Chapter 7

Coupled Problems

This chapter deals with two specific examples on systems of PDEs, concerning fluid-structure interaction and coupled heat and fluid flow. The exposition includes derivation of the PDEs, precise description of the numerical solution algorithms, and software design principles based on object-oriented programming and Diffpack tools. Contrary to Chapters 5 and 6, where *vector* PDEs were in main focus, we now address systems of PDEs where the different equations reflect different fundamental physical principles. Each scalar equation in the PDE system then has a life on its own. For example, the system treated in Chapter 7.2 consists of a momentum equation and an energy equation. From an implementational point of view, it would be advantagous to realize the compound solver for the system of PDEs as a simple combination of well-tested stand-alone solvers for the various scalar PDEs in the system. We shall pursue this idea in the present chapter.

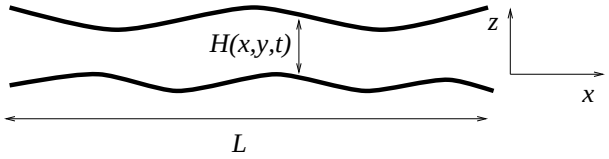
7.1 Fluid-Structure Interaction; Squeeze-Film Damping

Fluid-structure interaction is a basic problem in many engineering disciplines. The flow of a fluid around a structure gives rise to forces and associated motion of the structure, which again influences the fluid flow. The motion of the fluid is often described by the incompressible Navier-Stokes equations, whereas the structural deformations might be governed by the equations of linear elasticity. We could therefore, at least in principle, solve a class of fluid-structure interaction problems by combining solvers from Chapters 5 and 6. However, the implementational details of such a coupling of classes are better explained in a simplified problem. Of this reason, we address a fluid-structure problem where certain simplifications can be made, such that we end up with a PDE for the fluid flow and an ODE for the motion of the structure. This reduces the mathematical and numerical complexity and helps to expose the details of the software design. Furthermore, the simplifications demonstrate important mathematical modeling techniques for deriving PDEs.

7.1.1 The Physical and Mathematical Model

Sensors and actuators frequently contain small vibrating plates whose motion can be considerably influenced by induced viscous air flow in the surround-

ings. We shall focus on the situation where there are two vibrating plates separated by a narrow gap.



If the characteristic length of the plates is L and $H(x, y, t)$ is the width of the gap, we make the assumption that $L \gg H$ and that H is slowly varying. This particular physical problem is often referred to as *squeeze films.* The viscous fluid motion in the film can have a significant damping effect on the vibrations of the plates. In the design of sensors subject to large accelerations, e.g., during impact, one can rely on squeeze film damping to avoid destructive displacement of plate-like structures in the sensor system.

The equations governing squeeze-film damping will now be derived. Readers whose primary interest is the special software implementation technique being used for this simulator, can safely move on to Chapter 7.1.2.

The mathematical model for the fluid motion consists of the well-known incompressible or compressible Navier-Stokes equations. Depending on the constitutive law of the fluid, an energy equation might also be required. The fluid equations are defined in a domain whose shape is coupled to the movement of the plates. This movement can be modeled by standard equations for vibrating plates, with the fluid pressure as a driving force.

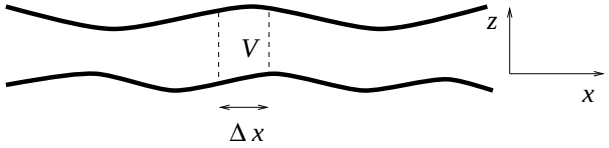
Derivation of the Fluid Flow Model. Physical problems involving domains where one dimension is much smaller than the others, can be effectively modeled by introducing quantities that are averaged over the thickness of the small dimension. In the present problem, we can introduce averaged velocities and pressure in the z direction. To derive the resulting equations, it is convenient to work with the fundamental balance laws on integral form. The general equation of continuity on integral form for an arbitrary control volume V reads

$$\int_V \frac{\partial \varrho}{\partial t} d\Omega + \int_{\partial V} \varrho \boldsymbol{v} \cdot \boldsymbol{n} d\Gamma = 0,$$

where ϱ is the fluid density, $\boldsymbol{v} = (u, v, w)^T$ is the velocity field, $\boldsymbol{n}$ is the outward unit normal vector to the surface ∂V of V, and t denotes time. For the purpose of deriving an averaged differential form of the continuity equation, we let

$$V = \{(x, y, z) \mid x \in [x_0, x_0 + \Delta x],\ y \in [y_0, y_0 + \Delta y],\\ z \in [h_B(x, y, t), h_T(x, y, t)]\},$$

The position of the lower plate is $z = h_B$, while $z = h_T$ is the equation for the upper plate. The gap is hence $H(x, y, t) = h_T(x, y, t) - h_B(x, y, t)$.



Evaluating the integrals for this particular choice of V gives

$$\Delta x \Delta y \int_{h_B}^{h_T} \frac{\partial \varrho}{\partial t} dz + \Delta y (U|_{x_0+\Delta x} - U|_{x_0})$$

$$+\Delta x (V|_{y_0+\Delta y} - V|_{y_0}) + \varrho \Delta x \Delta y \left(\frac{\partial h_T}{\partial t} - \frac{\partial h_B}{\partial t} \right) = 0, \tag{7.1}$$

Here,

$$U = \int_{h_B}^{h_T} \varrho u dz, \quad V = \int_{h_B}^{h_T} \varrho v dz$$

are the averaged horizontal mass fluxes. Moreover, we have used the boundary conditions on $z = h_B$ and $z = h_T$: $\boldsymbol{v} \cdot \boldsymbol{n} = \boldsymbol{v}_s \cdot \boldsymbol{n}$, where $\boldsymbol{v}_s$ is the velocity of the surface. The latter quantity equals $(\partial h_T/\partial t)\boldsymbol{k}$ for the surface $z = h_T$, and $(\partial h_B/\partial t)\boldsymbol{k}$ for the lower surface.

In the limit $\Delta x, \Delta y \to 0$, we obtain an equation of continuity that is averaged over the gap:

$$H \frac{\partial \varrho}{\partial t} + \frac{\partial U}{\partial x} + \frac{\partial V}{\partial y} + \varrho \frac{\partial H}{\partial t} = 0 \,. \tag{7.2}$$

Here we have assumed that $\partial \varrho/\partial t$ is independent of z. For an incompressible fluid, we take ϱ = constant, whereas in the compressible case we simply assume that ϱ is constant[1] in z direction.

The U and V quantities couple to the equation of motion. If $H \ll L$ and the variations of H are small (more precisely, the typical wave length of the deformation of the plates is much larger than H), one can assume that the flow is locally similar to steady rectilinear flow between two flat plates. With this assumption we can apply the expressions for U and V found from the equation of fluid motion in a channel of width H (Poiseuille flow [67]):

$$U = -\varrho \frac{H^3}{12\mu} \frac{\partial p}{\partial x}, \quad V = -\varrho \frac{H^3}{12\mu} \frac{\partial p}{\partial y}, \tag{7.3}$$

where μ is the fluid viscosity. Equation (7.3) is now our approximate form of the momentum equation in the squeeze-film problem. Note that the expressions for the mass fluxes stem from a simplified version of the *incompressible*

[1] This can be justified for barotropic fluids, since ϱ is then directly related to the pressure, which is expected to be approximately constant in z direction.

Navier-Stokes equations. We will, however, employ the same approximation in the compressible case as well.

Inserting U and V in the equation of continuity results in

$$H\frac{\partial \varrho}{\partial t} + \varrho\frac{\partial H}{\partial t} = \nabla\cdot\left[\frac{H^3}{12\mu}\varrho\nabla p\right]. \tag{7.4}$$

For incompressible flow this equation reduces to

$$\nabla\cdot\left[\frac{H^3}{12\mu}\nabla p\right] = \frac{\partial H}{\partial t}. \tag{7.5}$$

We employ the barotropic model $p/\varrho^\gamma = \text{const}$ in the case of compressible flow. From a numerical point of view, the equation for the fluid motion is easier to solve if we introduce $\tilde{u} = p^{1/\gamma}$. This results in

$$\frac{\partial}{\partial t}\left(H\tilde{u}\right) = \nabla\cdot\left[\frac{H^3}{12\mu}\gamma\tilde{u}^\gamma\nabla\tilde{u}\right]. \tag{7.6}$$

The fluid flow equations are to be solved in a domain Ω, covering the extent of the smallest plate. At the boundary $\partial\Omega$ of Ω, the pressure must match the atmospheric pressure, denoted here by p_0. The initial state is taken as $p = p_0$.

Exercise 7.1. Derive (7.1) and (7.3) in detail. ◇

Motion of the Plates. For the motion of the plates we can apply the standard equation for small deflection of thin plates. The total pressure $p(x,y,t) - p_0$ and the force $\varrho_S f(t)$ in an accelerated coordinate system comprise the loads on the plate. Here, $f(t)$ is the external acceleration of the coordinate system, and ϱ_S is the density per unit area of the plate. The governing equation for a vibrating plate can then be written as [59]

$$\varrho_S\frac{\partial^2 d}{\partial t^2} + D\nabla^4 d = p - p_0 + \varrho_S f, \quad D = \frac{Eq^3}{12(1-\nu^2)}. \tag{7.7}$$

The function $d(x,y,t)$ is the deflection of the plate, $\nabla^4 \equiv \nabla^2\nabla^2$ is the biharmonic operator, ν is Poisson's ratio, E is Young's modulus, and q is the thickness of the plate. We shall be particularly concerned with impulsively started vibrations from rest, modeled as rapid variation of $f(t)$:

$$f(t) = \begin{cases} I\sin^2\omega t, & t < \pi/\omega \\ 0, & t \geq \pi/\omega \end{cases} \tag{7.8}$$

The boundary conditions depend on the support of the plate. For example, on a clamped part of the plate, $d = \partial d/\partial n = 0$.

A finite element method for (7.7) requires quite complicated elements with twice differentiable basis functions, or we need to rewrite the plate equation

as a system of two PDEs, involving ∇^2 operators instead of ∇^4. A much simpler approach is to apply a spectral method for the spatial discretization of (7.7). This reduces the PDE to an initial-value problem involving only ordinary differential equations (ODEs). The disadvantage is that the spectral method outlined below is restricted to plates of rectangular or circular shape. However, such shapes are highly relevant in squeeze-film applications.

Assuming that the plate is rectangular and simply supported at $x = 0, L$ ($d = 0$), with the two other ends free ($\partial^2 d/\partial n^2 = 0$), a possible spatial expansion of d can be written as

$$d(x, y, t) \approx \hat{d}(x, t) = \sum_{j=1}^{M} a_j(t) \sin\left(j\pi \frac{x}{L}\right), \tag{7.9}$$

where $a_j(t)$ are amplitude functions to be computed. By this particular expansion we also assume that the fluid pressure does not vary with y, otherwise d would also depend on y. Since we know that damping effects may be significant in the squeeze-film problem, high-frequency basis functions are expected to have very small amplitude. We therefore attempt a one-term expansion,

$$\hat{d}(x, t) = a(t) \sin \pi \frac{x}{L}. \tag{7.10}$$

The equation for $a(t)$ can be derived from a Galerkin method applied to the vibrating plate equation with $\sin \pi x/L$ as weighting function. The result becomes[2]

$$\varrho_S \ddot{a} + \left(\frac{\pi}{L}\right)^4 Da = \frac{4}{\pi} \varrho_S f + \frac{2}{L} \int_0^L (p - p_0) \sin\left(\pi \frac{x}{L}\right) dx. \tag{7.11}$$

Exercise 7.2. Approximate the plate displacement d by a sum of M sinusoidal basis functions as in (7.9) and derive a decoupled system of ODEs for $a_j(t)$, $j = 1, \ldots, M$. Explain why the system becomes decoupled. ◇

Exercise 7.3. Suggest a generalization of (7.9) in the case of a rectangular plate and a two-dimensional pressure field, $p = p(x, y, t)$. Assume that all sides of the plate are simply supported. Compute the equation for the time-dependent coefficient in a one-term expansion. ◇

Summary of the Mathematical Model. For a sample application in micromechanical sensor technology, the lower plate is often stiff enough to remain plane ($h_B = \text{constant}$). The gap between the plates in the nondeformed state is h_0, and we place the z axis such that $z = h_B = 0$. The relation between H and d then becomes $H(x, y, t) = h_0 + d(x, y, t)$. Since h_0 is expected to be small, the coefficient H^3 can be very small and cause numerical problems.

[2] Useful formulas: $\int_0^L \sin(x\pi/L)dx = 2L/\pi$, $\int_0^L \sin^2(x\pi/L)dx = L/2$. Later we will also make use of $\int_0^L \sin^3(x\pi/L)dx = 4L/(3\pi)$.

A proper scaling of the equations would cure such problems. However, scaling of the present initial-boundary value problem quickly becomes a tedious procedure. We therefore apply the simpler approach of multiplying the flow equation by the factor $12\mu h_0^{-3}$, such that we avoid very small values in the coefficient in the Laplace term. Another problem is that $p_0 = 0$ implies $p = 0$ at all times. It is therefore advantageous to have the primary unknown u as a perturbation around unity. This is accomplished by introducing $u = (p/p_0)^{1/\gamma}$ as primary unknown.

The initial-boundary value problem for compressible flow can now be summarized.

$$\varrho_S \ddot{a} + \left(\frac{\pi}{L}\right)^4 Da = \frac{4}{\pi}\varrho_S f + \frac{2}{L}\int_0^L (p(\boldsymbol{x},t) - p_0)\sin\left(\pi\frac{x}{L}\right) dx, \quad (7.12)$$

$$a(0) = \dot{a}(0) = 0, \quad (7.13)$$

$$12\mu h_0^{-3}\frac{\partial}{\partial t}(Hu) = \nabla\cdot\left[\left(\frac{H}{h_0}\right)^3 p_0\gamma u^\gamma \nabla u\right], \quad \boldsymbol{x}\in\Omega,\ t>0, \quad (7.14)$$

$$u = \frac{p^{1/\gamma}}{p_0^{1/\gamma}}, \quad (7.15)$$

$$H = h_0 + a(t)\sin\left(\pi\frac{x}{L}\right), \quad (7.16)$$

$$u(\boldsymbol{x},t) = 1, \quad \boldsymbol{x}\in\partial\Omega_E, \quad (7.17)$$

$$u(\boldsymbol{x},0) = 1, \quad \boldsymbol{x}\in\Omega. \quad (7.18)$$

The domain Ω is either one- or two-dimensional, and $\partial\Omega_E$ denotes the complete boundary of Ω.

For incompressible flow, we simply replace (7.14) by

$$\nabla\cdot\left[\left(\frac{H}{h_0}\right)^3 p_0\nabla u\right] = 12\mu h_0^{-3}\frac{\partial H}{\partial t}, \quad \boldsymbol{x}\in\Omega,\ t>0. \quad (7.19)$$

In this case, $u = p/p_0$. To handle both the compressible and incompressible case within the numerical expressions and the same code lines, it is convenient to introduce a variable coefficient $k(u)$ in the Laplace term, where $k(u) = \gamma u^\gamma$ in compressible flow and $k(u) = 1$ when the flow is incompressible.

Analysis of a Simple Case. Valuable insight into the problem can be obtained by analyzing a special case where an analytical solution is straightforwardly derived. This is also fundamental for partial verification of a computer implementation. With $H(x,t) = h_0 + a(t)\sin(\pi x/L)$, the boundary-value problem of *an incompressible* fluid, in the one-dimensional case, becomes

$$\frac{\partial}{\partial x}\left(\left(1+\frac{a}{h_0}\sin\pi\frac{x}{L}\right)^3\frac{\partial p}{\partial x}\right) = 12\mu h_0^{-3}\dot{a}\sin\pi\frac{x}{L}, \quad p(0) = p(L) = p_0. \quad (7.20)$$

We can integrate (7.20) and make a first-order approximation to the resulting right-hand side:

$$\frac{\partial p}{\partial x} = 12\mu\dot{a}h_0^{-3}\left(\frac{L}{\pi}\cos\pi\frac{x}{L} + C_1\right) .$$

Here C_1 is an integration constant. Integrating once more and inserting the boundary values yields

$$p(x,t) = p_0 - \dot{a}\frac{12\mu L^2}{h_0^3\pi^2}\sin\pi\frac{x}{L} . \tag{7.21}$$

In the case of a *flat plate*, $d = a(t)$, we would get

$$p = p_0 - \dot{a}\frac{12\mu}{h_0^3}\frac{1}{2}x(L-x),$$

which leads to the same qualitative behavior of $p-p_0$. The maximum pressure disturbance is, however, affected by a factor of $\pi^2/8 \approx 1.23$.

The $p - p_0$ function can now be inserted in our simplified vibrating plate equation. The result becomes

$$\varrho_S\ddot{a} + \kappa\dot{a} + \left(\frac{\pi}{L}\right)^4 Da = \frac{4}{\pi}\varrho_S f, \quad \kappa = \frac{12\mu L^2}{h_0^3\pi^2} . \tag{7.22}$$

Observe that the "driving force" $\partial H/\partial t$ in the fluid flow equation leads to $p \sim \dot{a}$, which then gives rise to a *damping* term $\kappa\dot{a}$ in the vibrating plate equation. This means that for incompressible fluids, the squeeze film will always damp the structural vibration.

Exercise 7.4. Consider compressible flow with a prescribed $d = d_0\sin\omega t$. Scale the flow equation, using ω^{-1} as time scale, and show that only one dimensionless number, $\sigma = 12\mu\omega L^2/(p_0 h_0^2\gamma)$, appears in the scaled equation. One often refers to σ as the *squeeze film number*. ◇

7.1.2 Numerical Methods

Our mathematical model for the squeeze-film problem consists of a linear or nonlinear heat-conduction-like PDE, i.e. (7.19) or (7.14), coupled with a linear second-order ODE (7.12). The simplest solution strategy is to solve the equations in sequence. At each time level, we first solve for the fluid motion and compute the pressure load on the plate. Thereafter, we carry out one time step in the discrete plate equation.

The fluid flow equation (7.19) or (7.14) can be discretized by a Galerkin finite element method and a θ-rule in time (see Chapter 2.2.2). This yields a system of linear or nonlinear algebraic equations to be solved at each time level. In the nonlinear case, the system can be solved by Newton-Raphson or Successive Substitution (Picard iteration) techniques. We refer to Chapter 4

for details regarding discretization and implementation of a PDE like (7.14). The coefficient matrix $J_{i,j}$ and the right-hand side vector $-F_i$ of the linear system to be solved in each Newton-Raphson iteration take the following form:

$$J_{i,j} = \int_\Omega \Big[\tau 12\mu h_0^{-3} H N_i N_j + \theta \left(\frac{H}{h_0}\right)^3 \Delta t \left(k(u)\nabla N_i \cdot \nabla N_j + k'(u)\nabla N_i \cdot \nabla u N_j\right)\Big] d\Omega, \quad (7.23)$$

$$-F_i = -\int_\Omega \Big[12\mu h_0^{-3} N_i \Delta\Psi + \left(\frac{H}{h_0}\right)^3 \Delta t \left(\theta k(u)\nabla N_i \cdot \nabla u + (1-\theta)k(\bar{u})\nabla N_i \cdot \nabla \bar{u}\right)\Big] d\Omega\,. \quad (7.24)$$

The current time step size is denoted as Δt. Quantities with a bar denotes evaluation at the previous time level. For example, if Δt is constant, u is a short notation for the numerical approximation to $u(\boldsymbol{x}, \ell\Delta t)$, while $\bar{u}$ is the corresponding notation for the approximation to $u(\boldsymbol{x}, (\ell-1)\Delta t)$. To simplify the notation, we have dropped the iteration number as superscript, that is, the symbol u in the numerical formulas refers to the most recent approximation to the primary unknown function u. The parameter τ equals unity in compressible flow and vanishes for incompressible flow. Moreover,

$$\Delta\Psi = \begin{cases} Hu - \bar{H}\bar{u}, \text{compressible flow} \\ H - \bar{H}, \quad \text{incompressible flow} \end{cases}$$

In the Successive Substitution method the linear system in each iteration has a coefficient matrix

$$A_{i,j} = \int_\Omega \left(\tau 12\mu h_0^{-3} H N_i N_j + \theta \left(\frac{H}{h_0}\right)^3 \Delta t k(u)\nabla N_i \cdot \nabla N_j\right) d\Omega \quad (7.25)$$

and a right-hand side vector

$$b_i = \int_\Omega \left(12\mu h_0^{-3} N_i \Phi - (1-\theta)\left(\frac{H}{h_0}\right)^3 \Delta t k(\bar{u})\nabla N_i \cdot \nabla \bar{u}\right) d\Omega\,. \quad (7.26)$$

Here, $\Phi = \bar{H}\bar{u}$ in compressible flow and $\Phi = \bar{H} - H$ in incompressible flow.

Exercise 7.5. Derive the expressions (7.23)–(7.26). ◇

The second-order ODE for the plate motion can be compactly written as

$$c_1\ddot{a} + c_2\dot{a} + c_3 a = c_4, \quad (7.27)$$

with suitable definitions of c_1, c_2, c_3, and c_4 according to (7.12). A widely used *Newmark scheme* for (7.27) can be formulated as [128, Ch. 10, Vol II]

$$\ddot{a} = \left(c_1 + \beta_1 c_2 \Delta t + c_3 \beta_2 \frac{\Delta t^2}{2}\right)^{-1} \times \left(c_4 - c_2\left(\bar{\dot{a}} + (1-\beta_1)\Delta t \bar{\ddot{a}}\right)\right) - c_3(\bar{a} + \Delta t \bar{\dot{a}} + (1-\beta_2)\frac{\Delta t^2}{2}\bar{\ddot{a}}\Big) \quad (7.28)$$

$$a = \bar{a} + \Delta t \bar{\dot{a}} + (1-\beta_2)\frac{\Delta t^2}{2}\bar{\ddot{a}} + \beta_2 \frac{\Delta t^2}{2}\ddot{a} \quad (7.29)$$

$$\dot{a} = \bar{\dot{a}} + (1-\beta_1)\Delta t \bar{\ddot{a}} + \beta_1 \Delta t \ddot{a} \quad (7.30)$$

Again, the bar indicates quantities at the previous time level. Initially, $\bar{a}$ and $\bar{\dot{a}}$ are prescribed. The value of $\bar{\ddot{a}}$ for $t = 0$ follows from (7.27).

Various choices of the parameters correspond to different well-known schemes. For example, $\beta_1 = \beta_2 = 1/2$ results in a Crank-Nicolson-like scheme. In the present scalar case, we obtain an explicit formula for the new a value, although the finite difference scheme is implicit (an implicit *linear* equation in one variable can always be converted to an explicit form).

7.1.3 Implementation

The coupled fluid-structure problem modeling squeeze films is rather complicated in its original form. Through some reasonable assumptions the model has been reduced to a coupled system of a generally nonlinear scalar PDE (7.14) and a linear ODE (7.12).

For simplicity, we decide to solve the PDE and the ODE in sequence at each time level. The implementation of coupled models can quickly become an error-prone process. To ease the coding and the associated debugging, and at the same time increase the extensibility of the mathematical model, we propose to make separate solvers for the fluid PDE and the plate ODE. The equations can then be tested separately before we couple the two classes in a compound solver for the fluid-structure interaction problem. At any time in the debugging of the compound solver, the component simulators can be pulled apart again and retested separately. This is an attractive implementational approach, but we need to clarify the details of the software design. It is advantageous to have studied Chapter 3.5.6 before proceeding. The complete source code of the squeeze film solver is located in the directory tree `src/app/SqueezeFilm`.

The PDE and ODE are coupled through the H and p quantities. Therefore, if we want to solve the fluid PDE on its own, H must be a prescribed quantity. Similarly, when solving the plate ODE separately from the fluid equation, the p field must be specified.

Assume that we make a class `PlateVib1` that solves the general version of (7.27). For testing purposes, we derive a subclass `PlateSin` where we specialize

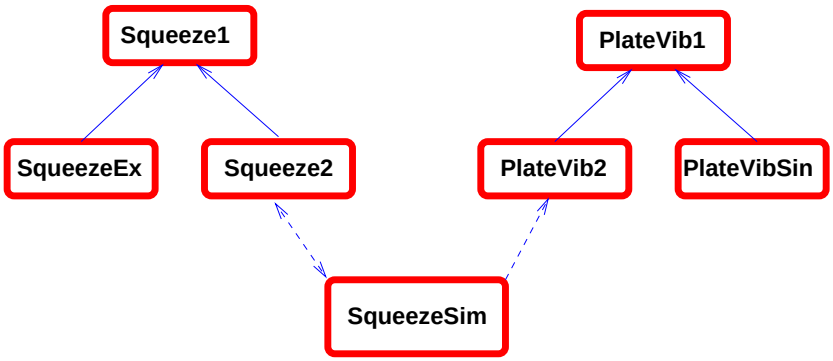


Fig. 7.1. Sketch of the squeeze film solver `SqueezeSim`, which consists of a fluid flow solver `Squeeze2` and a vibrating plate solver `PlateVib2`. The classes `SqueezeEx` and `PlateVibSin` are aimed at simplified test problems for the fluid flow and the structural vibration, respectively. Solid lines indicate inheritance ("is-a" relationship), while dashed lines indicate pointers ("has-a" relationship).

$f(t)$ as $f(t) = q\sin^2\omega t$, set $p - p_0 = 0$, and compare the numerical and analytical solution. In another subclass `PlateVib2` we implement the more application-relevant shock pulse (7.8) for $f(t)$. This version of the vibrating plate solver is to be coupled to the fluid flow equation. As usual, the code in subclass solvers is only a few lines.

The fluid PDE is implemented as a standard Diffpack finite element solver, like class `NlHeat1` from Chapter 4.2. The name of the class is `Squeeze1`, and it can handle both compressible and incompressible flow in one or two space dimensions. We leave the choice of H as a virtual function `hmodel`. In a subclass `SqueezeEx`, `hmodel` is implemented according to $H = h_0 + t$. The solution, assuming incompressible flow, is then $p = p_0 + 6x(x-1)/(h_0+t)^3$. We can hence use class `SqueezeEx` for partial verification of the implementation of the fluid flow solver.

Another subclass `Squeeze2` of `Squeeze1` implements the `hmodel` function with access to the plate deflection field: $H = h_0 + d$. To couple the `Squeeze2` and `PlateVib2` solvers, one can introduce a manager class `SqueezeSim` that holds a fluid flow solver `Squeeze2` and a vibrating plate solver `PlateVib2`. Figure 7.1 depicts the class relations in the squeeze film solver. As advocated in Chapter 3.5.6, the source code of the various subclasses in a solver hierarchy can conveniently be stored in different subdirectories. With the `AddMakeSrc` script one can tell the make program that the source is distributed across directories. The squeeze-film solver has a directory `coupling` for the compound simulator, `vib-test` for verifying the vibrating plate solver, and `film-test` for verifying the fluid flow simulator as a stand-alone solver.

For a more detailed explanation of the software design of the squeeze film simulator, it is convenient to start with the manager, class `SqueezeSim`:

```
class SqueezeSim : public SimCase
{
public:
  Handle(PlateVib2)  plate;
  Handle(Squeeze2)   film;

  Handle(GridFE)     grid;
  Handle(TimePrm)    tip;
  Handle(SaveSimRes) database;

  virtual void define (MenuSystem& menu, int level = MAIN);
  virtual void scan   ();
  virtual void timeLoop ();
  real computePressureLoad (FieldFE& pressure, real p0);
  SqueezeSim ();
 ~SqueezeSim () {}

  virtual void adm (MenuSystem& menu);
  virtual void solveProblem ();
  virtual void resultReport ();
};
```

The manager `SqueezeSim` is in charge of the grid, the time integration parameters, and the storage for later visualization. Other tasks are distributed to the fluid flow and vibrating plate solvers. The `define` function exemplifies how some menu items are specific to `SqueezeSim` and how others are completely handled by calling the `define` functions in the stand-alone solvers for each equation.

```
void SqueezeSim:: define (MenuSystem& menu, int level)
{
  menu.addItem (level, "gridfile", "readOrMakeGrid syntax",
                "P=PreproBox | d=1 [0,10] | d=1 elm_tp=ElmB2n1D "
                "div=[20], grading=[1]");
  menu.addItem (level, "time integration parameters",
                "TimePrm::scan(Is) syntax", "dt=0.1  t in [0,1]");
  SaveSimRes::defineStatic (menu, level+1);

  menu.setCommandPrefix ("plate");
  plate->define (menu, level, true);
  menu.setCommandPrefix ("film");
  film ->define (menu, level, true);
  menu.unsetCommandPrefix ();
}
```

The fluid flow and the vibrating plate simulators could in general happen to define menu items with the same name. This is the case if both of the solvers put a linear system and solver interface object (`LinEqAdmFE`) on the menu, a situation that occurs when coupling two or more finite element solvers (cf. the simulator in Chapter 7.2). However, the menu system offers the possibility to set a *command prefix* for all the proceeding command names. As we see from the `define` function above, one sets the command prefix "plate" before calling `plate->scan`. This means that the menu command `density` in the vibrating

plate solver actually gets the name `plate density`. Similarly, all the fluid flow menu items are prefixed by "film". If the fluid flow solver had defined a menu item `density` as well, the name of this item would be `film density`. During scanning of menu items, one can activate or deactivate the prefix feature of the menu system. For example, with the prefix "film", a command `menu.get("density")` will actually search for `film density`. The `scan` function can look like this:

```
void SqueezeSim:: scan ()
{
  MenuSystem& menu = SimCase::getMenuSystem();
  String gridfile = menu.get ("gridfile");
  grid.rebind (new GridFE());              // create empty grid object
  readOrMakeGrid (*grid, gridfile);        // fill grid
  tip.rebind (new TimePrm());
  tip->scan (menu.get ("time integration parameters"));
  database.rebind (new SaveSimRes());
  database->scan (menu, grid->getNoSpaceDim());

  menu.setCommandPrefix ("plate");
  plate->scan (menu, database->cplotfile, tip.getPtr());
  menu.setCommandPrefix ("film");
  film ->scan (grid.getPtr(), tip.getPtr(), database.getPtr());
  menu.unsetCommandPrefix ();
```

The `Squeeze1` solver must create its own grid in order to work as a stand-alone solver. However, when the class is used in conjunction with `SqueezeSim`, it is natural for the manager class to be responsible for the grid[3]. The `scan` function of a solver could then take a grid pointer as argument. If the pointer is null, the grid is allocated internally in class `Squeeze1`, otherwise the grid handle in `Squeeze1` is rebound to an external grid. The same strategy applies to the `TimePrm` and `SaveSimRes` objects, which are either internal in the fluid flow simulator or managed by class `SqueezeSim`.

```
void Squeeze1:: define (MenuSystem& menu, int level, bool externals)
{
  if (!externals)
    menu.addItem (level, "gridfile", "readOrMakeGrid syntax",
                  "P=PreproBox | d=1 [0,10] | d=1 elm_tp=ElmB2n1D "
                  "div=[20], grading=[1]");


void Squeeze1:: scan (GridFE* grid_, TimePrm* tip_,
                      SaveSimRes* database_)
{
  MenuSystem& menu = SimCase::getMenuSystem();
  if (grid_ != NULL)
    grid.rebind (grid_);  // bind to some external grid
  else {
```

[3] In the present case, only the fluid solver needs a grid, but in a more general problem setting, the manager class creates a common grid and distributes it to all the solvers.

```
    String gridfile = menu.get ("gridfile");
    grid.rebind (new GridFE());         // create empty grid object
    readOrMakeGrid (*grid, gridfile);   // fill grid
  }
```

With the use of handles, the origin of an object is of no interest; all solver classes can access the object through the handle, as if it were created by that class.

The heart of the `SqueezeSim` class is the `timeLoop` routine. This function demonstrates how the fluid flow and the vibrating plate solvers are supposed to work together. The basic numerical steps consist of advancing the fluid flow solver one time level to compute a new pressure field p. Then $p - p_0$ is integrated over the plate, using the function `SqueezeSim::computePressureLoad`. The resulting pressure load on the plate is transferred to the vibrating plate solver before asking that solver to update the d field at the new time level.

```
void SqueezeSim:: timeLoop ()
{
  tip->initTimeLoop();
  film ->timeLoopSetUp ();
  plate->timeLoopSetUp ();

  while(!tip->finished())
   {
     tip->increaseTime();
     film ->solveAtThisTimeStep ();
     plate->effectivePressureLoad
       (this->computePressureLoad (*film->p, film->p0));
     plate->solveAtThisTimeStep ();
     film ->saveResults ();
     plate->saveResults ();

     if (tip->getTimeStepNo() % 100 == 0)
       s_o << "t=" << tip->time() << endl;
   }
  plate->timeLoopFinish();
  film ->timeLoopFinish();
}
```

The fluid flow and vibrating plate solvers have split the traditional contents of a time loop function into `timeLoopSetUp` for storing/plotting the initial fields, `solveAtThisTimeStep` for solving the equations in a solver at a given time level, `saveResults` for calling `SaveSimRes` or related functionality for storing fields for later visualization, and `timeLoopFinish` for closing curve plots or post processing time series results. This high degree of modularity is not apparent for a single solver, but is very advantageous when combining stand-alone solvers into a simulator for a system of differential equations.

The basic fluid flow solver `Squeeze1` is very similar to, e.g., class `NlHeat1` so there is no need to explain the details here. The purpose of the `Squeeze2` subclass simulator is to implement the virtual function `hmodel` for computing H by calling the vibrating plate solver's function `computeDeflection`. This

latter function loads the deflection into a `FieldFE` object. The communication between the fluid flow solver and the vibrating plate solver is enabled by a two-way pointer between class `Squeeze2` and the manager, class `SqueezeSim`.

```
class Squeeze2 : public Squeeze1
{
public:
  SqueezeSim* manager;
  Squeeze2 (SqueezeSim* manager_) : manager(manager_) {}
  virtual void hmodel (FieldFE& H);
};
```

We can then simply compute $H = d + h_0$ like this:

```
void Squeeze2:: hmodel (FieldFE& H)
{
  manager->plate->computeDeflection (H);  // H  = d
  H.add (h0);                             // H += h0
}
```

Based on our brief review of the `SqueezeSim` solver, the reader is encouraged to study the source code in detail. Hopefully, one will realize that the classes `Squeeze1` and `PlateVib1` are similar to standard Diffpack simulators and that the extra glue for communication when solving the compound system is just very small subclasses. The ideas of coupling stand-alone simulators for solving a system of differential equations are developed further in Chapter 7.2.

Exercise 7.6. The squeeze-film simulator handles in principle 1D and 2D pressure fields. However, class `PlateVib1` does not support 2D functions for spectral discretization of the vibrating plate equation. Describe how class `PlateVib1` can be extended to handle rectangular plates (e.g. simply supported along all sides). ◇

Exercise 7.7. The efficiency of the simulator can be enhanced by introducing a spectral approximation for the pressure field p, similar to the representation of d. Formulate such a numerical method and discuss its impact on the design of the compound simulator `SqueezeSim`. Explain how we can choose between a finite element or a spectral solver at run time. ◇

Finally, we show a typical solution of the squeeze-film problem. Figure 7.2 displays the characteristic pressure and displacement response to an external acceleration pulse on the fluid-structure system. As expected, the vibration of the plate is damped due to the motion of the fluid film.

7.2 Fluid Flow and Heat Conduction in Pipes

This section addresses a coupled fluid-heat flow problem. We consider laminar flow of a non-Newtonian fluid in a straight pipe with a geometrically arbitrary

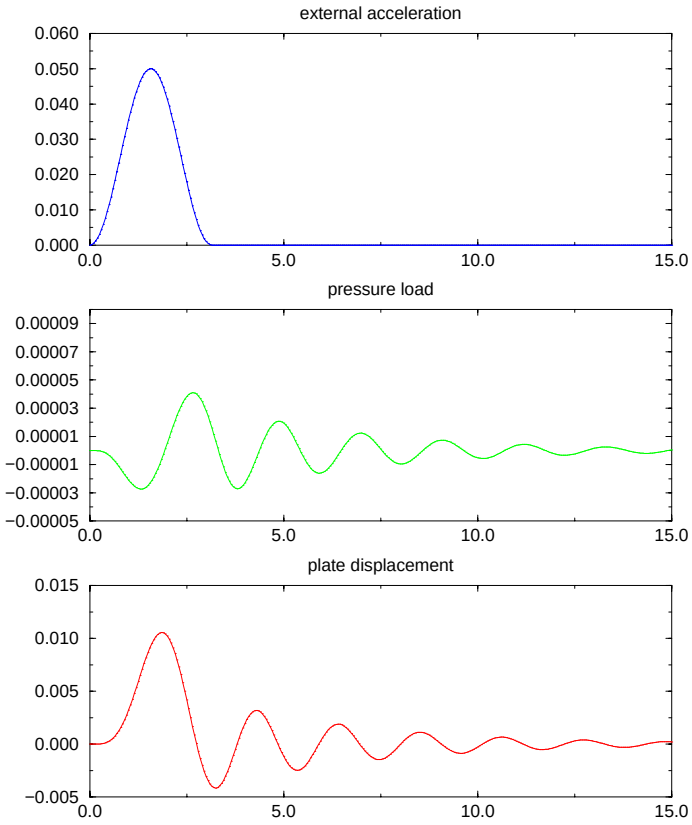


Fig. 7.2. Time series of the external acceleration $f(t)$ (top), the pressure load on the plate (middle), and the plate displacement $a(t)$ (bottom). The parameters were $E = 5000$, $\nu = 0.25$, $q = 0.01$, $L = 1$, $\beta_1 = \beta_2 = 0.5$, $\omega = 1$, $I = 0.05$, $\varrho_S = 0.5$, $\theta = 1$, $\gamma = 0$, $h_0 = 0.2$, $p_0 = 0$, and $\mu = 1.7 \cdot 10^{-5}$.

cross section. The viscosity of the fluid may depend on the temperature, and heat generation by internal friction in the fluid is an effect we shall include in the model. Possible applications of such a model cover extrusion of metal and flow of highly viscous polymers.

The present flow problem posed in a general 3D geometry is complicated, but the restriction of the flow region to a straight pipe makes it possible to introduce substantial simplifications. The resulting model consists of two coupled nonlinear Poisson equations. For some relevant values of the physical parameters the nonlinearities are severe, and many common methods, like Successive Substitution (Picard iteration) or Newton's method, face serious convergence problems. In the case of Newtonian flow, where the fluid properties are constant, we achieve two decoupled linear Poisson equations. This means that the present mathematical model is a nice test problem for numer-

ical solution of a system of PDEs where the coupling between the equations and the nonlinearities can be adjusted by varying the value of physical parameters in the problem. The main purpose of including this fluid-heat flow problem in the present book is for demonstrating how we can make a solver for the compound problem by combining independent solvers for the fluid and heat flow equations, respectively. This is a general technique that makes software development of simulators solving systems of PDEs faster and more reliable. It might be advantageous to have studied the basic software design principles in Chapter 7.1.3.

7.2.1 The Physical and Mathematical Model

The General Mathematical Model. Our derivation of the mathematical model for non-Newtonian pipeflow with thermal effects will make use of the indicial notation, including the summation convention and the comma notation for derivatives. Appendix A.2 explains this effective notation in detail. Readers who are not interested in the derivation of the model, can jump to the listing of the boundary-value problem (7.37)–(7.40) on page 474.

The Newtonian fluid model is characterized by a linear relation between internal forces in the fluid and the fluid motion. More precisely, there is a linear relation between the stress tensor σ_{rs} and the velocity gradients $v_{r,s}$:

$$\sigma_{rs} = -p\delta_{rs} + \mu(v_{r,s} + v_{s,r})\,. \tag{7.31}$$

Here, p is the pressure and δ_{rs} is the Kronecker delta. This is a constitutive law, which reflects physical properties of a particular fluid. The other equations that enter the complete mathematical model are of general type and apply to "all" fluids. For constant μ the presented equations reduce to the well-known Navier-Stokes equations. If the relation between the stress and the velocity gradients is nonlinear, the fluid is classified as non-Newtonian.

Non-Newtonian fluid flow is a huge topic and may involve complicated large-deformation viscoelastic and viscoplastic models. Nevertheless, many fluids that deviate from the common Newtonian model are satisfactorily described by a conceptually simple extension of (7.31). Instead of treating μ as a constant for the fluid in question, we allow the apparent viscosity to depend on the motion. Introducing a measure $\dot{\gamma}$ of the "intensity" of the deformation in the flow, $\dot{\gamma} = \sqrt{2\dot{\varepsilon}_{rs}\dot{\varepsilon}_{rs}}$, we simply postulate that μ is a function of $\dot{\gamma}$. This gives rise to the family of the so-called *generalized Newtonian* fluid models, which will attract our attention in the rest of this section.

A complete mathematical model for coupled fluid and heat flow involves a mass conservation equation, an equilibrium equation, a constitutive law like (7.31), and an energy equation. Many common non-Newtonian fluids are highly viscous and incompressible. The relevant mass conservation equation then becomes

$$v_{r,r} = 0\,. \tag{7.32}$$

The equilibrium equation for a continuum takes the form

$$\varrho(v_{r,t} + v_s v_{r,s}) = \sigma_{rs,s} + \varrho b_r, \tag{7.33}$$

where the left-hand side is the density ϱ times the acceleration and the right-hand side reflects the sum of forces on the continuum: stresses and body forces (b_r). A suitable form of the constitutive law for a generalized Newtonian fluid is

$$\sigma_{rs} = -p\delta_{rs} + \mu(v_{r,s} + v_{s,r}), \tag{7.34}$$

where μ depends on

$$\dot{\gamma} = \sqrt{\frac{1}{2}(v_{r,s} + v_{s,r})(v_{r,s} + v_{s,r})} \tag{7.35}$$

and possibly the temperature T. The latter quantity is governed by an energy equation, e.g.,

$$\varrho C_p (T_{,t} + v_k T_{,k}) = \kappa T_{,kk} + \mu\dot{\gamma}^2 . \tag{7.36}$$

Here, C_p is the heat capacity at constant pressure, and κ is the coefficient of thermal conduction. The left-hand side reflects the change of internal energy of a small fluid element, which is due to the right-hand side terms involving conduction $(\kappa T_{,kk} \equiv \kappa\nabla^2 T)$ and dissipation $(\mu\dot{\gamma}^2)$. The latter term represents internal heat generation from the work done by the stresses.

Combining the equations (7.32)–(7.36) yields a coupled system of a nonlinear convection-diffusion equation for T and a generalized Navier-Stokes equation for v_i and p. With a suitable Navier-Stokes solver, we could devise a finite element method for this system of PDEs, embedded in an iteration technique for handling the nonlinearities. Nevertheless, our purpose is to derive a simpler mathematical model by restricting the flow geometry to a straight pipe.

The Simplified Mathematical Model. The fundamental simplification of the coupled heat-fluid flow model is that we consider laminar flow in a *straight* pipe. Let the z axis be directed along the pipe. The velocity field is then expected to be $\boldsymbol{v} = w\boldsymbol{k}$, where $\boldsymbol{k}$ is a unit vector in z direction. The equation of continuity, $v_{r,r} = 0$, now immediately gives $\partial w/\partial z = 0$, which implies that $w = w(x, y, t)$. The special form of the velocity field leads to a dramatically simplified equation of motion for the fluid; the Navier-Stokes equations with a nonlinear viscosity model are reduced to a nonlinear scalar PDE $\varrho w_{,t} = \nabla\cdot[\mu\nabla w] + \text{const}$. If the problem is considered as stationary, as we do in the following, w is governed by a Poisson equation $\nabla\cdot[\mu\nabla w] = -\beta$, where β is the constant pressure gradient that drives the flow. Another basic assumption is that T does not vary along the pipe. The simplifications in the heat flow model due to rectilinear flow are not substantial; the convection-diffusion equation in the general case is only reduced to a Poisson equation $\nabla^2 T = f(T, w)$ in the present stationary problem. The equations for w and T are to be solved in a domain Ω which represents the *cross section of the pipe.*

The Boundary-Value Problem. The complete set of differential equations can be written as

$$\nabla \cdot [\mu \nabla w] = -\beta, \tag{7.37}$$

$$\nabla^2 T = -\kappa^{-1} \mu \dot{\gamma}^2, \tag{7.38}$$

$$\mu = \mu_T(T)\mu_w(\dot{\gamma}), \tag{7.39}$$

$$\dot{\gamma} = \sqrt{(w_{,x})^2 + (w_{,y})^2}. \tag{7.40}$$

The simplifications due to flow in a straight pipe also apply if we consider flow in a straight channel, both of Poiseuille type (driven by a pressure gradient β) or Couette type (driven by a moving channel wall). The PDEs are the same, but the number of effective space dimensions and the boundary conditions are different. By allowing some flexibility in setting the boundary conditions, the mathematical model and its computer implementation will be applicable to both pipe and channel flow.

Let $\partial\Omega_1$ be the part of the boundary corresponding to a fixed wall where $w = 0$, let $\partial\Omega_2$ be a wall moving with velocity W_1 in z direction, let $\partial\Omega_3$ be a possible symmetry plane where $\partial w/\partial n = 0$ and $\partial T/\partial n = 0$, let $\partial\Omega_4$ be a wall with fixed temperature $T = 0$, let $\partial\Omega_6$ be another wall with fixed temperature $T = T_1$, and finally let $\partial\Omega_7$ be a wall where a cooling condition

$$-\kappa \frac{\partial T}{\partial n} = h_T(T - T_s) \tag{7.41}$$

applies. Here, h_T is a coefficient that reflects the heat transfer through the channel wall to the pipe's surroundings, which have a constant temperature T_s. When assigning boundary conditions in a flow case, one must recall that we need exactly one condition on w and one condition on T at every point on the boundary.

Constitutive Laws. Some common models for μ_T and μ_w are listed next.

- The *Sisko* model:

$$\mu = \mu_\infty + \mu_0 \dot{\gamma}^{n-1}, \tag{7.42}$$

  where μ_∞ is the viscosity at very high shear rates, μ_0 is a reference viscosity[4], and n is the "power-law exponent", which is usually in the interval $[0.15, 0.6]$. When $\mu_\infty = 0$, this model reduces to the standard power-law model. The choice $n = 1$ and $\mu_\infty = 0$ leads to Newtonian flow with viscosity μ_0.
- The *Cross* model takes the form

$$\mu = \frac{\mu_0}{1 + (\mu_0 \dot{\gamma}/\tau_0)^{1-n}}, \tag{7.43}$$

  where τ_0 is the shear stress level at which the flow undergoes a transition from the Newtonian nature to the power-law region.

[4] One should notice that μ_0 does not have the dimension of viscosity unless $n = 1$.

- The *Herschel-Bulkley* model is more general:

$$\mu = \begin{cases} \mu_\infty \to \infty, & \tau \leq \tau_0 \\ \tau_0/\dot{\gamma} + \mu_0 \dot{\gamma}^{n-1} \end{cases} \tag{7.44}$$

  where $\tau = 2\mu\dot{\gamma}$ is the effective stress for plastic flow[5] and τ_0 is a critical value of τ for the transition between a rigid-body movement of the fluid ($\mu_\infty \to \infty$) and a modified power-law behavior. Notice that $\tau = 0$ recovers the standard power-law model, whereas $n = 1$ corresponds to a Bingham model.
- For the temperature dependence we may choose

$$\mu_T(T) = \exp\left(-\alpha(T - T_0)\right), \tag{7.45}$$

  which reflects that the viscosity is reduced when the temperature is increased, T_0 being a reference temperature.

Looking at the specific constitutive laws above, we see that the parameters n, α, μ_∞, and τ_0 are central. Setting $n = 1$, $\alpha = 0$, $\tau_0 = 0$, and $\mu_\infty = 0$ results in two decoupled linear PDEs for w and T, where we first can solve for w and then for T. Decreasing n towards zero and increasing α sharpen the coupling and the degree of nonlinearity. The present system of PDEs should therefore be everything from easy to very difficult to solve, depending on the values of n and α in particular.

7.2.2 Numerical Methods

Finite Element Formulation. The nonlinear Poisson equations for w and T are straightforwardly solved by a Galerkin finite element method. We set

$$w(x,y) \approx \sum_{j=1}^{m} w_j N_j(x,y), \quad T(x,y) \approx \sum_{j=1}^{m} T_j N_j(x,y),$$

where $N_j(x,y)$ are finite element basis functions in the grid over Ω. Notice that we use m as the number of nodes, and not n as in previous chapters, because n is a standard symbol for the "power-law exponent" in the literature on generalized Newtonian fluids. Multiplying the PDEs by N_i, integrating over Ω, and integrating the second order derivatives by parts, lead to a system of discrete equations on the form

$$F_i^{(w)}(w_1, \ldots, w_m, T_1, \ldots, T_m) = 0, \tag{7.46}$$

$$F_i^{(T)}(w_1, \ldots, w_m, T_1, \ldots, T_m) = 0, \tag{7.47}$$

[5] τ is similar to the von Mises stress in solid mechanics.

for $i = 1, \ldots, m$. This is a system of $2m$ coupled nonlinear algebraic equations. The exact expressions for $F_i^{(w)}$ and $F_i^{(T)}$ are given below.

$$F_i^{(w)} \equiv \int_\Omega \left(\mu(T,\dot{\gamma})\nabla w \cdot \nabla N_i - \beta N_i\right) d\Omega,$$

$$F_i^{(T)} \equiv \int_\Omega \left(\nabla T \cdot \nabla N_i - \kappa^{-1}\mu(T,\dot{\gamma})\dot{\gamma}^2\right) d\Omega + \int_{\partial\Omega_7} h_T(T - T_s)N_i d\Gamma \ .$$

Solution of Nonlinear Algebraic Equations. When μ depends on $\dot{\gamma}$ or on T, the algebraic equations $F_i^{(w)} = 0$ and $F_i^{(T)} = 0$ are nonlinear. There are two different basic strategies for solving these equations: either (i) solve the $F_i^{(w)} = 0$ and $F_i^{(T)} = 0$ equations in sequence with an outer iteration[6], or (ii) apply a standard nonlinear solution method, like the Newton-Raphson method or Successive Substitutions (Picard iteration) to the compound system $(F_i^{(w)} = 0, F_i^{(T)} = 0)$, and solve for w_i and T_i simultaneously. Strategy (ii) is often referred to as a *fully implicit approach*, whereas strategy (i) will be denoted as *Gauss-Seidel* or *Jacobi* iteration on the PDE level. To see why the names Gauss-Seidel and Jacobi are natural[7], we write the algorithm associated with strategy (i) in more detail. Let q be an iteration parameter. Quantities with superscript q denote approximations in the qth iteration. The Gauss-Seidel procedure can then be expressed as in Algorithm 7.1.

Algorithm 7.1.

Gauss-Seidel-type method for systems of nonlinear PDEs.

for $q = 1, 2, \ldots$ until convergence
  solve $F_i^{(w)}(w_1^q, \ldots, w_m^q, T_1^{q-1}, \ldots, T_m^{q-1}) = 0$
    with respect to $(w_1^q, \ldots, w_m^q)$, $i = 1, \ldots, m$
  solve $F_i^{(T)}(w_1^q, \ldots, w_m^q, T_1^q, \ldots, T_m^q) = 0$
    with respect to $(T_1^q, \ldots, T_m^q)$, $i = 1, \ldots, m$

In other words, we first solve (7.37) with respect to w, using the most recently computed T_i values in the formulas for μ. Thereafter we solve (7.38) with respect to T, using the most recently computed w_i values in the nonlinear term on the right-hand side.

The equation for w is still nonlinear and can be solved by, e.g., a Newton-Raphson method or Successive Substitutions. Jacobi's method is similar to the Gauss-Seidel approach, except that we use the old w_i^{q-1} values when solving for T in iteration q, see Algorithm 7.2.

[6] This is also referred to as an operator-splitting technique.

[7] We refer to Appendix C.1 for an introduction to the ideas of Gauss-Seidel and Jacobi iteration for solving systems of (linear) equations.

Algorithm 7.2.

Jacobi-type method for systems of nonlinear PDEs.

for $q = 1, 2, \ldots$ until convergence
  solve $F_i^{(w)}(w_1^q, \ldots, w_m^q, T_1^{q-1}, \ldots, T_m^{q-1}) = 0$
    with respect to $(w_1^q, \ldots, w_m^q)$, $i = 1, \ldots, m$
  solve $F_i^{(T)}(w_1^{q-1}, \ldots, w_m^{q-1}, T_1^q, \ldots, T_m^q) = 0$
    with respect to $(T_1^q, \ldots, T_m^q)$, $i = 1, \ldots, m$

The attractive feature of the Jacobi or Gauss-Seidel iteration approach to the nonlinear problem is that we only need to solve standard scalar PDEs. The fully implicit approach, on the contrary, requires us to consider a system of two PDEs with two unknowns per node, resulting in nonlinear algebraic equations in $2m$ unknowns.

Let us explain the details of the Newton-Raphson method applied to the fully implicit system of nonlinear algebraic equations. At each node i we have two equations, $F_i^{(w)} = 0$ and $F_i^{(T)} = 0$, and two unknowns w_i and T_i. The total system has $2m$ equations and $2m$ unknowns. We order the equations as follows.

$$F_1^{(w)} = 0, F_1^{(T)} = 0,\ F_2^{(w)} = 0, F_2^{(T)} = 0, \ldots, F_m^{(w)} = 0, F_m^{(T)} = 0\,. \tag{7.48}$$

The corresponding numbering of the unknowns is

$$(w_1, T_1, w_2, T_2, \ldots, w_m, T_m)^T\,. \tag{7.49}$$

This numbering gives smaller bandwidth compared with stacking together all the w equations and unknowns first, followed by all the T equations and unknows, see the next exercise.

Exercise 7.8. As an alternative to the numbering of equations and unknowns in (7.48)–(7.49) we consider

$$F_1^{(w)} = 0, F_2^{(w)} = 0, \ldots, F_m^{(w)} = 0,\ F_1^{(T)} = 0,\ F_2^{(T)} = 0, \ldots, F_m^{(T)} = 0\,. \tag{7.50}$$

and

$$(w_1, w_2, \ldots, w_m, T_1, T_2, \ldots, T_m)^T\,. \tag{7.51}$$

Find the bandwidth of the associated coefficient matrix for each of the two numbering strategies (7.48)–(7.49) and (7.50)–(7.51) by considering a 2D lattice domain with bilinear elements and a line-by-line node numbering. You can assume for simplicity that the equations $F_i^{(w)} = 0$ and $F_i^{(T)} = 0$ are linear, although the same reasoning can easily be extended to the nonlinear case as well. ◇

From each node i we get a 2×2 linear system that is to be included in the element matrix and vector and thereafter assembled into the global $2m\times 2m$ system. In the Newton-Raphson method the local 2×2 system reads

$$\begin{pmatrix} \frac{\partial F_i^{(w)}}{\partial w_j} & \frac{\partial F_i^{(w)}}{\partial T_j} \\ \frac{\partial F_i^{(T)}}{\partial w_j} & \frac{\partial F_i^{(T)}}{\partial T_j} \end{pmatrix} \begin{pmatrix} \delta w_j^q \\ \delta T_j^q \end{pmatrix} = \begin{pmatrix} -F_i^{(w)} \\ -F_i^{(T)} \end{pmatrix} . \tag{7.52}$$

In the matrix and the right-hand side we evaluate the expressions using old values, w^{q-1} and T^{q-1}:

$$F_i^{(w)} \equiv \int_\Omega \left(\mu_T(T^{q-1})\mu_w(\dot\gamma^{q-1})\nabla w^{q-1}\cdot\nabla N_i - \beta N_i\right) d\Omega, \tag{7.53}$$

$$\begin{aligned} F_i^{(T)} \equiv & \int_\Omega \left(\nabla T^{q-1}\cdot\nabla N_i - \kappa^{-1}\mu_T(T^{q-1})\mu_w(\dot\gamma^{q-1})(\dot\gamma^{q-1})^2\right) d\Omega \\ & + \int_{\partial\Omega_7} h_T(T^{q-1}-T_s)N_i d\Gamma . \end{aligned} \tag{7.54}$$

In the case where $\mu_w(\dot\gamma) = \dot\gamma^{n-1}$, the derivatives become as follows.

$$\begin{aligned} \frac{\partial F_i^{(w)}}{\partial w_j} = & \int_\Omega \left(\mu_T(T^{q-1})(n-1)\mu_w(\dot\gamma^{q-1})^{n-2}\frac{\partial\dot\gamma}{\partial w_j}\nabla w^{q-1}\cdot\nabla N_i + \right. \\ & \left. \mu_T(T^{q-1})\mu_w(\dot\gamma^{q-1})^{n-1}\nabla N_j\cdot\nabla N_i\right) d\Omega, \\ \frac{\partial F_i^{(w)}}{\partial T_j} = & \int_\Omega \left(\frac{\partial\mu_T}{\partial T_j}\mu_w(\dot\gamma^{q-1})^{n-1}\nabla N_i\cdot\nabla w^{q-1}\right) d\Omega, \\ \frac{\partial F_i^{(T)}}{\partial w_j} = & -\int_\Omega \left(\kappa^{-1}\mu_T(T^{q-1})(n+1)\mu_w(\dot\gamma^{q-1})^n N_i\frac{\partial\dot\gamma}{\partial w_j}\right) d\Omega, \\ \frac{\partial F_i^{(T)}}{\partial T_j} = & \int_\Omega \left(\nabla N_i\cdot\nabla N_j - \kappa^{-1}\frac{\partial\mu_T}{\partial T_j}\mu_w(\dot\gamma^{q-1})^{n+1}N_i\right) d\Omega \\ & + \int_{\partial\Omega_7} h_T N_i N_j\, d\Gamma, \\ \frac{\partial\dot\gamma}{\partial w_j} = & \mu_w(\dot\gamma^{q-1})^{-1}\nabla N_j\cdot\nabla w^{q-1}, \\ \frac{\partial\mu_T}{\partial T_j} = & -\alpha\mu_T(T^{q-1})N_j . \end{aligned}$$

In the Successive Substitution (Picard iteration) method we also have a 2×2 system at each node:

$$\begin{pmatrix} A_{ww} & A_{wT} \\ A_{Tw} & A_{TT} \end{pmatrix} \begin{pmatrix} w_j^q \\ T_j^q \end{pmatrix} = \begin{pmatrix} b_w \\ b_T \end{pmatrix}, \tag{7.55}$$

where

$$\begin{aligned}
A_{ww} &\equiv \int_\Omega \mu_T(T^{q-1})\mu_w(\dot{\gamma}^{q-1})\nabla N_i \cdot \nabla N_j \, d\Omega, \\
b_w &\equiv \beta \int_\Omega N_i d\Omega, \\
A_{TT} &\equiv \int_\Omega \nabla N_i \cdot \nabla N_j \, d\Omega + \int_{\partial\Omega_7} h_T N_i N_j \, d\Gamma, \\
b_T &\equiv \int_\Omega \kappa^{-1}\mu_T(T^{q-1})\mu_w(\dot{\gamma}^{q-1})(\dot{\gamma}^{q-1})^2) d\Omega + \int_{\partial\Omega_7} h_T N_i T_s \, d\Gamma, \\
A_{wT} &\equiv 0, \\
A_{Tw} &\equiv 0 \, .
\end{aligned}$$

(We remark that the result $A_{wT} = A_{Tw} = 0$ is not a general property of this method.)

A Continuation Method. We know that $n = 1$ is an easy problem to solve, whereas convergence problems are expected as n approaches zero or n is significantly greater than unity. This points in the direction of formulating a continuation method, see Chapter 4.1.8, using $\lambda = (1-n)/(1-n_0)$ as continuation parameter, with n_0 being the target value of n for the computations. By defining a set of proper values $\lambda_0 = 0 < \lambda_1 < \cdots < \lambda_p = 1$ of λ, and using the solution obtained with λ_{i-1} as initial guess for the nonlinear solvers in the problem corresponding to λ_i, we might hope to establish convergence for small n values. The α parameter can be used as continuation parameter in a similar way.

Remark. From the theory and practice of iterative methods for linear systems it is known that Jacobi's method is generally slower than Gauss-Seidel iteration. This is intuitively expected in the present nonlinear situation as well, since the Gauss-Seidel algorithm incorporates new approximations as soon as they are available. Nevertheless, when applying these iterative strategies at the PDE level, Jacobi iteration sometimes have important advantages with respect to conservation properties of the PDEs. For example, a Jacobi method conserves mass in multi-phase reactive flow problems. This is occasionally a fundamental property of the numerical formulation, although it is not of that importance in the present relatively simple flow case.

7.2.3 Implementation

Looking back at the mathematical and numerical model, there are several open questions regarding the choice of constitutive laws and nonlinear iteration strategies. The influence of the element type and preconditioning strategies for the linear systems is also not known. This calls for *flexibility* in the implementation, like we have emphasized many other places in this book. More specifically, a flexible simulation tool must deal with the following aspects of the present problem.

- It must be easy to switch between the Gauss-Seidel, Jacobi, or fully implicit solution strategies.
- The nonlinear algebraic equations in the inner iterations of the Gauss-Seidel, Jacobi, or fully implicit methods must be solved by either Newton-Raphson iteration or Successive Substitutions.
- Several methods must be available for solving the linear systems arising in each nonlinear iteration.
- Any combination of solution strategies for nonlinear and linear equations must be easily available at run time.
- The implementation must work for any finite element grid with any isoparametric element.
- It should be easy to redefine boundary indicators such that the solver can also handle channel flow of Poiseuille or Couette type. Analytical solutions are known for these flow cases and will therefore constitute an important tool in the verification of the implementation.

The purpose of the present section is to introduce a software design for the coupled heat-fluid flow simulator that meets the flexibility requirements listed above. Despite the fact that the present problem is a quite simple coupled problem, the basic software design principles are general and applicable to much more complicated systems of PDEs. The material to be presented constitute a further development and improvement of the ideas from [21].

We assume that the reader is familiar with standard finite element-based PDE solvers in Diffpack. Our basic idea for the present simulator is to develop independent standard Diffpack solvers for the momentum and energy equations and then couple these solver classes. This will work when the PDE system is solved by Gauss-Seidel or Jacobi iteration strategies. We follow the design from Chapter 7.1 and emphasize the possibility to pull the classes apart at any time such that we can check that each PDE in the system is correctly solved when its coefficients are not coupled to other equations. Moreover, we present a way of programming constitutive laws and other common relations in a separate module that can be accessed by all PDE solvers in the system. Chapter 3.5.6 provides valuable background information for the design issues discussed below.

The source code of the coupled heat-fluid flow solver is located in the directory `src/app/Pipeflow`.

The Momentum Equation Solver. Equation (7.37) is implemented in a class `Momentum1`, but with a simple prescribed form of μ to assist the debugging and verification of the implementation. Introducing a virtual function `viscosity` for evaluating μ, `Momentum1` can let `viscosity` return a constant, whereas subclasses of `Momentum1` can implement `viscosity` with a call to physically relevant viscosity models. Class `Momentum1` is similar to class `NlHeat1` from Chapter 4.2, but with automatic report generation facilities built into the class. The nonlinear solver in `Momentum1` is a `NonLinEqSolvers` object, which has the same behavior as class `NonLinEqSolver` and its subclasses, but one can switch between different iteration methods within a call to `solve`. For example, one can apply Successive Substitutions for the first iterations and then switch to Newton-Raphson for hopefully faster convergence when the approximate solution is sufficiently close to the exact solution. A `NonLinEqSolvers` object typically contains an array of `NonLinEqSolver` handles; here we use two such handles, pointing to a `NewtonRaphson` and a `SuccessiveSubst` object.

The `NonLinEqSolver` class offers a function `continuationSolve` that implements the continuation method in Algorithm 4.3 on page 351. We make use of this functionality in the `Momentum1` class. The `NonLinEqSolver_prm` object can read a set of continuation parameters from the menu, and if there are more than one parameter, the `solve` function in `NonLinEqSolver` calls the `continuationSolve` function. Each time this latter function invokes an ordinary nonlinear solve phase, it first calls a virtual function

```
void beforeSolveInContinuationMethod (real lambda, int niter);
```

in the simulator class. The purpose of this function is to use the information about the current value of the continuation parameter, $\Lambda \in [0,1]$ (`lambda`), for adjusting the corresponding physical continuation parameter in the simulator prior to computing the linear systems in each nonlinear iteration. Continuation methods are hence available from the `NonLinEqSolver` tool by just writing a small additional function for linking $\Lambda \in [0,1]$ to a physical parameter. The `niter` parameter reflects the current number of iterations in the continuation method itself.

Class `Momentum2`, which is a subclass of `Momentum1`, implements a new version of the `viscosity` function, where a real generalized Newtonian viscosity model is used: $\mu = \mu_w(\dot{\gamma})\mu_T(T)$. To this end, `Momentum2` needs to access the temperature field. We will come back to the details on how this is achieved technically. One could also think of several stepwise refinements of class `Momentum1`, for example, first making a subclass for a prescribed variable coefficient μ, then a subclass for a nonlinear μ, before the coupling to the real physical viscosity model is realized. The purpose of each step is to create test problems of increasing complexity as this will aid the debugging of the final momentum equation solver. We refer to Chapter 3.5.6 for a detailed explanation of the ideas of making a class hierarchy for solving various versions of $\nabla \cdot [\mu \nabla w] = -\beta$.

The energy equation solver is also realized as a class hierarchy, with class `Energy1` as a stand-alone solver for $\nabla^2 T = f$, $f = \text{const}$, and subclass `Energy2` with the physically relevant $f = \dot{\gamma}^2 \mu_w(\dot{\gamma})\mu_T(T)$.

Software Components for the Coupled System of PDEs. The PDEs are coupled through the coefficients in the equations. We represent these coefficients by virtual functions taking a `FiniteElement` object as argument, e.g.,

```
void Momentum1::viscosity (const FiniteElement& fe)
```

With the `fe` variable at hand we can perform evaluation of explicit formulas as well as efficient interpolation in Diffpack's field objects (see pages 249 and 332 for more information).

The base class solvers have simple versions of the virtual coefficient functions, often corresponding to constant coefficients, such that the solver can be tested separately from the other solvers. Subclasses implement more complicated forms of the variable coefficients and couple the coefficients to other solvers. This coupling can be accomplished by calling a module that holds formulas for μ_w and μ_T.

The coefficients in the original PDEs involve in general a set of constitutive laws that are common to several PDEs. This is the case in the present problem, where both PDE solvers need the quantities $\dot{\gamma}$, $\mu_w(\dot{\gamma})$, and $\mu_T(T)$. All common relations for a system of PDEs can be collected in a separate class, here called `CommonRel`. This class contains physical parameters like μ_0, μ_∞, T_0, α, n, and τ_0, which are initialized using the menu system. In addition, class `CommonRel` needs to interpolate and store the values of w, $\dot{\gamma}$, T, $\nabla\dot{\gamma}$, and so on, to avoid unnecessary recomputation of mathematical expressions. A function `tabulate` performs this task and must be called prior to functions for calculating $\mu_w(\dot{\gamma})$ and $\mu_T(T)$. The latter functions are virtual in class `CommonRel` and can be redefined in subclasses that implement different physical viscosity models. Class `CommonRel` implements the widely used power-law viscosity model, in the slightly extended Sisko form (7.42). A subclass `Cross` implements the Cross model (7.43). Class `CommonRel` is sketched below.

```
class CommonRel : public HandleId
{
public:
  real mu_0;              // μ_0
  real mu_inf;            // μ_∞
  real T_0;               // T_0
  real alpha;             // α
  real n;                 // power law exponent n
  real tau;               // τ_0
  Handle(FieldFE) w, T;   // w(x,y) and T(x,y)

  // tabulated values at a point:
  real w_pt, T_pt;        // w and T
  Ptv(real) dw_pt;        // ∇ w
  real gamma_pt;          // γ̇
```

```
  real gamma_n;          // \dot{\gamma}^n
  real gamma_nm1;        // \dot{\gamma}^{n-1}
  Vec(real) grad_gamma_dot_gradN;  // \nabla\dot{\gamma}\cdot\nabla N_j

  virtual void tabulate (const FiniteElement& fe);

  virtual real muw () const;  // fast evaluation of \mu_w
  virtual real muT () const;  // fast evaluation of \mu_T

  real viscosity () const   { return muw()*muT(); }
  real dissipation () const { return viscosity()*sqr(gamma_pt); }
  ...
};
```

Class `CommonRel` is derived from `HandleId` because we want to access `CommonRel` objects through a handle (cf. page 84).

Newton-like methods require differentiation of the quantities in the constitutive relations with respect to the nodal unknowns w_j and T_j. In a general case with a nonlinear function $f(u, u_{,1}, u_{,2}, u_{,3})$, $u \approx \sum_{j=1}^{n} N_j u_j$, one needs

$$\frac{\partial}{\partial u_j} f(u, u_{,1}, u_{,2}, u_{,3}) = \frac{\partial f}{\partial u} N_j + \sum_{k=1}^{d} \frac{\partial f}{\partial u_{,k}} N_{j,k},$$

which is conveniently implemented in a function returning the vector $\partial f/\partial u_j$ at an evaluation point inside the element. Here, $j = 1, \ldots, n_e$, where n_e is the number degrees of freedom of u in an element. Class `CommonRel` hence offers the functions `dmuw_dwj` for $\partial\mu_w/\partial w_j$ and `dmuT_dTj` for $\partial\mu_T/\partial T_j$.

```
class CommonRel : public HandleId
{
public:
  ...

  virtual void dmuw_dwj (Vec(real)& ddwj, const FiniteElement& fe);
  virtual void dmuT_dTj (Vec(real)& ddTj, const FiniteElement& fe);

  void viscosity_dwj (Vec(real)& ddwj, const FiniteElement& fe)
    { dmuw_dwj(ddwj,fe); ddwj.mult(muT()); }

  void viscosity_dTj (Vec(real)& ddTj, const FiniteElement& fe)
    { dmuT_dTj(ddTj,fe); ddTj.mult(muw()); }
  ...
};
```

So far we have developed separate solvers for the momentum and energy equation, in addition to a common pool of relations that are needed in both PDEs. The final step is to make a class `Manager`, which holds the momentum and energy equation solvers, the common data structures (grid, linear system), a `CommonRel` object, in addition to administering the whole solution process.

```
class Manager : public SimCase
{
public:
  Handle(Energy2)      Energy_eq;
  Handle(Momentum2)    Momentum_eq;
  Handle(CommonRel)    constrel;

  // common data structures for the momentum and energy eq solvers:
  Handle(GridFE)       grid;
  Handle(SaveSimRes)   database;
  Handle(LinEqAdmFE)   lineq;

  virtual void scan ();
  virtual void define (MenuSystem& menu, int level=MAIN);
  virtual void solveProblem ();
  virtual real solveThisIteration ();
  ...
};
```

The `define` and `scan` functions put the local data, like `grid`, `database`, `lineq`, and parameters for the outer Gauss-Seidel/Jacobi nonlinear iteration, on the menu and initializes these data structures. Thereafter the `define` and `scan` functions call the `define` and `scan` functions in the momentum and energy equation solvers. The `solveProblem` function implements the nonlinear outer iteration method. This is basically a loop with calls to `solveThisIteration`. The latter function calls up `solve` functions in the momentum and energy equation solvers (in sequence) to compute new w and T fields. The `solve` functions have the same purpose as `solveProblem` in the a stationary solver like `Poisson1` in Chapter 3.2, but the current solution should not be dumped to file, because we do not yet know if the solution has converged.

An overview of the various classes in the heat-fluid flow simulator is presented in Figure 7.3. We are now able to explain how `Momentum2` can override the `viscosity` function in order to compute a physically relevant expression for μ:

```
class Momentum2 : public Momentum1
{
  Manager* manager;
  virtual real viscosity (const FiniteElement& fe)
    { return manager->constrel->viscosity(); }
  ...
};
```

The Fully Implicit Simulator. It would be advantageous to combine independent solvers for each PDE also in the fully implicit case, but this is technically more difficult. However, using the Diffpack module for generalized and mixed finite element methods, the previously described software design can be applied also to the fully implicit solver, but we will not present the details here. Instead, we realize the fully implicit solution method in class `FullyImplicit`, where the two PDEs are tightly coupled also in the implementation. Recall

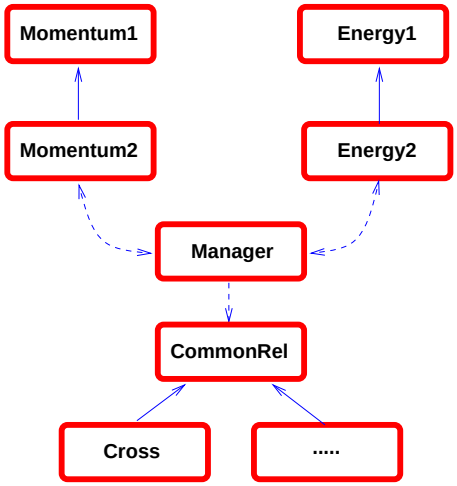
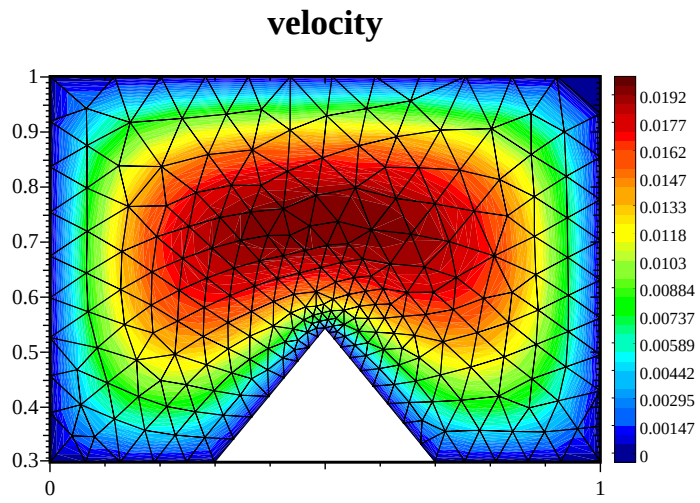


Fig. 7.3. Relation between solver classes and the pool of common relations in the simulator for coupled heat and fluid pipeflow. Solid arrows indicate inheritance ("is-a" relationship). Dashed arrows indicate pointers ("has-a" relationship).

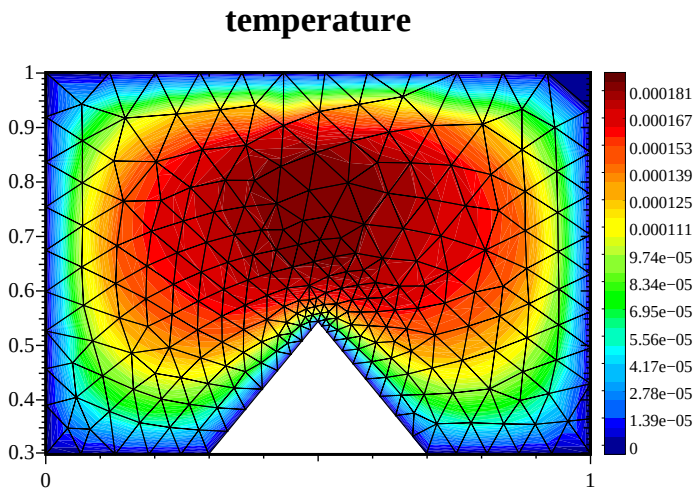
that the nonlinear algebraic equation system has two unknowns at node no. i (w_i and T_i). The related book-keeping is managed by the `DegFreeFE` object, much in the same way as we did in the elasticity solver in Chapter 5.1 and the Navier-Stokes solver in Chapter 6.3. The `integrands` function in class `FullyImplicit` builds the element matrix and vector in terms of blocks as explained in Chapters 5.1 and 6.3. The reader is encouraged to study the numerics of the fully coupled formulation and the corresponding implementation of the `integrands` function. The rest of class `FullyImplicit` is some kind of a sum of class `Momentum1-2` and `Energy1-2`. Class `CommonRel` is of course a valueable tool also in the fully implicit solver.

Without doubt, it is considerably more difficult to code and especially debug class `FullyImplicit` compared to the more modular approach of the classes `Momentum`, `Energy`, and `Manager`. An effective implementation strategy for a fully implicit solver is therefore to first establish a working sequential solver using the component-based software design described in this chapter and then utilize parts of the code and results from test problems when developing a separate implicit simulator.

The source code of the pipeflow simulator employs some constructs that are not completely described in this book. However, the material here gives an overview of the problem and a motivation for the basic features of the software design. Remaining details can be looked up in the man pages. We recommend readers who are interested in a modular approach for solving system of PDEs to study the source code of the pipeflow simulator in detail.



(a)



(b)

Fig. 7.4. Coupled heat and fluid flow in a straight pipe. (a) $w(x, y)$ in a cross section; (b) $T(x, y)$ in a cross section. A power-law viscosity model with $n = 0.8$ and $\alpha = 0$ was used.

7.3 Projects

7.3.1 Transient Thermo-Elasticity

Mathematical Problem. The current project aims at solving the PDE system (3.35) and (5.10), where the u in (3.35) is identical to the temperature T in (5.10).

Physical Model. The elasticity solver in Chapter 5.1 is able to handle the effect of a general thermal "load" $T(\boldsymbol{x})$. A time-dependent temperature field $T(\boldsymbol{x},t)$ is also allowed in the formulation of the governing equations, as long as the time rate of change of T is not so fast that accelerations are induced in the elastic medium. In general, the temperature field $T(\boldsymbol{x},t)$ must be found from a heat equation. The aim of this project is therefore to couple the elasticity solver `Elasticity1` from Chapter 5.1 with the heat equation solver `Heat2` from Chapter 3.10, using the software techniques of Chapters 7.1 or 7.2. The result is a *transient* thermo-elastic solver.

We assume that the (rate of) elastic deformations do not modify the heat equation, such that (3.35) is a sufficient model for the temperature evolution. That is, (5.10) depends on (3.35), but (3.35) is independent of (5.10). A sequential solution approach is hence exact.

Implementation. The coupling of the classes `Elasticity1` and `Heat2` can follow the recipe for coupling in Chapter 7.1.3. Notice that the situation is simpler than in Chapter 7.2 as there is no need for a common pool of fields and constitutive relations; instead the `T` handle in class `Elasticity1` can just be bound to the `u` field in class `Heat2`. Create an administering class that has an `Elasticity1` and a `Heat2` solver as members. At each time level, we first compute the temperature field in the `Heat2` solver and then compute the displacement field and stresses in the `Elasticity1` solver. It is advantageous if the managing class distributes a common grid and a `SaveSimRes` object to the two solvers (i.e. the `define` and `scan` functions in classes `Elasticity1` and `Heat2` should be generalized as demonstrated in Chapters 7.1.3 and 7.2.3).

A basic test problem for verification of the implementation is to consider a box or rod with no normal displacement at two ends and the other sides free of stress. Develop a formula for the stress in the body as a function of a uniform temperature. Arrange the temperature boundary conditions such that the temperature becomes constant in space, but linearly varying in time. (The solvers should then reproduce stress and temperature variations exactly.)

Another suitable test problem is transient thermal loading of a hollow cylinder. The elastic solution for a general temperature variation is listed in [115, Ch. 13]. The temperature field must be found from analytical solution of the transient heat equation in radial coordinates[8], with some appropriate initial and boundary conditions.

[8] This solution involves Bessel functions. The standard C math library offers some Bessel functions, see `math.h`.

Computer Experiments. Consider welding of an elastic material, where the welding model can be as explained in Example 3.1 on page 312. Suggest a suitable geometry, assign stress free conditions at all surfaces of the body, make a scripting interface for efficient handling of simulation and visualization, and demonstrate how the parameters in the welding model from Example 3.1 affect the stress picture in the elastic material.

7.3.2 Convective-Diffusive Transport in Viscous Flow

Mathematical Problem. This project concerns computation of convective-diffusive transport, governed by (6.1), in a fluid whose motion is computed from the Navier-Stokes equations (6.50). That is, we shall solve the system (6.50) and (6.1), where the former couples to the latter through the velocity field v.

Implementation. The coupling can be realized by combining the `CdBase` solver from Chapter 6.1 and the `NsPenalty1` solver from Chapter 6.3, using the software methodology from Project 7.3.1. As the present coupling is almost identical to the one in Project 7.3.1, we recommend to study the text in that project and understand the software details before carrying out the current project. We assume that the velocity field is time-dependent such that both equations must be solved at each time level. (If the velocity field is stationary, it is easier to compute this by the `NsPenalty1` solver and load the field into the `CdBase` solver.)

As test problem for program verification one can try channel flow in a 2D geometry $[0,1]\times[-1,1]$, where the concentration u of a specie in the flow is 0 initially, but with $u = 0.1$ at the inlet boundary for all times $t > 0$. Without diffusion, the specie will be passively transported along the streamlines, that is, $u(x,y,t) = 1 - H(x - U_0(1-y^2)t)$, where $U_0(1-y^2)$ is the inlet velocity profile and H is the Heaviside function. (We remark that it will be hard to approximate $u(x,y,t)$ well).

7.3.3 Chemically Reacting Fluid

Mathematical Problem. The current project considers the temperature distribution in a chemically reacting fluid. A simplified mathematical model, where a chemical specie with concentration a is transformed into another specie, consists of a system of two reaction-diffusion equations:

$$\frac{\partial a}{\partial t} = D\nabla^2 a - sa^m \exp\left(-\frac{E}{RT}\right), \tag{7.56}$$

$$\varrho C_p \frac{\partial T}{\partial t} = \kappa\nabla^2 T + Qsa^m \exp\left(-\frac{E}{RT}\right). \tag{7.57}$$

The functions $a(\boldsymbol{x},t)$ and $T(\boldsymbol{x},t)$ are primary unknowns, while s, m, E, R, D, ϱ, C_p, κ, and Q are viewed as prescribed constants.

Physical Model. Equation (7.56) reflects conservation of the specie mass, where $D\nabla^2 a$ models transport by diffusion and the last term models mass reduction due to chemical reactions. Equation (7.57) is a standard energy equation where the effect of heat generation from chemical reactions is taken into account. The interpretation of the constants in the model is as follows: m is the order of the reaction, E is the activation energy, R is the universal gas constant, s is an adjustable constant, D is the diffusion coefficient of the chemical specie (according to Fick's law), ϱ is the density of the fluid, C_p is the heat capacity of the fluid, κ is the heat conduction coefficient in the fluid (according to Fourier's law), and Q is the heat released by the chemical reactions. A brief description of the model can be found in [71, p. 238].

We remark that (7.56) and (7.57) can be extended to mass and energy transport in a flowing fluid by including appropriate convection terms. If the flow field is unknown, the equations for a and T must be coupled to the Navier-Stokes equations. Using the suggested software design and a sequential solution method, these extensions of the model are easily accomplished. One will often also include more than one substance. This gives rise to a series of equations like (7.56) for S substances $a_1, \ldots, a_S$. Such extensions are also quickly incorporated in our suggested design of the simulator.

Numerical Method. The system of PDEs can be discretized by a finite element method in space and a θ-rule in time. Develop complete expressions for the integrands in the element matrices and vectors, using both a squential and a fully implicit method (follow the ideas in Chapter 7.2.2). Use the Successive Substitution method for solving the nonlinear systems of equations.

Implementation. The implementation can make use of the solvers described in Chapter 7.2.3. The modifications consists in adding functionality for the time-dependency and changing the `integrands` functions. In the sequential solution approach, we make separate solvers for (7.56) and (7.57), and each of these can be tested separately as explained in Chapter 7.2.3. The source terms in (7.56) and (7.57) are conveniently implemented in a `CommonRel` class such that it is easy at a later stage to include additional models for the chemical reactions. Also make a fully implicit solver for the two PDEs.

Computer Experiments. Limit the computational study to a rectangular 2D domain with $\partial a/\partial n = \partial T/\partial n = 0$ on the boundary. The initial concentration a can be taken as a Gaussian bell function (exploit symmetry and place the center of the bell at the lower left corner of the domain). The initial temperature can be constant. Tro to assess the impact of the parameters m, E, κ, and D. Investigate the relative efficiency and stability of the fully implicit versus the sequential solution strategy.