Chapter 6

Fluid Mechanics Applications

This chapter brings together numerical and implementational topics from the previous chapters in three application areas taken from fluid mechanics. First we present a solver for a general time-dependent and possibly nonlinear convection-diffusion equation, where the implementation constitutes a synthesis of most of the Diffpack tools mentioned in Chapters 3 and 4.2. The next application concerns waves in shallow water. We first treat finite difference methods for the system of PDEs on staggered grids in space and time. Thereafter we describe suitable finite element methods for weakly nonlinear and dispersive shallow water waves. The rest of the chapter is devoted to incompressible viscous flow governed by the Navier-Stokes equations. A classical finite difference method on staggered grid in 3D extends the ideas of the finite difference-based numerical model for shallow water waves. A penalty method for the Navier-Stokes equations, in combination with finite element discretization, demonstrates how numerical and implementational tools from the `Poisson2`, `NlHeat1`, and `Elasticity1` solvers in previous chapters can be combined to solve a time-dependent nonlinear vector PDE. Another finite element method for the Navier-Stokes equations, based on operator splitting, is also discussed, with special emphasis on efficient implementation.

6.1 Convection-Diffusion Equations

6.1.1 The Physical and Mathematical Model

The Governing Equations. Convection-diffusion equations appear in a wide range of mathematical models. The particular initial-boundary value problem to be addressed here reads

$$b\left(\alpha\frac{\partial u}{\partial t} + \boldsymbol{v}\cdot\nabla u\right) = \nabla\cdot(k\nabla u) - au + f, \quad \boldsymbol{x}\in\Omega\in\mathbb{R}^d,\ t>0, \tag{6.1}$$

$$u(\boldsymbol{x},0) = I(\boldsymbol{x}), \quad \boldsymbol{x}\in\Omega, \tag{6.2}$$

$$u(\boldsymbol{x},t) = D_1, \quad \boldsymbol{x}\in\partial\Omega_{E_1},\ t>0, \tag{6.3}$$

$$u(\boldsymbol{x},t) = D_2, \quad \boldsymbol{x}\in\partial\Omega_{E_2},\ t>0, \tag{6.4}$$

$$-k\frac{\partial u}{\partial n}(\boldsymbol{x},t) = \nu, \quad \boldsymbol{x}\in\partial\Omega_N,\ t>0, \tag{6.5}$$

$$-k\frac{\partial u}{\partial n}(\boldsymbol{x},t) = h_T(u-U_0), \quad \boldsymbol{x}\in\partial\Omega_R,\ t>0, \tag{6.6}$$

where b, k, $\boldsymbol{v}$, a, f, h_T, and U_0 are functions of $\boldsymbol{x}$ and possibly t, and D_1, D_2, and ν are constants. Moreover, α is an indicator (1 or 0) that turns the time dependence on or off. The complete boundary $\partial\Omega$ consists of the four parts $\partial\Omega_{E_1}$, $\partial\Omega_{E_2}$, $\partial\Omega_N$, and $\partial\Omega_R$, having Dirichlet, Neumann, and Robin conditions.

Convection-diffusion equations are of particular importance in heat transfer and specie transport problems. Moreover, such equations also arise in the intermediate steps of numerical methods for the Navier-Stokes equations and multi-phase porous media flow.

Physical Interpretations. In the field of heat transfer, equation (6.1) stems from the first law of thermodynamics and expresses energy balance of a continuous medium. The primary unknown $u(\boldsymbol{x}, t)$ represents the temperature, b is the product of the density of the medium times the heat capacity, $\boldsymbol{v}$ is the velocity of the medium, k is the medium's heat conduction coefficient, and $f - au$ represents external heat sources. The time-derivative term is the accumulation of internal energy at a fixed point in space, the *convective* term $\boldsymbol{v} \cdot \nabla u$ models transport of internal energy with the flow, $\nabla \cdot (k \nabla u)$ reflects transport of thermal energy by molecular vibrations (i.e. heat conduction), and the source term $f - au$ might represent heat generation or extraction due to, for example, internal friction in the fluid, chemical reactions, or radioactivity. The boundary condition (6.6) is explained in Project 2.6.1.

One can also interpret equation (6.1) as a mass conservation equation that governs specie transport in a fluid. In this case, u is the concentration of the specie, b is the density of the specie, $\boldsymbol{v}$ is the velocity field of the fluid, k is a diffusion coefficient, which is normally constant, and $f - au$ represents specie production or destruction. The time-derivative term expresses accumulation of mass at a point in space, while the convection $(\boldsymbol{v} \cdot \nabla u)$ and the diffusion $(k \nabla^2 u)$ terms reflects transport of mass with the flow and due to molecular diffusion, respectively. The source or sink term $f - au$ might model, for instance, injection or extraction of the specie or mass loss/gain due to chemical reactions. If the heat transfer or the specie transport takes place in a porous medium, the governing PDE is still the same, but the interpretation of the coefficients must be slightly adjusted.

It must also be mentioned that special cases of equation (6.1) appear in many other branches of engineering and science. For example, simple model equations like the Laplace, Poisson, and Helmholtz equations are contained in (6.1). We can also make (6.1) nonlinear, e.g., by letting $k = k(u)$ and replacing $f - au$ by $f(u)$. Such nonlinearities arise both in simple model problems as well as in the heat transfer (cf. Chapter 1.3.7) and specie transport problems.

6.1.2 A Finite Element Method

By means of a θ-rule in time and the weighted residual method in space we can derive the following discrete equations:

$$\int_\Omega \Big[\left(\theta b^\ell + (1-\theta)b^{\ell-1}\right)\alpha\left(\hat{u}^\ell - \hat{u}^{\ell-1}\right)W_i + \theta\Delta t b^\ell W_i \boldsymbol{v}^\ell\cdot\nabla\hat{u}^\ell + (1-\theta)\Delta t b^{\ell-1}W_i\boldsymbol{v}^{\ell-1}\cdot\nabla\hat{u}^{\ell-1} + \theta\Delta t k^\ell\nabla W_i\cdot\nabla\hat{u}^\ell + (1-\theta)\Delta t k^{\ell-1}\nabla W_i\cdot\nabla\hat{u}^{\ell-1} + \theta\Delta t W_i a^\ell\hat{u}^\ell + (1-\theta)\Delta t W_i a^{\ell-1}\hat{u}^{\ell-1} - \left(\theta\Delta t W_i f^\ell + (1-\theta)\Delta t W_i f^{\ell-1}\right)\Big] d\Omega + \int_{\partial\Omega_N} W_i\Delta t\nu d\Gamma + \int_{\partial\Omega_R} W_i\Delta t\left[h_T^\ell\theta(u^\ell - U_0^\ell) + h_T^{\ell-1}(1-\theta)(u^{\ell-1}-U_0^{\ell-1})\right]d\Gamma = 0 \quad (6.7)$$

Superscript ℓ denotes the time level, $\hat{u}^\ell(\boldsymbol{x}) = \sum_{j=1}^n u_j^\ell N_j(\boldsymbol{x})$ is an approximation to $u(\boldsymbol{x},t)$ at time level ℓ, and (6.7) is supposed to hold for n linearly independent weighting functions W_i, $i = 1,\ldots,n$. If some of the details in the derivation of (6.7) are unclear, we refer to similar examples in Chapter 2.

The formula for the element matrix follows from restricting the domain of integration to an element, replacing $\hat{u}^\ell$ by $\sum_j u_j^\ell N_j$ and collecting the terms at level ℓ containing the indices i and j. The remaining terms belong to the element vector.

Exercise 6.1. Write down the precise expressions for the integrands of the element matrix and vector associated with (6.7). ◇

6.1.3 Incorporation of Nonlinearities

A flexible convection-diffusion solver must handle nonlinear coefficients. Here we suppose that b, k, and f can possibly depend on u. To solve the resulting system of nonlinear algebraic equations, we introduce an iteration with q as iteration index, and where $\hat{u}^{\ell,q}$ is the approximation to $\hat{u}^\ell$ in iteration q. The Successive Substitution method (Picard iteration) implies that one simply evaluates the expressions b^ℓ, k^ℓ, and f^ℓ as $b(\hat{u}^{\ell,q-1})$, $k(\hat{u}^{\ell,q-1})$, and $f(\hat{u}^{\ell,q-1})$. The corresponding modifications of the expressions in the element matrix and vector are trivial to incorporate.

As usual, the Newton-Raphson method involves more book-keeping. A term like $W_i b(\hat{u}^\ell)\hat{u}^\ell$ now gives the contribution

$$\frac{\partial}{\partial u_j^\ell}\left(W_i b(\hat{u}^\ell)\hat{u}^\ell\right) = W_i b(\hat{u}^{\ell,q-1})N_j + W_i\frac{db}{du}(\hat{u}^{\ell,q-1})N_j\hat{u}^{\ell,q-1}$$

to the integrands in the expression for the element matrix. Notice that only the first term is used in the Successive Substitution method.

Exercise 6.2. Derive the precise expressions for the integrands of the element matrix and vector when b, k, and f can depend on u. Try to write the expressions in a form that is valid both in the Successive Substitution and Newton-Raphson methods (introduce for example an on-off indicator as coefficient in the Newton-Raphson-specific terms related to derivatives of b, k, and f). ◇

6.1.4 Software Tools

Ideally, we would like to have a flexible solver for the linear model problem (6.1)–(6.6), with a fast specialized version in the case the coefficients are not time dependent and the matrix assembly process can be avoided at each time level, *and* another version that treats the computationally more demanding problem when b, k, and f depend on u. The different program modules should share as much common code as possible. This is straightforwardly realized using the ideas of Chapter 3.5.7 and the optimization technique from Appendix B.7.2.

We create a base class `CdBase` that implements the linear version of (6.1)–(6.6) in a flexible way, with the coefficients b, k, $\boldsymbol{v}$, a, f, and U_0 as virtual functions that can be customized in specialized subclasses written by a user. The default implementation of these functions in class `CdBase` applies the general field representation `Handle(Field)` for the coefficients, as we explained in Chapter 3.15.4. This means that we typically implement f as

```
virtual real f (const FiniteElement& fe, real t = DUMMY)
  { return f_field->valueFEM (fe, t); }
```

where `f_field` is of type `Handle(Field)`. A `FieldFormat` object is used to allocate and initialize `f_field` based on menu information at run time. In `integrands` we call `f` before the loop over the element matrix and vector entries,

```
const real f_value = f(fe,t);
```

A subclass `CdEff` specializes class `CdBase` in the case where the coefficients and boundary conditions in the PDE are time independent. Two matrices are then assembled initially, and the actual coefficient matrix and right-hand side in the linear system at each time level are obtained by efficient matrix-vector operations. The algorithms and software tools are explained in Appendix B.7.2.

Another subclass `CdNonlin` of `CdBase` implements the nonlinear version of (6.1)–(6.6). New virtual functions for db/du, dk/du, and df/du are introduced. The typical representation of, for example, the function k in the code becomes

```
virtual real ku   (real u, const FiniteElement& fe, real t = DUMMY)
  { return u*u/2; }
virtual real dkdu (real u, const FiniteElement& fe, real t = DUMMY)
  { return u; }
```

Here $k(u, \boldsymbol{x}, t) = u^2/2$ is just an example. In `integrands` we evaluate k and dk/du by statements like

```
const real u_pt       = u->valueFEM(fe);  // u at current point
const real k_value    = ku   (u_pt, fe, t);
const real dkdu_value = dkdu (u_pt, fe, t);
```

Notice that we actually do not use the function for k as defined in the base class `CdBase`, i.e. a `k(const FiniteElement&, real)` function as we had in the solvers in Chapter 3, because we find it more convenient to have u as an explicit argument. However, the `k` function is convenient when computing the flux by the `FEM::makeFlux` function, and its proper form for this purpose is

```
real CdNonlin:: k (const FiniteElement& fe, real t)
{
  const real u_pt = u->valueFEM(fe); return ku (u_pt, fe, t); }
}
```

In `integrands` it is better to use `ku` instead of `k` since this allows precomputation of `u_pt` and reuse in several functions.

Particular expressions for the coefficients $b(u)$, $k(u)$, and $f(u)$, as well as their derivatives, must be hardcoded in subclasses of `CdNonlin`. Class `CdNonlin` must also implement a generalized edition of `integrands` and `integrands4side`.

In the case (6.1) is convection dominated, the numerical solution can develop nonphysical oscillations at high mesh Peclet numbers $\mathrm{Pe}_\Delta = b||\boldsymbol{v}||h/k$ (h reflects the element size). Chapters 2.9 and 3.9 outline algorithms and software tools for handling numerical problems associated with convection-dominated phenomena. Class `UpwindFE` is a convenient tool for representing different choices of W_i in the `CdBase` class.

Figure 6.1 depicts the class hierarchy for the convection-diffusion solver. The source code is located in the directory `src/app/Cd`.

Exercise 6.3. Suppose you want to apply the suggested framework as basis for your own software development, but that you need to make an efficient solver for the Poisson equation on grids with linear triangular elements. Although class `CdBase` will work in this problem, the implementation of `integrands` can be made much more efficient (see Appendix B.7.3). Suggest how to derive a subclass `CdTriPoisson` where you rely on data structures in class `CdBase`, but avoid the `integrands` function and fill analytically integrated expressions for the element matrix and vector directly in the `ElmMatVec` object in the `calcElmMatVec` function. The material in Chapter 2.7.3 is useful for developing the relevant analytical expressions. ◇

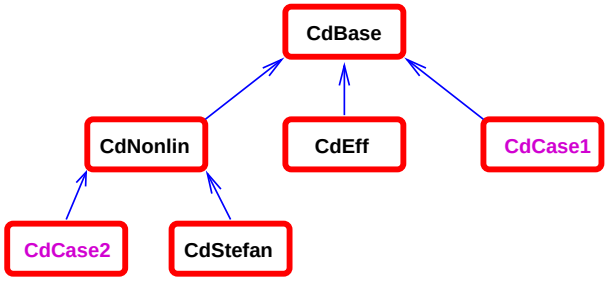

Fig. 6.1. A sketch of the convection-diffusion solver, with the base class `CdBase`, the efficient implementation `CdEff`, the more general nonlinear convection-diffusion solver `CdNonlin`, and the specialization `CdStefan` of `CdNonlin` to solve problems with freezing or melting. The classes `CdCase1` and `CdCase2` just indicate possible user-defined small classes that customize the b, k, and f functions in a particular problem. The arrows indicate class derivation.

Remark. The framework for the convection-diffusion solver as sketched in this chapter is quite flexible, but some users may find it too flexible and not very easy to use for a novice C++ programmer. In such cases it is advantageous to build an interface to the convection-diffusion solver and only apply the class hierarchy as a hidden computational engine. The ideas from Chapter 3.12.9 can be used as a starting point for building an easy-to-use, perhaps graphical, interface in (e.g.) Python. The interface should allow the user to change only some of the input data to the solver, while others are kept at suitable default values. The script must process input data from the user, run the simulator, and visualize the results. Further development of such scripting interfaces might lead to truly easy-to-use flexible simulation environments, where the user can assign even mathematical expressions to b, k, etc. in the interface and interactively specify the grid and boundary conditions. Diffpack was designed for being a flexible computational engine in such problem solving environments.

6.1.5 Melting and Solidification

Many heat transfer applications also involve solidification or melting, i.e., phase changes. An example is heat conduction in a fluid, where parts of the fluid are frozen, while other parts are in a liquid state. The interface between the frozen and melted region is an unknown moving internal boundary. Let us consider a basic one-dimensional mathematical model for such a problem. There are two substances, denoted by the subscripts s (solid) and l (liquid). At the temperature $T = T_b$, a phase change between solid and liquid takes place. At time t we assume that the liquid part of the substance occupies the region $0 \leq x \leq b(t)$, whereas the solid part is located for $b(t) < x \leq a$. In

each of these domains, a heat conduction equation is valid:

$$\varrho_l C_l \frac{\partial T_l}{\partial t} = \kappa_l \frac{\partial^2 T_l}{\partial x^2}, \quad 0 < x < b(t),\ t > 0, \tag{6.8}$$

$$\varrho_s C_s \frac{\partial T_s}{\partial t} = \kappa_s \frac{\partial^2 T_s}{\partial x^2}, \quad b(t) < x < a,\ t > 0, \tag{6.9}$$

$$T_l = \tau_0, \quad x = 0, \tag{6.10}$$

$$T_s = \tau_a, \quad x = a\,. \tag{6.11}$$

Here, ϱ is the density, C is the heat capacity, T is the temperature, κ is the heat conduction coefficient, and τ_0 and τ_a are prescribed temperatures at the end points of the domain. At the interface $x = b$ we have continuity in the temperature and a jump in the heat flux:

$$\kappa_s \frac{\partial T_s}{\partial x} - \kappa_l \frac{\partial T_l}{\partial x} = L \frac{db}{dt}, \quad x = b(t),\ t > 0, \tag{6.12}$$

$$T_l = T_s = T_b, \quad x = b(t),\ t > 0, \tag{6.13}$$

with L being the latent heat of phase transformation per unit volume. One difficulty with such a moving boundary problem is that different PDEs must be solved in different parts of the domain $(0, a)$. However, in the present problem it is possible to formulate a unified PDE that can be solved over the whole domain $(0, a)$, with the interface condition (6.12) being automatically satisfied without explicitly tracking the internal boundary. The key to this simplification is to employ an *enthalpy* formulation.

We introduce the enthalpy $H(T)$ according to

$$H(T) = \begin{cases} \varrho_s C_s T, & T < T_b \\ \varrho_l C_l T + L, & T > T_b \end{cases}$$

with

$$\varrho_s C_s T \le H(T) \le \varrho_l C_l T + L, \quad T = T_b\,.$$

The PDEs and the interface conditions in (6.8)–(6.13) can now be recast in the unified form

$$\frac{\partial H}{\partial t} = \frac{\partial}{\partial x}\left(\kappa(T)\frac{\partial T}{\partial x}\right), \quad 0 \le x \le a,\ t > 0, \tag{6.14}$$

where

$$\kappa(T) = \begin{cases} \kappa_s, & T < T_b \\ \kappa_l, & T > T_b \end{cases}$$

Usually, one solves (6.14) with respect to H. We then need the function $T(H)$:

$$T(H) = \begin{cases} H/(\varrho_s C_s), & H < \varrho_s C_s T_b \\ T_b, & \varrho_s C_s T_b \le H \le \varrho_l C_l T_b + L \\ (H - L)/(\varrho_l C_l), & H > \varrho_l C_l T_b + L \end{cases} \tag{6.15}$$