we must have $dp/dz = g\rho(p)$. In our simple setup, this differential equation can be solved analytically, but for demonstration purposes, we use one of MATLAB's built-in ODE-solvers to compute the hydrostatic distribution numerically, relative to a fixed datum point $p(z_0) = p_r$. Without lack of generality, we set $z_0 = 0$ since the reservoir geometry is defined relative to this height:

```
gravity reset on, g = norm(gravity);
[z_0, z_max] = deal(0, max(G.cells.centroids(:,3)));
equil  = ode23(@(z,p) g .* rho(p), [z_0, z_max], p_r);
p_init = reshape(deval(equil, G.cells.centroids(:,3)), [], 1);
```

This finishes the model setup, and at this stage we plot the reservoir with well and initial pressure as shown in Figure 7.1.

### 7.2.2 *Discrete Operators and Equations*

We are now ready to discretize the model. As seen in Section 4.4.2, the discrete version of the gradient operator maps from the set of cells to the set of faces. For a pressure field, it computes the pressure difference between neighboring cells of each face. Likewise, the discrete divergence operator is a linear mapping from the set of faces to the set of cells. For

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

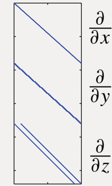
a flux field, it sums the outward fluxes for each cell. The complete code needed to form the `grad` and `div` operators has already been presented in Examples 4.4.2 and 4.4.3, but here we repeat it in order to make the example more self-contained.

To define the discrete operators, we must first compute the map between interior faces and cells

```
C  = double(G.faces.neighbors);
C  = C(all(C ~= 0, 2), :);
```

Exterior faces need not be included since they have zero flow, given our assumption of no-flow boundary conditions. It now follows that $\texttt{grad}(\boldsymbol{x}) = \boldsymbol{x}(C(:,2)) - \boldsymbol{x}(C(:,1)) = \boldsymbol{D}\boldsymbol{x}$, where $\boldsymbol{D}$ is a sparse matrix with values $\pm 1$ in columns $C(i,2)$ and $C(i,1)$ for row $i$. As a linear mapping, the discrete `div`-function is simply the negative transpose of `grad`; this follows from the discrete version of the Gauss–Green theorem, (4.58). In addition, we define an *averaging* operator that for each face computes the arithmetic average of the neighboring cells, which we will need to evaluate density values at grid faces:

```
n = size(C,1);
D = sparse([(1:n)'; (1:n)'], C, ...
           ones(n,1)*[-1 1], n, G.cells.num);
grad = @(x) D*x;
div  = @(x) -D'*x;
avg  = @(x) 0.5 * (x(C(:,1)) + x(C(:,2)));
```




This is all we need to define the spatial discretization for a homogeneous medium on a grid with cubic cells. To make a generic spatial discretization that also can account for more general cell geometries and heterogeneities, we must include transmissibilities. To this end, we first compute one-sided transmissibilities $T_{i,j}$ using the function `computeTrans`, which was discussed in detail in Section 5.2, and then use harmonic averaging to obtain face-transmissibilities. That is, for neighboring cells $i$ and $j$, we compute $T_{ij} = (T_{i,j}^{-1} + T_{j,i}^{-1})^{-1}$ as in (4.52) on page 133.

```
hT = computeTrans(G, rock);                   % Half-transmissibilities
cf = G.cells.faces(:,1);
nf = G.faces.num;
T  = 1 ./ accumarray(cf, 1 ./ hT, [nf, 1]); % Harmonic average
T  = T(intInx);                               % Restricted to interior
```

Having defined the necessary discrete operators, we are in a position to use the basic implicit discretization from (7.2). We start with Darcy's law (7.2b),

$$\vec{v}[f] = -\frac{\boldsymbol{T}[f]}{\mu}\big(\texttt{grad}(\boldsymbol{p}) - g\,\boldsymbol{\rho}_a[f]\,\texttt{grad}(z)\big), \tag{7.8}$$

where the density at the interface is evaluated using the arithmetic average

$$\boldsymbol{\rho}_a[f] = \tfrac{1}{2}\big(\boldsymbol{\rho}[C_1(f)] + \boldsymbol{\rho}[C_2(f)]\big). \tag{7.9}$$

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

Similarly, we can write the continuity equation for each cell $c$ as

$$\frac{1}{\Delta t}\Big[\big(\boldsymbol{\phi}(\boldsymbol{p})[c]\,\boldsymbol{\rho}(\boldsymbol{p})[c]\big)^{n+1} - \big(\boldsymbol{\phi}(\boldsymbol{p})[c]\,\boldsymbol{\rho}(\boldsymbol{p})[c]\big)^{n}\Big] + \texttt{div}(\boldsymbol{\rho}_a\boldsymbol{v})[c] = \mathbf{0}. \tag{7.10}$$

The two residual equations (7.8) and (7.10) are implemented as anonymous functions of pressure:

```
gradz = grad(G.cells.centroids(:,3));
v  = @(p)  -(T/mu).*( grad(p) - g*avg(rho(p)).*gradz );

presEq = @(p,p0,dt)  (1/dt)*(pv(p).*rho(p) - pv(p0).*rho(p0)) ...
                     + div( avg(rho(p)).*v(p) );
```

In the code above, `p0` is the pressure field at the *previous* time step (i.e., $\boldsymbol{p}^n$), whereas `p` is the pressure at the *current* time step ($\boldsymbol{p}^{n+1}$). Having defined the discrete expression for Darcy fluxes, we can check that this is in agreement with our initial pressure field by computing the magnitude of the flux, `norm(v(p_init))*day`. The result is $1.5 \times 10^{-6}\ \mathrm{m}^3/\mathrm{day}$, which should convince us that the initial state of the reservoir is sufficiently close to equilibrium.

### 7.2.3 Well Model

The production well will appear as a source term in the pressure equation. We therefore need to define an expression for flow rate in all cells the well is connected to the reservoir (which we refer to as well connections). Inside the well, we assume instantaneous flow so that the pressure drop is always hydrostatic. For a horizontal well, the hydrostatic term is zero and could obviously be disregarded, but we include it for completeness and as a robust precaution in case we later want to reuse the code with a different well path. Approximating the fluid density in the well as constant, computed at bottom-hole pressure, the pressure $\boldsymbol{p}_c[w]$ in connection $w$ of well $N_w(w)$ is given by

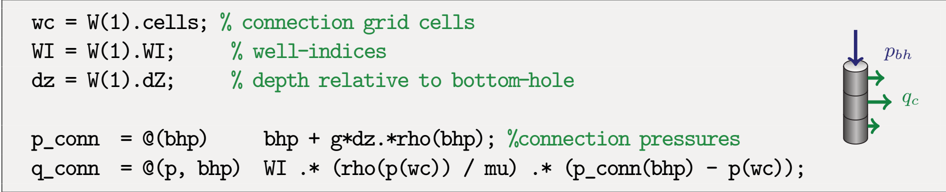
$$\boldsymbol{p}_c[w] = \boldsymbol{p}_{bh}[N_w(w)] + g\,\boldsymbol{\Delta z}[w]\,\rho(\boldsymbol{p}_{bh}[N_w(w)]), \tag{7.11}$$

where $\boldsymbol{\Delta z}[w]$ is the vertical distance from the bottom-hole to the connection. We use the standard Peaceman model introduced in Section 4.3.2 to relate the pressure at the well connection to the average pressure inside the grid cell. Using the well-indices from `W`, the mass flow-rate at connection $c$ reads

$$\boldsymbol{q}_c[w] = \frac{\rho(\boldsymbol{p}[N_c(w)])}{\mu}\,\mathrm{WI}[w]\,\big(\boldsymbol{p}_c[w] - \boldsymbol{p}[N_c(w)]\big), \tag{7.12}$$

where $\boldsymbol{p}[N_c(w)]$ is the pressure in cell $N_c(w)$ containing connection $w$. In our code, this model is implemented as follows:

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press



```
wc = W(1).cells; % connection grid cells
WI = W(1).WI;     % well-indices
dz = W(1).dZ;     % depth relative to bottom-hole

p_conn  = @(bhp)     bhp + g*dz.*rho(bhp); %connection pressures
q_conn  = @(p, bhp)  WI .* (rho(p(wc)) / mu) .* (p_conn(bhp) - p(wc));
```



We also include the total volumetric well-rate at surface conditions as a free variable. This is simply given by summing all mass well-rates and dividing by the surface density:

```
rateEq = @(p, bhp, qS)  qS-sum(q_conn(p, bhp))/rhoS;
```

With free variables p, bhp, and qS, we lack exactly one equation to close the system. This equation should account for *boundary conditions* in the form of a well control. Here, we choose to control the well by specifying a fixed bottom-hole pressure

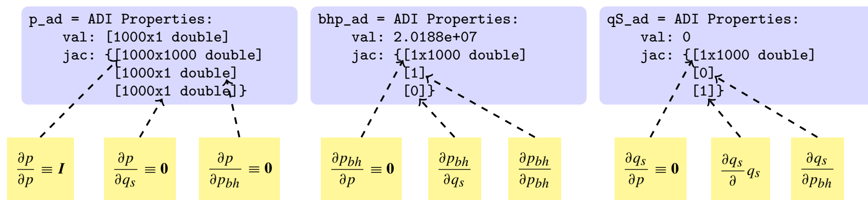
```
ctrlEq = @(bhp)  bhp-100*barsa;
```

### 7.2.4 The Simulation Loop

What now remains is to set up a simulation loop that will evolve the transient pressure. We start by initializing the AD variables. For clarity, we append _ad to all variable names to distinguish them from doubles. The initial bottom-hole pressure is set to the corresponding grid-cell pressure.

```
[p_ad, bhp_ad, qS_ad] = initVariablesADI(p_init, p_init(wc(1)), 0);
```

This gives the following AD pairs that make up the unknowns in our system:





To solve the global flow problem, we must stack all the equations into one big system, compute the corresponding Jacobian, and perform a Newton update. We therefore set indices for easy access to individual variables

```
[p_ad, bhp_ad, qS_ad] = initVariablesADI(p_init, p_init(wc(1)), 0);
nc = G.cells.num;
[pIx, bhpIx, qSIx] = deal(1:nc, nc+1, nc+2);
```

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

Next, we set parameters to control the time steps in the simulation and the iterations in the Newton solver:

```
[numSteps, totTime] = deal(52, 365*day);  %  time-steps/ total simulation time
[tol, maxits]  = deal(1e-5, 10)           % Newton tolerance / maximum Newton its
dt = totTime / numSteps;
```

Simulation results from all time steps are stored in a structure `sol`. For efficiency, this structure is preallocated and initialized so that the first entry is the initial state of the reservoir:

```
sol = repmat(struct('time',[], 'pressure',[], 'bhp',[], 'qS',[]), [numSteps+1, 1]);
sol(1) = struct('time', 0, 'pressure', double(p_ad), ...
                'bhp', double(bhp_ad), 'qS', double(qS_ad));
```

We now have all we need to set up the time-stepping algorithm, which consists of an outer and an inner loop. The outer loop updates the time step, advances the solution one step forward in time, and stores the result in the `sol` structure. This procedure is repeated until we reach the desired final time:

```
t = 0; step = 0;
while t < totTime,
   t = t + dt; step = step + 1;
   fprintf('\nTime step %d: Time %.2f -> %.2f days\n', ...
           step, convertTo(t - dt, day), convertTo(t, day));
   % Newton loop
   [resNorm, nit] = deal(1e99, 0);
   p0  = double(p_ad); % Previous step pressure
   while (resNorm > tol) && (nit <= maxits)
      : % Newton update
      :
      resNorm = norm(res);
      nit     = nit + 1;
      fprintf('  Iteration %3d:  Res = %.4e\n', nit, resNorm);
   end
   if nit > maxits, error('Newton solves did not converge')
   else % store solution
      sol(step+1)  = struct('time', t, 'pressure', double(p_ad), ...
                            'bhp', double(bhp_ad), 'qS', double(qS_ad));
   end
end
```

The inner loop performs the Newton iteration by computing and assembling the Jacobian of the global system and solving the linearized residual equation to compute an iterative

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

update. The first step to this end is to evaluate the residual for the flow pressure equation and add source terms from wells:

```
eq1      = presEq(p_ad, p0, dt);
eq1(wc) = eq1(wc) - q_conn(p_ad, bhp_ad);
```

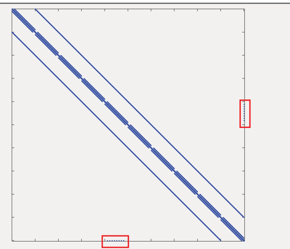
Most of the lines we have implemented so far are fairly standard, except perhaps for the definition of the residual equations as anonymous functions. Equivalent statements can be found in almost any computer program solving this type of time-dependent equation by an implicit method. Now, however, comes what is normally the tricky part: linearization of the equations that make up the whole model and assembly of the resulting Jacobian matrices to generate the Jacobian for the full system. And here you have the magic of automatic differentiation: *you do not have to do this at all!* The computer code necessary to evaluate all Jacobians has been defined implicitly by the functions in the AD library in MRST, which overloads the elementary operators used to define the residual equations. An example of a complete calling sequence for a simple calculation is shown in Figure A.7 on page 625. The sequence of operations we use to compute the residual equations is obviously more complex than this example, but the operators used are in fact only the three elementary operators plus, minus, and multiply applied to scalars, vectors, and matrices, as well as element-wise division by a scalar and evaluation of exponential functions. When the residuals are evaluated by use of the anonymous functions defined in Sections 7.2.2 and 7.2.3, the AD library also evaluates the derivatives of each equation with respect to each independent variable and collects the corresponding sub-Jacobians in a list. To form the full system, we simply evaluate the residuals of the remaining equations (the rate equation and the equation for well control) and concatenate the three equations into a cell array:

```
eqs = {eq1, rateEq(p_ad, bhp_ad, qS_ad), ctrlEq(bhp_ad)};
eq  = cat(eqs{:});
```

In doing this, the AD library will correctly combine the various sub-Jacobians and set up the Jacobian for the full system. Then, we can extract this Jacobian, solve for the Newton increment, and update the three primary unknowns:

```
J   = eq.jac{1};  % Jacobian
res = eq.val;      % residual
upd = -(J \ res); % Newton update

% Update variables
p_ad.val   = p_ad.val   + upd(pIx);
bhp_ad.val = bhp_ad.val + upd(bhpIx);
qS_ad.val  = qS_ad.val  + upd(qSIx);
```



The sparsity pattern of the Jacobian is shown in the plot to the left of the code for the Newton update. The use of a two-point scheme on a 3D Cartesian grid gives a Jacobi matrix that has a heptadiagonal structure, except for the off-diagonal entries in the two red rectangles. These arise from the well equation and correspond to derivatives of this equation with respect to cell pressures.

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

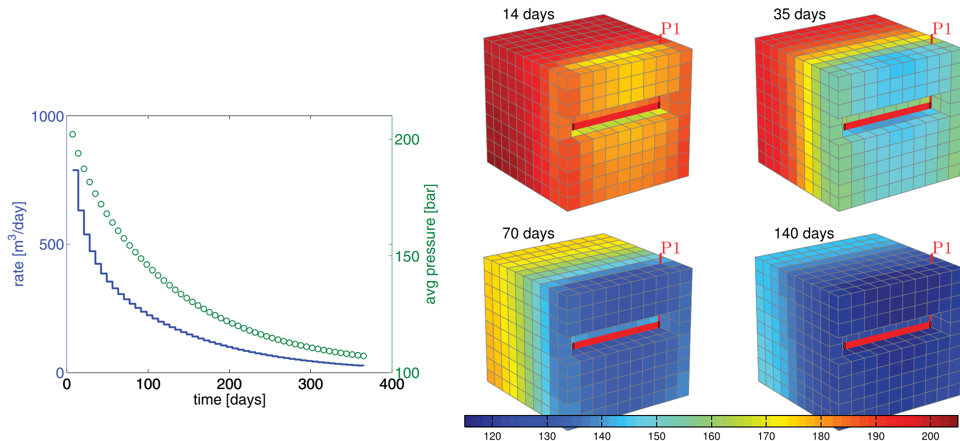



Figure 7.2 Time evolution of the pressure solution for the compressible single-phase problem. The plot to the left shows the well rate (blue line) and average reservoir pressure (green circles) as function of time, and the plots to the right show the pressure after 2, 5, 10, and 20 pressure steps.

Figure 7.2 plots how the dynamic pressure evolves with time. Initially, the pressure is in hydrostatic equilibrium as shown in Figure 7.1. When the well starts to drain the reservoir, the pressure drawdown near the well will start to gradually propagate outward from the well. As a result, the average pressure inside the reservoir is reduced, which again causes a decay in the production rate.

## Computer exercises

7.2.1 Apply the compressible pressure solver to the quarter five-spot problem from Section 5.4.1.

7.2.2 Rerun compressible simulations for the three different grid models that were derived from the `seamount` data set Section 5.4.3. Replace the fixed boundary conditions by a no-flow condition.

7.2.3 Use the implementation from Section 7.2 as a template to develop a solver for slightly compressible flow (7.3). More details about this model can be found on page 118 in Section 4.2. How large can $c_f$ be before the assumptions of slight compressibility become inaccurate? Use different heterogeneities, well placements, and model geometries to investigate this.

7.2.4 Extend the compressible solver developed in this section to incorporate other boundary conditions than no flow.

7.2.5 Try to compute time-of-flight by extending the equation set to also include the time-of-flight equation (4.40). Hint: the time-of-flight and the pressure equations need not be solved as a coupled system.

7.2.6 Same as the previous exercise, except that you should try to reuse the solver from in Section 5.3. Hint: you must first reconstruct fluxes from the computed pressure and then construct a state object to communicate with the TOF solver.

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

## 7.3 Pressure-Dependent Viscosity

One particular advantage of using automatic differentiation in combination with the discrete differential and averaging operators is that it simplifies the testing of new models and alternative computational approaches. In this section, we discuss two examples that hopefully demonstrate this aspect.

In the model discussed in the previous section, the viscosity was assumed to be constant. However, viscosity will generally increase with increasing pressures and this effect may be significant for the high pressures seen inside a reservoir, as we will see later in the book when discussing black-oil models in Chapter 11. To illustrate, we introduce a linear dependence, rather than the exponential pressure-dependence used for pore volume (7.6) and the fluid density (7.7). That is, we assume the viscosity is given by

$$\mu(p) = \mu_0\big[1 + c_\mu(p - p_r)\big]. \tag{7.13}$$

Having a pressure dependence means that we have to change two parts of our discretization: the approximation of the Darcy flux across a cell face (7.8) and the flow rate through a well connection (7.12). Starting with the latter, we evaluate the viscosity using the same pressure as was used to evaluate the density, i.e.,

$$\boldsymbol{q}_c[w] = \frac{\rho(\boldsymbol{p}[N_c(w)])}{\mu(\boldsymbol{p}[N_c(w)])}\,\mathrm{WI}[w]\,\big(\boldsymbol{p}_c[w] - \boldsymbol{p}[N_c(w)]\big). \tag{7.14}$$

For the Darcy flux (7.8), we have two choices: either use a simple arithmetic average as in (7.9) to approximate the viscosity at each cell face,

$$\boldsymbol{v}[f] = -\frac{\boldsymbol{T}[f]}{\boldsymbol{\mu}_a[f]}\big(\mathtt{grad}(\boldsymbol{p}) - g\,\boldsymbol{\rho}_a[f]\,\mathtt{grad}(z)\big), \tag{7.15}$$

or replace the quotient of the transmissibility and the face viscosity by the harmonic average of the mobility $\lambda = \mathbf{K}/\mu$ in the adjacent cells. Both choices introduce changes in the structure of the discrete nonlinear system, but because we are using automatic differentiation, all we have to do is code the corresponding formulas. Let us look at the details of the implementation, starting with the arithmetic approach.

*Arithmetic Average*

First, we introduce a new anonymous function to evaluate the relation between viscosity and pressure:

```
[mu0,c_mu] = deal(5*centi*poise, 2e-3/barsa);
mu = @(p) mu0*(1+c_mu*(p-p_r));
```

Then, we can replace the definition of the Darcy flux (changes marked in red):

```
v  = @(p)  -(T./mu(avg(p))).*( grad(p) - g*avg(rho(p)).*gradz );
```

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

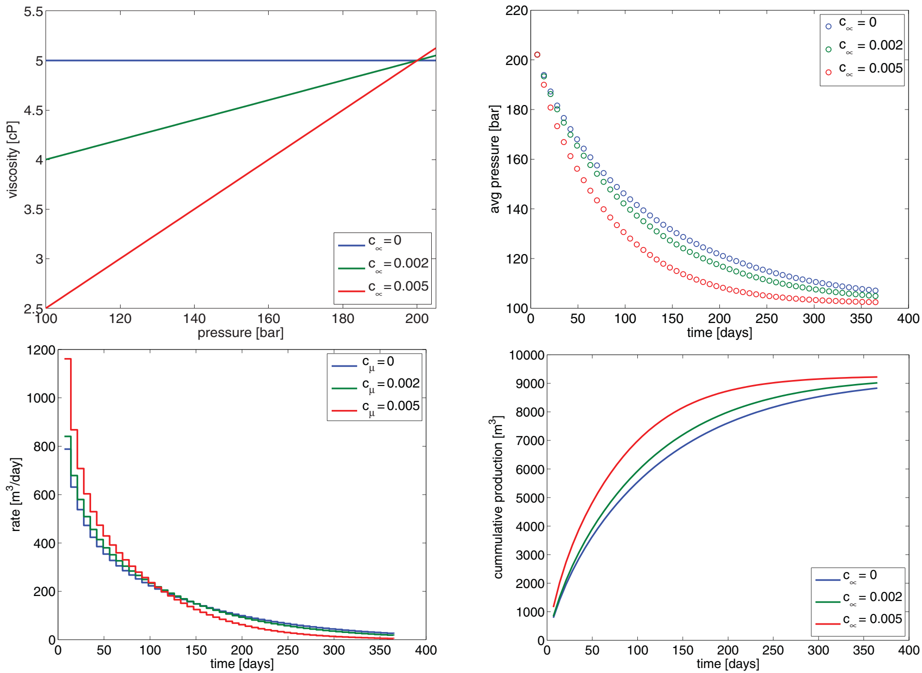



Figure 7.3 The effect of increasing the degree of pressure-dependence for the viscosity.

and similarly for flow rate through each well connection:

```
q_conn  = @(p,bhp) WI.*(rho(p(wc))./ mu(p(wc))) .* (p_conn(bhp) - p(wc));
```

Figure 7.3 illustrates the effect of increasing the pressure dependence of the viscosity. Since the reference value is given at $p = 200$ bar, which is close to the initial pressure inside the reservoir, the more we increase $c_\mu$, the lower $\mu$ will be in the pressure-drawdown zone near the well. Hence, we see a significantly higher initial production rate for $c_\mu = 0.005$ than for $c_\mu = 0$. On the other hand, the higher value of $c_\mu$, the faster the drawdown effect of the well will propagate into the reservoir, inducing a reduction in reservoir pressure that eventually will cause production to cease. In terms of overall production, a stronger pressure dependence may be more advantageous as it leads to a higher total recovery and higher cumulative production early in the production period.

*Face Mobility: Harmonic Average*

A more correct approximation is to write Darcy's law based on mobility instead of using the quotient of the transmissibility and an averaged viscosity:

$$\boldsymbol{v}[f] = -\boldsymbol{\Lambda}[f]\big(\texttt{grad}(\boldsymbol{p}) - g\,\boldsymbol{\rho}_a[f]\,\texttt{grad}(\boldsymbol{z})\big). \tag{7.16}$$

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

The face mobility $\boldsymbol{\Lambda}[f]$ can be defined in the same way as the transmissibility is defined in terms of the half transmissibilities using harmonic averages. That is, if $\boldsymbol{T}[f,c]$ denotes the half transmissibility associated with face $f$ and cell $c$, the face mobility $\boldsymbol{\Lambda}[f]$ for face $f$ can be written as

$$\boldsymbol{\Lambda}[f] = \left(\frac{\boldsymbol{\mu}[C_1(f)]}{\boldsymbol{T}[f,C_1(f)]} + \frac{\boldsymbol{\mu}[C_2(f)]}{\boldsymbol{T}[f,C_2(f)]}\right)^{-1}. \tag{7.17}$$

In MRST, the corresponding code reads:

```
hf2cn = getCellNoFaces(G);
nhf   = numel(hf2cn);
hf2f  = sparse(double(G.cells.faces(:,1)),(1:nhf)',1);
hf2if = hf2f(intInx,:);
fmob  = @(mu,p) 1./(hf2if*(mu(p(hf2cn))./hT));

v   = @(p) -fmob(mu,p).*( grad(p) - g*avg(rho(p)).*gradz );
```

Here, `hf2cn` represents the maps $C_1$ and $C_2$, which enable us to sample the viscosity value in the correct cell for each half-face transmissibility, whereas `hf2if` represents a map from half-faces (i.e., faces seen from a single cell) to global faces (which are shared by two cells). The map has a unit value in row $i$ and column $j$ if half-face $j$ belongs to global face $i$. Hence, premultiplying a vector of half-face quantities by `hf2if` amounts to summing the contributions from cells $C_1(f)$ and $C_2(f)$ for each global face $f$.

Using the harmonic average for a homogeneous model should produce simulation results that are identical (to machine precision) to those produced by the arithmetic average. With heterogeneous permeability, there will be small differences in well rates and averaged pressures for the specific parameters considered herein. For sub-samples of the SPE 10 data set, we typically observe maximum relative differences in well rates of the order $10^{-3}$.

#### Computer exercises

7.3.1 Investigate the claim that the difference between using an arithmetic average of the viscosity and a harmonic average of the fluid mobility is typically small. To this end, you can for instance use the following sub-sample from the SPE 10 data set: `rock = getSPE10rock(41:50,101:110,1:10)`.

## 7.4 Non-Newtonian Fluid

Viscosity is the material property that measures a fluid's resistance to flow, i.e., the resistance to a change in shape, or to the movement of neighboring portions of the fluid relative to each other. The more viscous a fluid is, the less easily it will flow. In Newtonian fluids, the shear stress or the force applied per area tangential to the force at any point is proportional to the strain rate (the symmetric part of the velocity gradient) at that point, and the viscosity is the constant of proportionality. For non-Newtonian fluids, the relationship is no longer

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

linear. The most common nonlinear behavior is *shear thinning*, in which the viscosity of the system decreases as the shear rate increases. An example is paint, which should flow easily when leaving the brush, but stay on the surface and not drip once it has been applied. The second type of nonlinearity is *shear thickening*, in which the viscosity increases with increasing shear rate. A common example is the mixture of cornstarch and water. If you search YouTube for "cornstarch pool" you will find several spectacular videos of pools filled with this mixture. When stress is applied to the mixture, it exhibits properties like a solid and you may be able to run across its surface. However, if you go too slow, the fluid behaves more like a liquid and you fall in.

Solutions of large polymeric molecules are another example of shear-thinning liquids. In enhanced oil recovery, polymer solutions may be injected into reservoirs to improve unfavorable mobility ratios between oil and water and improve the sweep efficiency of the injected fluid. At low flow rates, the polymer molecule chains tumble around randomly and present large resistance to flow. When the flow velocity increases, the viscosity decreases as the molecules gradually align themselves in the direction of increasing shear rate. A model of the rheology is given by

$$\mu = \mu_\infty + (\mu_0 - \mu_\infty)\left(1 + \left(\frac{K_c}{\mu_0}\right)^{\frac{2}{n-1}} \dot{\gamma}^2\right)^{\frac{n-1}{2}}, \tag{7.18}$$

where $\mu_0$ represents the Newtonian viscosity at zero shear rate, $\mu_\infty$ represents the Newtonian viscosity at infinite shear rate, $K_c$ represents the consistency index, and $n$ represents the power-law exponent ($n < 1$). The shear rate $\dot{\gamma}$ in a porous medium can be approximated by

$$\dot{\gamma}_{\text{app}} = 6\left(\frac{3n+1}{4n}\right)^{\frac{n}{n-1}} \frac{|\vec{v}|}{\sqrt{\mathbf{K}\phi}}. \tag{7.19}$$

Combining (7.18) and (7.19), we can write our model for the viscosity as

$$\mu = \mu_0\left(1 + \bar{K}_c \frac{|\vec{v}|^2}{\mathbf{K}\phi}\right)^{\frac{n-1}{2}}, \quad \bar{K}_c = 36\left(\frac{K_c}{\mu_0}\right)^{\frac{2}{n-1}}\left(\frac{3n+1}{4n}\right)^{\frac{2n}{n-1}}, \tag{7.20}$$

where we for simplicity have assumed that $\mu_\infty = 0$.

*Rapid Development of Proof-of-Concept Codes*

We now demonstrate how easy it is to extend the simple simulator developed so far in this chapter to model non-Newtonian fluid behavior (see `nonNewtonianCell.m`). To simulate injection, we increase the bottom-hole pressure to 300 bar. Our rheology model has parameters:

```
mu0 = 100*centi*poise;
nmu = .3;
Kc  = .1);
Kbc = (Kc/mu0)^(2/(nmu-1))*36*((3*nmu+1)/(4*nmu))^(2*nmu/(nmu-1));
```

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

In principle, we could continue to solve the system using the same primary unknowns as before. However, it has proved convenient to write (7.20) in the form $\mu = \eta\,\mu_0$, and introduce $\eta$ as an additional unknown. In each Newton step, we start by solving the equation for the shear factor $\eta$ exactly for the given pressure distribution. This is done by initializing an AD variable for $\eta$, but not for $p$ in `etaEq` so that this residual now only has one unknown, $\eta$. This will take out the implicit nature of Darcy's law and hence reduce the nonlinearity and simplify the solution of the global system.

```
while (resNorm > tol) && (nit < maxits)
  % Newton loop for eta (shear multiplier)
  [resNorm2,nit2] = deal(1e99, 0);
  eta_ad2 = initVariablesADI(eta_ad.val);
  while (resNorm2 > tol) && (nit2 <= maxits)
    eeq = etaEq(p_ad.val, eta_ad2);
    res = eeq.val;
    eta_ad2.val = eta_ad2.val - (eeq.jac{1} \ res);
    resNorm2 = norm(res);
    nit2     = nit2+1;
  end
  eta_ad.val = eta_ad2.val;
```

Once the shear factor has been computed for the values in the previous iterate, we can use the same approach as earlier to compute a Newton update for the full system. (Here, `etaEq` is treated as a system with two unknowns, $p$ and $\eta$.)

```
eq1     = presEq(p_ad, p0, eta_ad, dt);
eq1(wc) = eq1(wc) - q_conn(p_ad, eta_ad, bhp_ad);
eqs = {eq1, etaEq(p_ad, eta_ad), ...
    rateEq(p_ad, eta_ad, bhp_ad, qS_ad), ctrlEq(bhp_ad)};
eq  = cat(eqs{:});
upd = -(eq.jac{1} \ eq.val); % Newton update
```

To finish the solver, we need to define the flow equations and the extra equation for the shear multiplier. The main question now is how we should compute $|\vec{v}|$? One solution could be to define $|\vec{v}|$ on each face as the flux divided by the face area. In other words, use a code like

```
phiK   = avg(rock.perm.*rock.poro)./G.faces.areas(intInx).^2;
v      = @(p, eta)   -(T./(mu0*eta)).*( grad(p) - g*avg(rho(p)).*gradz );
etaEq  = @(p, eta)   eta - (1 + Kbc*v(p,eta).^2./phiK).^((nmu-1)/2);
```

Although simple, this approach has three potential issues: First, it does not tell us how to compute the shear factor for the well perforations. Second, it disregards contributions from any tangential components of the velocity field. Third, the number of unknowns in the linear system increases by almost a factor six since we now have one extra unknown per internal face. The first issue is easy to fix: To get a representative value in the well cells, we simply average the $\eta$ values from the cells' faces. If we now recall how the discrete

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

divergence operator was defined, we realize that this operation is almost implemented for us already: if `div(x)=-D'*x` computes the discrete divergence in each cell of the field `x` defined at the faces, then `wavg(x)=1/6*abs(D)'*x` computes the average of `x` for each cell. In other words, our well equation becomes:

```
wavg   = @(eta) 1/6*abs(D(:,W.cells))'*eta;
q_conn = @(p, eta, bhp) ...
  WI .* (rho(p(wc)) ./ (mu0*wavg(eta))) .* (p_conn(bhp) - p(wc));
```

The second issue would have to be investigated in more detail, and this is not within the scope of this book. The third issue is simply a disadvantage.

To get a method that consumes less memory, we can compute one $\eta$ value per cell. Using the following formula, we can reconstruct an approximate velocity $\vec{v}_i$ at the center of cell $i$

$$\vec{v}_i = \sum_{j \in N(i)} \frac{v_{ij}}{V_i}\left(\vec{c}_{ij} - \vec{c}_i\right), \tag{7.21}$$

where $N(i)$ is the map from cell $i$ to its neighboring cells, $v_{ij}$ is the flux between cell $i$ and cell $j$, $\vec{c}_{ij}$ is the centroid of the corresponding face, and $\vec{c}_i$ is the centroid of cell $i$. For a Cartesian grid, this formula simplifies so that an approximate velocity can be obtained as the sum of the absolute value of the flux divided by the face area over all faces that make up a cell. Using a similar trick as we used in the `wavg` operator to compute $\eta$ in the well cells, our implementation follows trivially. We first define the averaging operator to compute cell velocity

```
aC = bsxfun(@rdivide, 0.5*abs(D), G.faces.areas(intInx))';
cavg = @(x) aC*x;
```

In doing so, we also rename our old averaging operator `avg` as `favg` to avoid confusion and make it more clear that this operator maps from cell values to face values. Then we can define the needed equations:

```
phiK  = rock.perm.*rock.poro;
gradz = grad(G.cells.centroids(:,3));
v = @(p, eta)  -(T./(mu0*favg(eta))).*( grad(p) - g*favg(rho(p)).*gradz );
etaEq = @(p, eta)
   eta - ( 1 + Kbc* cavg(v(p,eta)).^2 ./phiK ).^((nmu-1)/2);
presEq= @(p, p0, eta, dt)  ...
   (1/dt)*(pv(p).*rho(p) - pv(p0).*rho(p0)) + div(favg(rho(p)).*v(p, eta));
```

With this approach, the well equation becomes particularly simple, since all we need to do is sample the $\eta$ value from the correct cell:

```
q_conn = @(p, eta, bhp) ...
    WI .* (rho(p(wc)) ./ (mu0*eta(wc))) .* (p_conn(bhp) - p(wc));
```

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

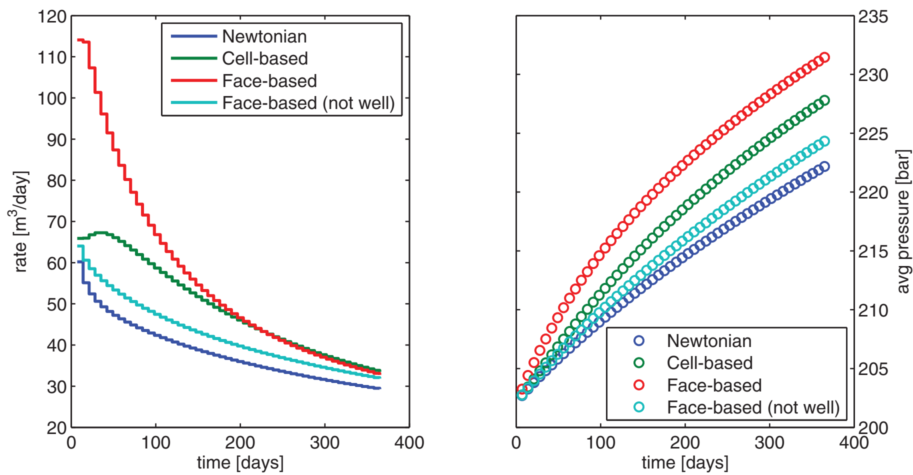



Figure 7.4 Single-phase injection of a highly viscous, shear-thinning fluid computed by four different simulation methods: (i) fluid assumed to be Newtonian, (ii) shear multiplier $\eta$ computed in cells, (iii) shear multiplier computed at faces, and (iv) shear multiplier computed at faces, but $\eta \equiv 1$ used in well model.

A potential drawback of this second approach is that it may introduce numerical smearing, but this will, on the other hand, most likely increase the robustness of the resulting scheme.

Figure 7.4 compares the predicted flow rates and average reservoir pressure for two different fluid models: one that assumes a standard Newtonian fluid (i.e., $\eta \equiv 1$) and one that models shear thinning. With shear thinning, the higher pressure in the injection well causes a decrease in the viscosity, which leads to significantly higher injection rates than for the Newtonian fluid and hence a higher average reservoir pressure. Perhaps more interesting is the large discrepancy in rates and pressures predicted by the face-based and cell-based simulation algorithms. If we disregard the shear multiplier `q_conn` in the face-based method, the predicted rate and pressure buildup is smaller than what is predicted by the cell-based method, and closer to the Newtonian fluid case. We take this as evidence that the differences between the cell and the face-based methods to a large extent can be explained by differences in the discretized well models and their ability to capture the formation and propagation of the strong initial transient. To further back this up, we have included results from a simulation with ten times as many time steps in Figure 7.5, which also includes plots of the evolution of $\min(\eta)$ as function of time. Whereas the face-based method predicts a large, immediate drop in viscosity in the near-well region, the viscosity drop predicted by the cell-based method is much smaller during the first 20–30 days. This results in a delay in the peak of the injection rate and a much smaller injected volume.

We leave the discussion here. The parameters used in the example were chosen quite haphazardly to demonstrate a pronounced shear-thinning effect. Which method is the most correct for real computations is a question that goes beyond the current scope, and could

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

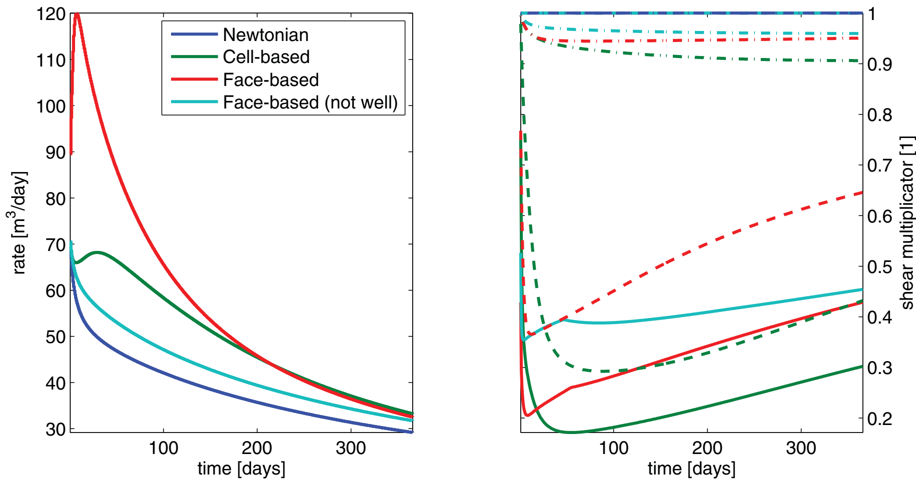



Figure 7.5 Single-phase injection of a highly viscous, shear-thinning fluid; simulation with $\Delta t = 1/520$ year. The right plot shows the evolution of $\eta$ as a function of time: solid lines show $\min(\eta)$ over all cells, dashed lines $\min(\eta)$ over the perforated cells, and dash-dotted lines average $\eta$ value.

probably best be answered by verifying against observed data for a real case. Our point here was mainly to demonstrate the capability for rapid implementation of proof-of-concept codes that comes with the use of MRST. However, as the example shows, this lunch is not completely free: you still have to understand features and limitations of the models and discretizations you choose to implement.

#### Computer exercises

7.4.1 Investigate whether the large differences observed in Figures 7.4 and 7.5 between the cell-based and face-based approaches to the non-Newtonian flow problem is a result of insufficient grid resolution.

7.4.2 The non-Newtonian fluid has a strong transient during the first 30–100 days. Try to implement adaptive time steps that utilize this fact. Can you come up with a strategy that automatically choose good time steps?

## 7.5 Thermal Effects

As another example of rapid prototyping, we extend the single-phase flow model (7.1) to account for thermal effects. That is, we assume that $\rho(p,T)$ is now a function of pressure and temperature $T$ and extend our model to also include conservation on energy,

$$\frac{\partial}{\partial t}\big[\phi\rho\big] + \nabla\cdot\big[\rho\vec{v}\big] = q, \qquad \vec{v} = -\frac{\mathbf{K}}{\mu}\big[\nabla p - g\rho\nabla z\big], \tag{7.22a}$$

$$\frac{\partial}{\partial t}\big[\phi\rho E_f(p,t) + (1-\phi)E_r\big] + \nabla\cdot\big[\rho H_f\vec{v}\big] - \nabla\cdot\big[\boldsymbol{\kappa}\nabla T\big] = q_e. \tag{7.22b}$$

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

Here, the rock and the fluid are assumed to be in local thermal equilibrium. In the energy equation (7.22b), $E_f$ is energy density per mass of the fluid, $H_f = E_f + p/\rho$ is enthalpy density per mass, $E_r$ is energy per volume of the rock, and $\kappa$ is the heat conduction coefficient of the rock. Fluid pressure $p$ and temperature $T$ are used as primary variables.

As in the original isothermal simulator, we must first define constitutive relationships that express the new physical quantities in terms of the primary variables. The energy equation includes heating of the solid rock, and we therefore start by defining a quantity that keeps track of the solid volume, which also depends on pressure:

```
sv = @(p) G.cells.volumes - pv(p);
```

For the fluid model, we use

$$\begin{aligned} \rho(p,T) &= \rho_r\big[1 + \beta_T(p - p_r)\big]e^{-\alpha(T-T_r)}, \\ \mu(p,T) &= \mu_0\big[1 + c_\mu(p - p_r)\big]e^{-c_T(T-T_r)}. \end{aligned} \tag{7.23}$$

Here, $\rho_r = 850$ kg/m$^3$ is the density and $\mu_0 = 5$ cP the viscosity of the fluid at reference conditions with pressure $p_r = 200$ bar and temperature $T_r = 300$ K. The constants are $\beta_T = 10^{-3}$ bar$^{-1}$, $\alpha = 5 \times 10^{-3}$ K$^{-1}$, $c_\mu = 2 \times 10^{-3}$ bar$^{-1}$, and $c_T = 10^{-3}$ K$^{-1}$. This translates to the following code:

```
[mu0,cmup]  = deal( 5*centi*poise, 2e-3/barsa);
[cmut,T_r]  = deal( 1e-3, 300);
mu  = @(p,T)  mu0*(1+cmup*(p-p_r)).*exp(-cmut*(T-T_r));

[alpha, beta] = deal(5e-3, 1e-3/barsa);
rho_r = 850*kilogram/meter^3;
rho   = @(p,T) rho_r .* (1+beta*(p-p_r)) .* exp(-alpha*(T-T_r));
```

We use a simple linear relation for the enthalpy, which is based on the thermodynamical relations that give

$$\mathrm{d}H_f = c_p\,\mathrm{d}T + \left(\frac{1-\alpha T_r}{\rho}\right)\mathrm{d}p, \qquad \alpha = -\frac{1}{\rho}\frac{\partial \rho}{\partial T}\bigg|_p, \tag{7.24}$$

where $c_p = 4 \times 10^3$ J/kg. The code for enthalpy/energy densities reads:

```
Cp = 4e3;
Hf = @(p,T) Cp*T+(1-T_r*alpha).*(p-p_r)./rho(p,T);
Ef = @(p,T) Hf(p,T) - p./rho(p,T);
Er = @(T)   Cp*T;
```

We defer discussing details of these new relationships and only note that it is important that the thermal potentials $E_f$ and $H_f$ are consistent with the equation of state $\rho(p,T)$ to get a physically meaningful model.

Having defined all constitutive relationships in terms of anonymous functions, we can set up the equation for mass conservation and Darcy's law (with transmissibility renamed to Tp to avoid name clash with temperature):

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

```
v   = @(p,T) -(Tp./mu(avg(p),avg(T))).*(grad(p) - avg(rho(p,T)).*gdz);
pEq = @(p,T,p0,T0,dt) ...
       (1/dt)*(pv(p).*rho(p,T) - pv(p0).*rho(p0,T0)) ...
       + div( avg(rho(p,T)).*v(p,T) );
```

In the energy equation (7.22b), the accumulation and the heat-conduction terms are on the same form as the operators appearing in (7.22a) and can hence be discretized in the same way. We use an artificial "rock object" to compute transmissibilities for $\kappa$ instead of $\mathbf{K}$:

```
tmp = struct('perm',4*ones(G.cells.num,1));
hT  = computeTrans(G, tmp);
Th  = 1 ./ accumarray(cf, 1 ./ hT, [nf, 1]);
Th  = Th(intInx);
```

The remaining term in (7.22b), $\nabla \cdot [\rho H_f \vec{v}]$, represents advection of enthalpy and has a differential operator on the same form as the transport equations discussed in Section 4.4.3 and must hence be discretized by an upwind scheme. To this end, we introduce a new discrete operator that will compute the correct upwind value for the enthalpy density,

$$\texttt{upw}(\boldsymbol{H})[f] = \begin{cases} \boldsymbol{H}[C_1(f)], & \text{if } \boldsymbol{v}[f] > 0, \\ \boldsymbol{H}[C_2(f)], & \text{otherwise.} \end{cases} \tag{7.25}$$

With this, we can set up the energy equation in residual form in the same way as we previously have done for Darcy's law and mass conservation:

```
upw  = @(x,flag)   x(C(:,1)).*double(flag)+x(C(:,2)).*double(~flag);
hEq  = @(p, T, p0, T0, dt) ...
   (1/dt)*(pv(p ).*rho(p, T ).*Ef(p ,T ) + sv(p ).*Er(T ) ...
         - pv(p0).*rho(p0,T0).*Ef(p0,T0) - sv(p0).*Er(T0)) ...
   + div( upw(Hf(p,T),v(p,T)>0).*avg(rho(p,T)).*v(p,T) ) ...
   + div( -Th.*grad(T));
```

With this, we are almost done. As a last technical detail, we must also make sure that the energy transfer in injection and production wells is modeled correctly using appropriate upwind values:

```
qw       = q_conn(p_ad, T_ad, bhp_ad);
eq1      = pEq(p_ad, T_ad, p0, T0, dt);
eq1(wc)  = eq1(wc) - qw;
hq       = Hf(bhp_ad,bhT).*qw;
Hcells   = Hf(p_ad,T_ad);
hq(qw<0) = Hcells(wc(qw<0)).*qw(qw<0);
eq2      = hEq(p_ad,T_ad, p0, T0,dt);
eq2(wc)  = eq2(wc) - hq;
```

Here, we evaluate the enthalpy using *cell values* for pressure and temperature for production wells (for which `qw<0`) and pressure and temperatures at the *bottom hole* for injection wells.

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

What remains are trivial changes to the iteration loop to declare the correct variables as AD structures, evaluate the discrete equations, collect their residuals, and update the state variables. These details can be found in the complete code given in `singlePhaseThermal.m` and have been left out for brevity.

*Understanding Thermal Expansion*

Except for the modifications discussed in the previous subsection, the setup is the exact same as in Section 7.2. That is, the reservoir is a $200 \times 200 \times 50$ m$^3$ rectangular box with homogeneous permeability of 30 mD, constant porosity 0.3, and a rock compressibility of $10^{-6}$ bar$^{-1}$, realized on a $10 \times 10 \times 10$ Cartesian grid. The reservoir is realized on a $10 \times 10 \times 10$ Cartesian grid. Fluid is drained from a horizontal well perforated in cells with indices $i = 2$, $j = 2, \ldots, 9$, and $k = 5$, and operating at a constant bottom-hole pressure of 100 bar. Initially, the reservoir has constant temperature of 300 K and is in hydrostatic equilibrium with a datum pressure of 200 bar specified in the uppermost cell centroids.

In the same way as in the isothermal case, the open well will create a pressure drawdown that propagates into the reservoir. As more fluid is produced from the reservoir, the pressure will gradually decay towards a steady state with pressure values between 101.2 and 104.7 bar. Figure 7.6 shows that the simulation predicts a faster pressure drawdown, and hence a faster decay in production rates, if thermal effects are taken into account.

The change in temperature of an expanding fluid will not only depend on the initial and final pressure, but also on the type of process in which the temperature is changed:

- In a *free expansion*, the internal energy is preserved and the fluid does no work. That is, the process can be described by the following differential:

$$\frac{\mathrm{d}E_f}{\mathrm{d}p}\Delta p + \frac{\mathrm{d}E_f}{\mathrm{d}T}\Delta T = 0. \tag{7.26}$$

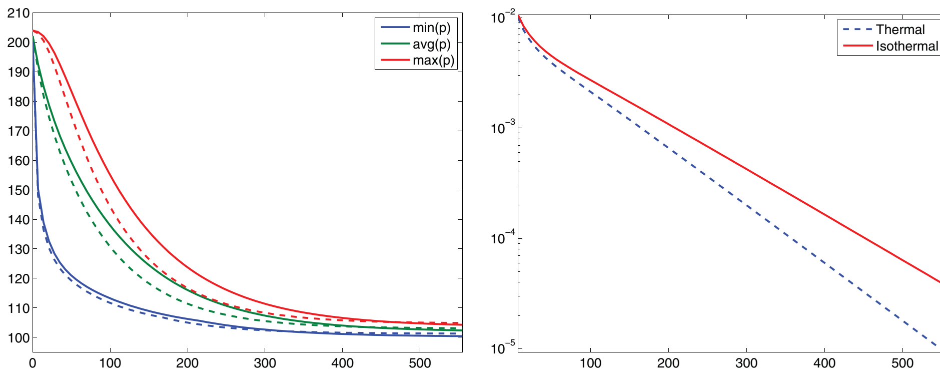



Figure 7.6 To the left, time evolution for pressure for an isothermal simulation (solid lines) and a thermal simulation with $\alpha = 5 \times 10^{-3}$ (dashed lines). To the right, decay in production rate at the surface.

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

When the fluid is an ideal gas, the temperature is constant, but otherwise the temperature will either increase or decrease during the process depending on the initial temperature and pressure.

- In a reversible process, the fluid is in thermodynamical equilibrium and does positive work while the temperature decreases. The linearized function associated with this *adiabatic expansion* reads

$$\mathrm{d}E + \frac{p}{\rho V}\,\mathrm{d}V = \mathrm{d}E + p\,\mathrm{d}\left(\frac{1}{\rho}\right) = 0. \tag{7.27}$$

- In a Joule–Thomson process, the enthalpy remains constant while the fluid flows from higher to lower pressure under steady-state conditions and without change in kinetic energy. That is,

$$\frac{\mathrm{d}H_f}{\mathrm{d}p}\Delta p + \frac{\mathrm{d}H_f}{\mathrm{d}T}\Delta T = 0. \tag{7.28}$$

Our case is a combination of these three processes and their interplay will vary with the initial temperature and pressure as well as with the constants in the fluid model for $\rho(p,T)$. To better understand a specific case, we can use (7.26–7.28) to compute the temperature change that would take place for an observed pressure drawdown if only one of the processes took place. Computing such linearized responses for thermodynamical functions is particularly simple using automatic differentiation. Assuming we know the reference state $(p_r, T_r)$ at which the process starts and the pressure $p_e$ after the process has taken place, we initialize the AD variables and compute the pressure difference:

```
[p,T] = initVariablesADI(p_r,T_r);
dp    = p_e - p_r;
```

Then, we can solve (7.26) or (7.28) for $\Delta T$ and use the result to compute the temperature change resulting from a free expansion or a Joule–Thomson expansion:

```
E    = Ef(p,T);  dEdp = E.jac{1}; dEdT = E.jac{2};
Tfr  = T_r - dEdp*dp/dEdT;

hf   = Hf(p,T);  dHdp = hf.jac{1}; dHdT = hf.jac{2};
Tjt  = T_r - dHdp*dp/dHdT;
```

The temperature change after a reversible (adiabatic) expansion is not described by a total differential. In this case we have to specify that $p$ should be kept constant. This is done by replacing the AD variable `p` by an ordinary variable `double(p)` in the code at the specific places where $p$ appears in front of a differential; see (7.27).

```
E   = Ef(p,T) + double(p)./rho(p,T);
dEdp = hf.jac{1};
dEdT = hf.jac{2};
Tab  = T_r - dEdp*dp/dEdT;
```

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

The same kind of manipulation can be used to study alternative linearizations of systems of nonlinear equations and the influence of neglecting some of the derivatives when forming Jacobians.

To illustrate how the interplay between the three processes can change significantly and lead to quite different temperature behavior, we compare the predicted evolution of the temperature field for $\alpha = 5 \times 10^{-n}$, $n = 3, 4$, as shown in Figures 7.7 and 7.8. The change in behavior between the two figures is associated with the change in sign of $\partial E/\partial p$,

$$\mathrm{d}E = \left(c_p - \frac{\alpha T}{\rho}\right)\mathrm{d}T + \left(\frac{\beta_T p - \alpha T}{\rho}\right)\mathrm{d}p, \qquad \beta_T = \frac{1}{\rho}\frac{\partial \rho}{\partial p}\Big|_T. \tag{7.29}$$

In the isothermal case and for $\alpha = 5 \times 10^{-4}$, we have that $\alpha T < \beta_T p$ so that $\partial E/\partial p > 0$. The expansion and flow of fluid will cause an instant heating near the well-bore, which is what we see in the initial temperature increase for the maximum value in Figure 7.7. The Joule–Thomson coefficient $(\alpha T - 1)/(c_p \rho)$ is also negative, which means that the fluid gets heated if it flows from high pressure to low pressure in a steady-state flow. This is seen by observing the temperature in the well perforations. The fast pressure drop in these cells causes an almost instant cooling effect, but soon after we see a transition in which most of the cells containing a well perforation start having the highest temperature in the reservoir because of heating from the moving fluids. For $\alpha = 5 \times 10^{-3}$, we have that $\alpha T > \beta_T p$ so that $\partial E/\partial p < 0$ and likewise the Joule–Thomson coefficient is positive. The moving fluids will induce a cooling effect and hence the minimum temperature is observed at the well for a longer time. The weak kink in the minimum temperature curve is the result of the point of minimum temperature moving from being at the bottom front side to the far back of the reservoir. The cell with lowest temperature is where the fluid has done most work, neglecting heat conduction. In the beginning, this is the cell near the well because the pressure drop is largest there. Later, it will be the cell furthest from the well, since this is where the fluid can expand most. The discussion in this subsection is only meant to illustrate physical effect and does not necessarily represent realistic wells.

### *Computational Performance*

If you are observant, you may have realized that the code presented in this chapter contains a number of redundant function evaluations that may potentially add significantly to the overall computational cost: in each nonlinear iteration we keep reevaluating quantities that depend on `p0` and `T0` even though these stay constant for each time step. We can easily avoid this by moving the definition of the anonymous functions evaluating the residual equations inside the outer time loop. The main contribution to potential computational overhead, however, comes from repeated evaluations of fluid viscosity and density. Because each residual equation is defined as an anonymous function, `v(p,T)` appears three times for each residual evaluation, once in `pEq` and twice in `hEq`. This, in turn, translates to three calls to `mu(avg(p),avg(T))` and *seven* calls to `rho(p,T)`, and so on. In practice, the number of actual function evaluations is smaller, since the MATLAB interpreter most likely has some kind of built-in intelligence to spot and reduce redundant function evaluations.

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

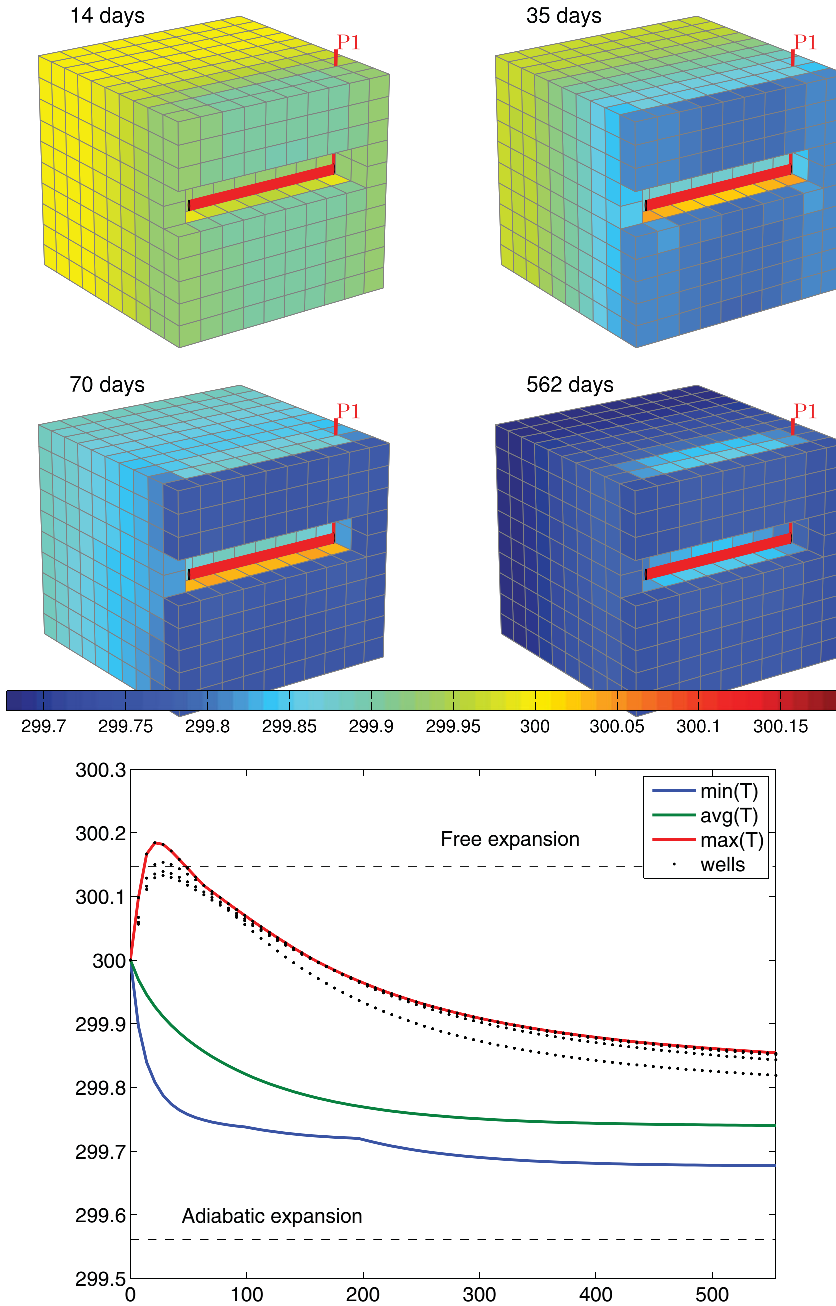



Figure 7.7 Time evolution of temperature for a compressible, single-phase problem with $\alpha = 5 \cdot 10^{-4}$. The upper plots show four snapshots of the temperature field. The lower plot shows minimum, average, maximum, and well-perforation values.

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

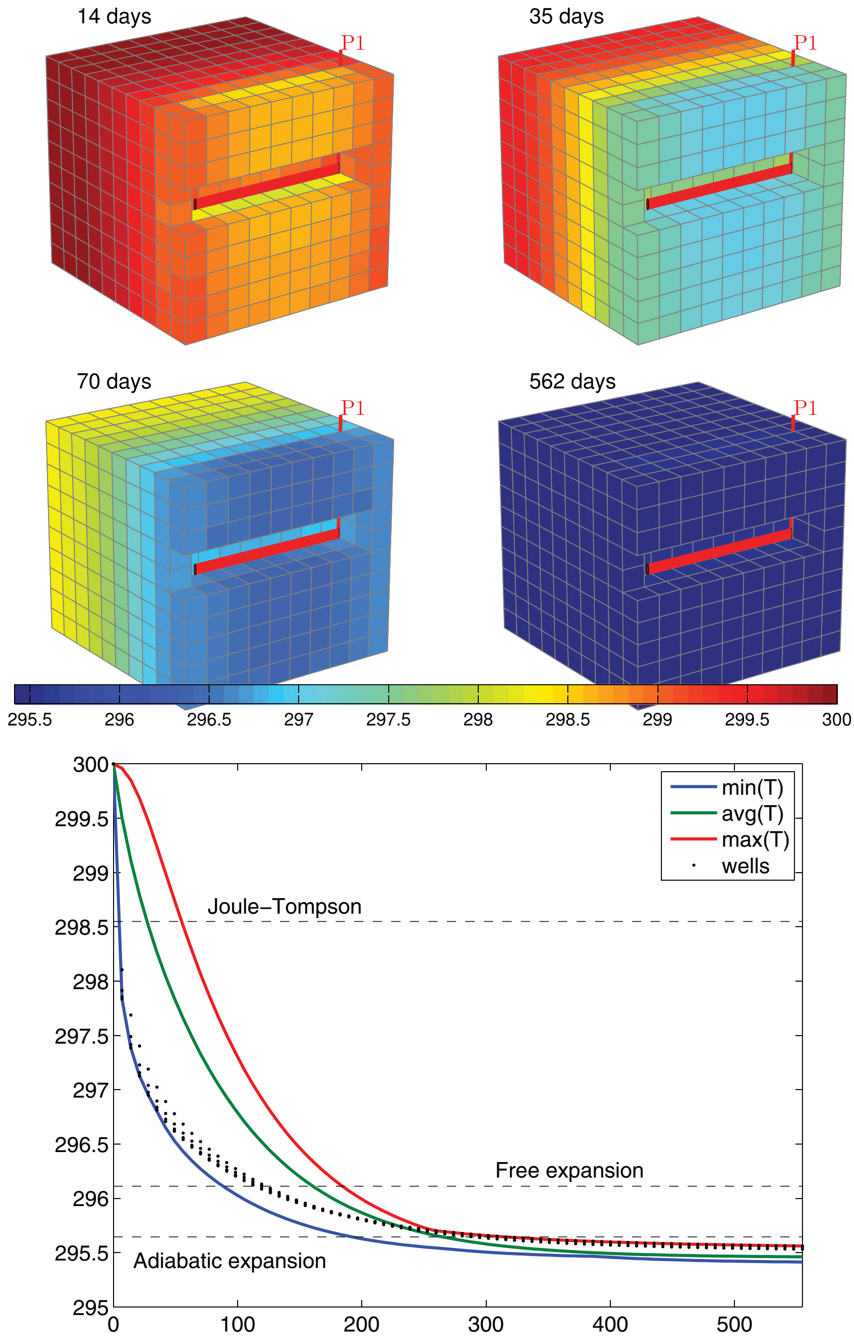



Figure 7.8 Time evolution of temperature for a compressible, single-phase problem with $\alpha = 5 \times 10^{-3}$. The upper plots show four snapshots of the temperature field. The lower plot shows minimum, average, maximum, and well-perforation values.

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press

Nonetheless, to cure this problem, we can move the computations of residuals inside a function so that the constitutive relationships can be computed one by one and stored in temporary variables. The disadvantage is that we increase the complexity of the code and move one step away from the mathematical formulas describing the method. This type of optimization should therefore only be introduced after the code has been profiled and redundant function evaluations have proved to have a significant computational cost.

## Computer exercises

7.5.1 Perform a more systematic investigation of how changes in $\alpha$ affect the temperature and pressure behavior. To this end, you should change $\alpha$ systematically, e.g., from 0 to $10^{-2}$. What is the effect of changing $\beta$, the parameters $c_\mu$ and $c_T$ for the viscosity, or $c_p$ in the definition of enthalpy?

7.5.2 Use the MATLAB profiling tool to investigate to what extent the use of nested anonymous functions causes redundant function evaluations or introduces other types of computational overhead. Hint: to profile the CPU usage, you can use the following call sequence

```
profile on, singlePhaseThermal; profile off; profile report
```

Try to modify the code as suggested above to reduce the CPU time. How low can you get the ratio between the cost of constructing the linearized system and the cost of solving it?

https://doi.org/10.1017/9781108591416.010 Published online by Cambridge University Press