NITRO

A parallel solver of nonlinear balance systems

Gustavo Hime

FLUID DYNAMICS LABORATORY
IMPA, RIO DE JANEIRO, BRAZIL

In this document we describe the modeling of certain classes of phenomena, using balance or conservation equations, and the corresponding computer implementations. These models describe the evolution of physical systems along time, where transport, diffusion and reaction phenomena may or may not be present. In order to use these models in the effective study of these phenomena, high quality numerical simulations must be performed. But increasing the quality of a simulation invariably increases its computational cost. We do not wish to lessen the quality of the numerical solution for performance sake: our primary goal (motivation) is to perform the best possible simulations. The secondary goal (constraint) is to make optimal use of the computational resources employed.

We draw the common denominator of the many models presented herein in order to produce a single family of high performance simulators for all of them. The purpose of the document is therefore threefold:

- Each model is discussed in its own right, with its assumptions and simplifications. From this mathematical starting point, one or more numerical schemes are applied to the model equations, and the resulting numerical problems and its solutions are discussed.
- Each numerical scheme is formally derived. For each physical model there is at least one particular variant of the general numerical method, and all these variants are covered independently of which physical models will make use of it.
- The rationales of the actual computer program are presented. We provided cross-referencing between the mathematical formulations and their implementation. Issues that arise only at the point of computer implementation are discussed separately from the mathematical issues in the previous two items.

The comprehensive understanding of these three intertwined aspects of numerical simulation is required in order to properly engineer such computer software. The purpose of this document is to provide sufficient insight from all three perspectives and blend them into one seamless whole, enabling newcomers from any related background to understand the complete picture. Doing so fulfills a third goal, that of protecting the further development of the simulator from events such as changes in the group of people involved.

Index

CHAPTER 1

Balance Systems

Many physical phenomena are studied using models based on equations that describe the conservation of mass and of other quantities along time. These are typical problems in fluid mechanics: in particular, problems related to multi–phase flow in porous media have been the object of intense investigation in the past few decades, as they are related to economically and socially critical applications such as petroleum recovery, water supply and soil decontamination. Efficient exploitation of these limited natural resources requires good understanding of the effects the different exploitation approaches have.

Coherent and consistent models for these and other phenomena exist and have been exhaustively studied with the techniques available up to now. In the early days of their conception, when numerical computing was still unavailable or unaffordable, most studies were done on purely theoretical ground. The analytical solutions, however, are more often than not hard to obtain, if not impossible. Moreover, they do not always provide direct insight to the behavior of the model in realistic situations, but only in highly idealized ones with restrictive assumptions. The relaxation of these assumptions may render the analytical solution unfeasible, and requires the model to be studied through numerical simulations.

In this chapter, we introduce the class of problems that we seek to solve. We introduce notation that will be used in each particular problem, and which will map into the naming conventions used in the computer code.

1.1. Problem Statement

We seek the numerical solution of a one–dimensional initial/boundary value problem governed by a system of M partial differential equations in $\mathbf{u}(x,t) \in \mathbb{R}^M$ which we denote, in its most general form, by

$$\widetilde{\boldsymbol{\mathcal{E}}}(\mathbf{u}) = \mathbf{0}, \quad \text{with} \quad \widetilde{\boldsymbol{\mathcal{E}}} : \mathbb{R}^M \to \mathbb{R}^M; \tag{1.1}$$

the state vector $\mathbf{u}$ contains the M model variables $u^{(m)}$, $m = 1, \ldots, M$, defined in the (x,t) physical domain $x \in [x_a, x_b]$ and $t \geq 0$, governed by a nonlinear PDE $\widetilde{\boldsymbol{\mathcal{E}}} : \mathbb{R}^M \to \mathbb{R}^M$. Similarly, we denote each of the M scalar equations in $\widetilde{\boldsymbol{\mathcal{E}}}$ as $\widetilde{\mathcal{E}}^{(m)}$, so $\widetilde{\mathcal{E}}^{(m)} : \mathbb{R}^M \to \mathbb{R}$.

If $\widetilde{\boldsymbol{\mathcal{E}}}$ contains a first order derivative in time ($\partial/\partial t$) it is a time evolution problem; if only spatial derivatives are present, it is a steady state problem. Any other terms expressing spatial derivatives of up to second order are admissible. Clearly, the general reaction–convection–diffusion equation for incompressible flows

$$\frac{\partial \mathbf{h}(\mathbf{u})}{\partial t} + \frac{\partial \mathbf{f}(\mathbf{u})}{\partial x} = \frac{\partial}{\partial x}\left(\mathbf{g}(\mathbf{u})\frac{\partial \mathbf{u}}{\partial x}\right) + \mathbf{q}(\mathbf{u}), \quad \text{with} \quad \mathbf{u} \in \mathbb{R}^M, \tag{1.2}$$

where $\mathbf{f}, \mathbf{h}, \mathbf{q} : \mathbb{R}^M \to \mathbb{R}^M$ and $\mathbf{g} : \mathbb{R}^M \to \mathbb{R}^{M\times M}$ are differentiable, is a particular example of (1.1), and many — if not most — of the models we target are actually particular cases of (1.2): these will be discussed in Chapter 2.

However, we wish to maintain a higher level of generality and use (1.1) as starting point, because it can also represent the equation for compressible flows

$$\frac{\partial \mathbf{h}(\mathbf{u})}{\partial t} + \frac{\partial v\mathbf{f}(\mathbf{u})}{\partial x} = \frac{\partial}{\partial x}\left(\mathbf{g}(\mathbf{u})\frac{\partial \mathbf{u}}{\partial x}\right) + \mathbf{q}(\mathbf{u}), \quad \text{with} \quad \mathbf{u} \in \mathbb{R}^M, \tag{1.3}$$

where $p \equiv u^{(M)}$ denotes the pressure and v is the flow velocity, usually related to p through Darcy's Law

$$v = -k(\mathbf{u})\frac{\partial p}{\partial x} + r(\mathbf{u}), \tag{1.4}$$

where $r(\mathbf{u})$ is the gravity term. The continuity equation for the total flow is one of the M scalar equations in (1.3): we denote it as the M-th equation. Usually it has the form

$$\frac{\partial h^{(M)}(\mathbf{u})}{\partial t} + \frac{\partial v f^{(M)}(\mathbf{u})}{\partial x} = 0, \tag{1.5}$$

i.e., $g^{(M)} \equiv 0$ and $q^{(M)} \equiv 0$.

Although v is not a parameter of the $\mathbf{h}$, $\mathbf{f}$, $\mathbf{g}$ and $\mathbf{q}$ functions, it is also a state variable. The nonlinear PDE (1.1) becomes

$$\widetilde{\boldsymbol{\mathcal{E}}}(\mathbf{u}; v) = \mathbf{0}, \quad \text{with} \quad \widetilde{\boldsymbol{\mathcal{E}}} : \mathbb{R}^{M+1} \to \mathbb{R}^{M+1}; \tag{1.6}$$

the equations $\widetilde{\mathcal{E}}^{(m)}$, $m = 1, \ldots, M$ are taken from $\mathbf{f}, \mathbf{h}, \mathbf{q} : \mathbb{R}^M \to \mathbb{R}^M$ and $\mathbf{g} : \mathbb{R}^M \to \mathbb{R}^{M\times M}$ like in the simpler version (1.2), and $\widetilde{\mathcal{E}}^{(M+1)}$ is obtained from (1.4).

Before we apply finite difference discretizations to (1.1), we must define how the physical domain itself is discretized. In the following subsections, we present two different approaches to best suit either equation (1.2) or (1.3). In both cases, the grid is uniformly spaced: the use of a nonuniform grid would greatly encumber the following derivations, and is not of primary interest.

1.1.1. Simple uniform grid. For flow models where pressure and velocity are not state variables, we use a simple grid in space, i.e., we discretize the x domain $[x_a, x_b]$ into $N-1$ subintervals of length $\delta x = (x_b - x_a)/(N-1)$, delimited by points x_i, $i = 1, \cdots, N$, with $x_1 = x_a$ and $x_N = x_b$. Boundary conditions are enforced through the equations related to the boundary nodes, as will be explained in section 1.5. The total number of grid points is N, including the boundaries.

As for the temporal dimension, we adopt discrete intervals of length δt for t, i.e., $t^n - t^{n-1} = \delta t$, and we index the times t^n, $n = 1, \cdots$, with $t^1 \equiv 0$. Figure 1.1 depicts this simple uniform grid.

For all purposes, state variables — e.g. saturations — are defined on each grid node as a point value. For visualization purposes, we perform linear interpolation between grid nodes.

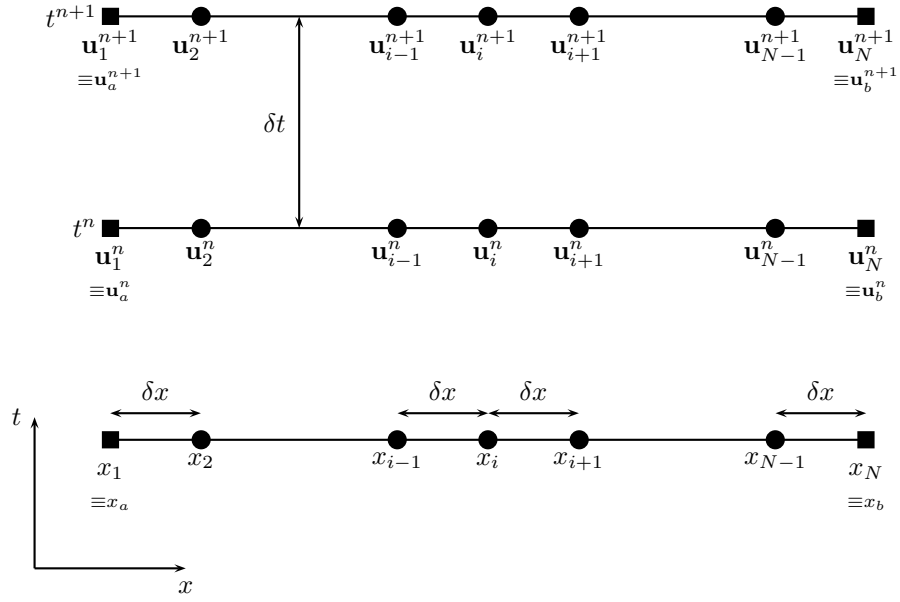


FIGURE 1.1. One–dimensional uniform grid. The boundary nodes are singled out as squares, and the inner nodes are shown as circles.

1.1.2. Staggered uniform grid. For models where pressure and velocity are state variables, a "double", "dual" or "staggered" grid is required, shown in Figure 1.2. The x domain $[x_a, x_b]$ is divided into $N-2$ subintervals of length $\delta x = (x_b - x_a)/(N-2)$, and the "primary" grid points x_i, $i = 2, \cdots, N-1$ correspond to the center of each such interval, with $x_2 = x_a + \delta x/2$ and $x_{N-1} = x_b - \delta x/2$. Each "secondary" grid point $x_{\widehat{i}}$ is located at $x_i + \delta x/2$. All state variables are defined on the primary grid points except the velocity, which is defined on the secondary grid. Boundary conditions are enforced through the nodes $x_1 = x_a - \delta x/2$ and $x_N = x_b + \delta x/2$, *outside* the physical domain, usually referred to as "ghost" nodes. The boundaries correspond to secondary grid locations $x_{\widehat{1}}$ and $x_{\widehat{N-1}}$.

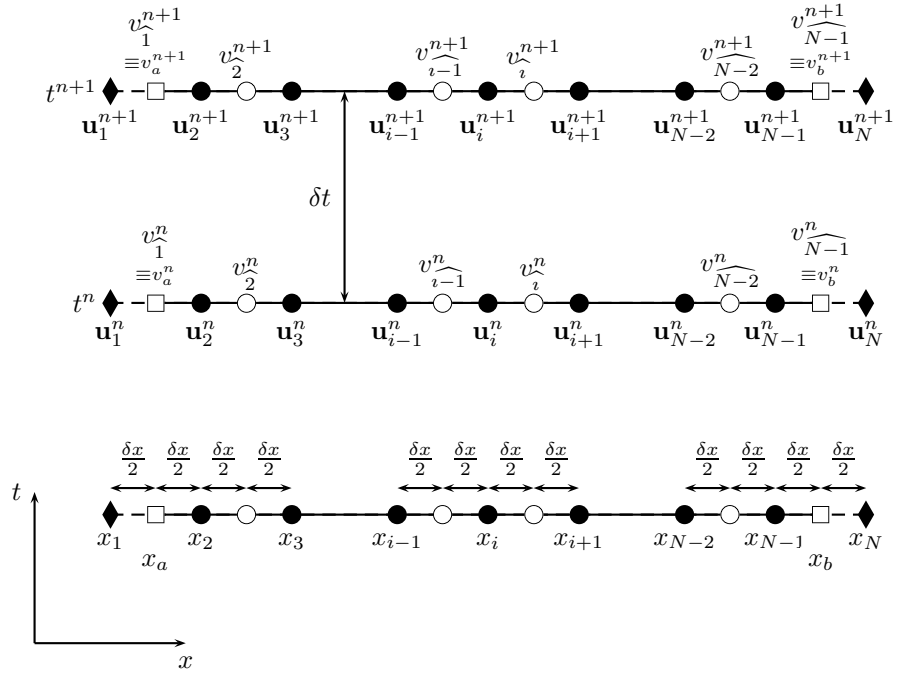


FIGURE 1.2. One–dimensional staggered grid. The primary locations are filled, and the secondary locations are outlined. The circles are inside the physical domain, the boundaries are represented by squares, the ghost nodes are represented by diamonds. All primary state variables are defined at the primary locations, and the velocity v is staggered midway between each two primary grid nodes.

1.1.3. Discrete problem. To lighten the notation, we denote $\mathbf{u}(x_i, t^n) \equiv \mathbf{u}_i^n$. The discretization of the continuous $\widetilde{\boldsymbol{\mathcal{E}}}$ given by (1.2) on the simple grid yields an expression of the form

$$\boldsymbol{\mathcal{E}}(\mathbf{u}_{i-1}^{n+1}, \mathbf{u}_i^{n+1}, \mathbf{u}_{i+1}^{n+1}; \mathbf{u}_{i-1}^n, \mathbf{u}_i^n, \mathbf{u}_{i+1}^n), \tag{1.7}$$

using three spatial nodes and two time levels to represent all admissible derivatives that may be present in $\widetilde{\boldsymbol{\mathcal{E}}}$. The expression for the staggered grid would include the velocities at positions $\widehat{\imath - 1}$ and $\widehat{\imath}$ as well.

In steady state problems, there is only one time level to be considered. In time evolution problems, the values at time level n are known, whereas the unknowns are at time $n+1$; we consider initial conditions corresponding to the Cauchy problem

$$\mathbf{u}(x, 0) = \mathbf{u}^0(x),\ x_a \le x \le x_b;$$

so the values at time t^n in (1.7) are always known; thus, in both steady state and evolution cases, the actual discrete operator to be taken into consideration has the form

$$\boldsymbol{\mathcal{E}}(\mathbf{u}_{i-1}, \mathbf{u}_i, \mathbf{u}_{i+1}), \quad \text{with} \quad \boldsymbol{\mathcal{E}} : \mathbb{R}^{3M} \to \mathbb{R}^M.$$

The discrete problem in the simple grid can be written as a nonlinear system

$$\boldsymbol{\Xi}(\mathbf{u}) = 0 \quad \text{with} \quad \boldsymbol{\Xi}_i \equiv \boldsymbol{\mathcal{E}}(\mathbf{u}_{i-1}, \mathbf{u}_i, \mathbf{u}_{i+1}), \quad i = 2, \ldots, N-1, \tag{1.8}$$

which is underdetermined, since it has N unknowns and $N-2$ equations in $\mathbb{R}^M$. By the preceeding apparent clash of nomenclature we mean that the $(N-2)M$-sized system $\boldsymbol{\Xi}(\mathbf{u})$ is the set of $(N-2)$ vector equations $\boldsymbol{\Xi}_i$, each of size M.

The system (1.8) becomes fully determined by imposing boundary conditions, which may vary. In the next section, we consider those corresponding to the Dirichlet problem:

$$\mathbf{u}(x_a, t) = \boldsymbol{\gamma}_a(t), \quad \mathbf{u}(x_b, t) = \boldsymbol{\gamma}_b(t). \tag{1.9}$$

We introduce the full solution procedure for this particular, simpler case. In Section 1.5 we extend the solution procedure to allow for other types of boundary conditions.

Fixing the boundary values $\mathbf{u}_1 = \boldsymbol{\gamma}_a$ and $\mathbf{u}_N = \boldsymbol{\gamma}_b$, we reduce the number of unknowns to $N-2$. Notice that the full equation $\boldsymbol{\Xi}_2$ is given by

$$\boldsymbol{\mathcal{E}}(\mathbf{u}_1^{n+1}, \mathbf{u}_2^{n+1}, \mathbf{u}_3^{n+1}; \mathbf{u}_1^n, \mathbf{u}_2^n, \mathbf{u}_3^n) = 0,$$

so the value of $\mathbf{u}_1^n = \boldsymbol{\gamma}_a^n$ may differ from $\mathbf{u}_1^{n+1} = \boldsymbol{\gamma}_a^{n+1}$: still, they are both given in (1.9). The same holds for the other boundary.

1.2. Classic Newton's Method

We apply Newton's method to find the root of $\boldsymbol{\Xi}(\mathbf{u})$, i.e., the value of $\mathbf{u}$ for which $\boldsymbol{\Xi}(\mathbf{u}) = 0$: from an initial guess $\mathbf{u}^{(0)}$, we proceed using Newton's iteration

$$\mathbf{u}^{(l+1)} = \mathbf{u}^{(l)} + \delta\mathbf{u}^{(l)}, \quad \text{where} \quad \frac{\partial \boldsymbol{\Xi}}{\partial \mathbf{u}}(\mathbf{u}^{(l)})\, \delta\mathbf{u}^{(l)} = -\boldsymbol{\Xi}(\mathbf{u}^{(l)}) \tag{1.10}$$

until some convergence criteria are satisfied. At a given Newton l-iteration, the problem is the construction and solution of a linear system. From the definition of $\boldsymbol{\Xi}_i$, the Jacobian matrix $\mathbf{J} = \dfrac{\partial \boldsymbol{\Xi}}{\partial \mathbf{u}}$ in (1.10) is block tridiagonal, with $N-2 \times N-2$ blocks of size $M \times M$. We denote

$$\mathbf{B}_i \equiv \mathbf{J}_{i,i-1} = \frac{\partial \boldsymbol{\Xi}_i}{\partial \mathbf{u}_{i-1}}, \quad \mathbf{A}_i \equiv \mathbf{J}_{i,i} = \frac{\partial \boldsymbol{\Xi}_i}{\partial \mathbf{u}_i}, \quad \text{and} \quad \mathbf{C}_i \equiv \mathbf{J}_{i,i+1} = \frac{\partial \boldsymbol{\Xi}_i}{\partial \mathbf{u}_{i+1}}. \tag{1.11}$$

The i subscripts corresponds to grid position in the expressions for derivatives, mapped into matrix rows and columns of the jacobian $\mathbf{J}$, mapped into block array indexes in $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$.

These partial derivatives are known once $\widetilde{\boldsymbol{\mathcal{E}}}$ is defined: then the matrix coefficients given in (1.11) and the right hand side $\boldsymbol{\Xi}(\mathbf{u})$ can be computed, and $\delta\mathbf{u}$ can be determined by

solving the linear system

$$\begin{bmatrix} \mathbf{A}_2 & \mathbf{C}_2 & 0 & 0 & 0 & \cdots & & & \\ \mathbf{B}_3 & \mathbf{A}_3 & \mathbf{C}_3 & 0 & 0 & 0 & \cdots & & \\ 0 & \mathbf{B}_4 & \mathbf{A}_4 & \mathbf{C}_4 & 0 & 0 & 0 & \cdots & \\ 0 & 0 & \mathbf{B}_5 & \mathbf{A}_5 & \mathbf{C}_5 & 0 & 0 & 0 & \cdots \\ & & & \ddots & \ddots & \ddots & & & \\ \cdots & 0 & 0 & 0 & \mathbf{B}_{N-4} & \mathbf{A}_{N-4} & \mathbf{C}_{N-4} & 0 & 0 \\ & \cdots & 0 & 0 & 0 & \mathbf{B}_{N-3} & \mathbf{A}_{N-3} & \mathbf{C}_{N-3} & 0 \\ & & \cdots & 0 & 0 & 0 & \mathbf{B}_{N-2} & \mathbf{A}_{N-2} & \mathbf{C}_{N-2} \\ & & & \cdots & 0 & 0 & 0 & \mathbf{B}_{N-1} & \mathbf{A}_{N-1} \end{bmatrix} \begin{bmatrix} \delta\mathbf{u}_2 \\ \delta\mathbf{u}_3 \\ \delta\mathbf{u}_4 \\ \delta\mathbf{u}_5 \\ \vdots \\ \delta\mathbf{u}_{N-4} \\ \delta\mathbf{u}_{N-3} \\ \delta\mathbf{u}_{N-2} \\ \delta\mathbf{u}_{N-1} \end{bmatrix} = \begin{bmatrix} -\boldsymbol{\Xi}_2 \\ -\boldsymbol{\Xi}_3 \\ -\boldsymbol{\Xi}_4 \\ -\boldsymbol{\Xi}_5 \\ \vdots \\ -\boldsymbol{\Xi}_{N-4} \\ -\boldsymbol{\Xi}_{N-3} \\ -\boldsymbol{\Xi}_{N-2} \\ -\boldsymbol{\Xi}_{N-1} \end{bmatrix}, \tag{1.12}$$

with both the system coefficient matrix and right hand sides evaluated at each Newton iteration using $\mathbf{u}^{(l)}$. The first approximation $\mathbf{u}^{(0)}$ for the root $\mathbf{u}^{n+1}$ may be the value $\mathbf{u}^n$ from the previous time level, or the initial value $\mathbf{u}^0$ in the case of the first time step or of stationary problems.

The sizes $N-2 \times N-2$ and $M \times M$ are for this specific case, based on equation (1.8), the simple grid and Dirichlet boundary conditions. In the general case, presented in Section 1.5, the Jacobian will have $N \times N$ blocks.

1.2.1. Convergence Criteria. There is no universal convergence criterion for Newton's method, i.e., no single rule that guarantees solution accuracy while avoiding unnecessary iterations. Convergence criteria are therefore tied to each particular problem and to the discrete method applied to it. The most commonly found criterion is a threefold one. The iteration proceeds until (i) $\max_{i,m} |\boldsymbol{\Xi}_i^{(m)}(\mathbf{u}^{(l)})/\boldsymbol{\Xi}_{\text{ref}}^{(m)}| < \epsilon_1$ or (ii) $\max_{i,m} |\delta\mathbf{u}_i^{(m)}/u_{\text{ref}}^{(m)}| < \epsilon_2$, for some appropriate values of ϵ and some reference values $\mathbf{u}_{\text{ref}}$ and $\boldsymbol{\Xi}_{\text{ref}}$. Notice the notation clashes in the preceeding expressions: the parenthesised superscript index (m) over which the maxima in (i) and (ii) are taken refers to the state component, not to the Newton iteration, which is denoted by superscript index (l). Lastly, the algorithm halts if (iii) a maximum number of iterations has been reached. This last criterion is actually a failsafe in case the method does not converge.

The tolerances ϵ_1 and ϵ_2 are only meaningful in conjunction with the reference values $\boldsymbol{\Xi}_{\text{ref}}$ and $\mathbf{u}_{\text{ref}}$. The first criterion is the most desirable one to satisfy, as it implies convergence to a solution, i.e., the residual is very near zero. The second criterion means convergence only in the sense that the update $\delta\mathbf{u}^{(l)} = \mathbf{u}^{(l+1)} - \mathbf{u}^{(l)}$ is such that the method is not changing the current guess anymore — however, if the method does converge, this implies that $\boldsymbol{\Xi}(\mathbf{u}^{(l)}) \approx \mathbf{0}$.

It is common practice to provide $\mathbf{u}_{\text{ref}}$, but a useful value can be obtained from $u_{\text{ref}}^{(m)} = \max_i |\mathbf{u}_i^{0,(m)}|$, i.e., from the maxima of each state component in the initial data. Likewise, the value of $\boldsymbol{\Xi}_{\text{ref}}$ can be obtained from $\Xi_{\text{ref}}^{(m)} = \max_i |\boldsymbol{\Xi}_i^{(m)}(\mathbf{u}^0)|$.

1.3. A Note on Parallelism

Newton's method, as presented in the previous section, solves a nonlinear system of equations through consecutive linear approximations, i.e., it iteratively builds and solves linear systems. The linear solver is usually where computer parallelism is exploited, since linear solvers are relatively easy to parallelize. Parallel algorithms for the direct solution of linear systems introduce redundant calculations that increase the workload by a factor of about two, depending on the parallelization approach and the number of processors; on the other hand, they produce the same results as serial solvers and scale well.

Linear systems $\mathbf{A}\mathbf{x} = \mathbf{y}$ can be solved in either one or two stages. The one–stage approach has both the matrix $\mathbf{A}$ and the right hand side $\mathbf{y}$ as input data, and provides the solution $\mathbf{x}$ on output; if the system must be solved for a new $\mathbf{y}$, then the whole process may have to be repeated with both $\mathbf{A}$ and this new $\mathbf{y}$. The two–stage approach takes only $\mathbf{A}$ as input in the first stage, and produces a factorization that can be used later for any given $\mathbf{y}$. The second stage, which is computationally much cheaper, uses this factorization and $\mathbf{y}$ to determine the solution. The application we have in view, i.e., the solution of nonlinear systems through Newton's method as given in equation (1.10) makes the one–stage approach the unequivocal choice: each system will have its coefficient matrix and right hand side computed (i.e., available) at the same point of execution, and will be solved only once.

1.4. Reaction–Convection–Diffusion Equation

Many models of interest are governed by equations of the form (1.2)

$$\frac{\partial \mathbf{h}(\mathbf{u})}{\partial t} + \frac{\partial \mathbf{f}(\mathbf{u})}{\partial x} = \frac{\partial}{\partial x}\left(\mathbf{g}(\mathbf{u})\frac{\partial \mathbf{u}}{\partial x}\right) + \mathbf{q}(\mathbf{u});$$

this is the general convection–diffusion–reaction equation for incompressible flows, where $\mathbf{u} \in \mathbb{R}^M$ is the state vector, $\mathbf{f}, \mathbf{h}, \mathbf{q} : \mathbb{R}^M \to \mathbb{R}^M$ and $\mathbf{g} : \mathbb{R}^M \to \mathbb{R}^{M\times M}$ are differentiable, the physical domain is $x \in [x_a, x_b]$ and $t \geq 0$. The differential operator $\widetilde{\boldsymbol{\mathcal{E}}}$ corresponding to (1.2) is

$$\widetilde{\boldsymbol{\mathcal{E}}}(\mathbf{u}) \equiv \frac{\partial \mathbf{h}(\mathbf{u})}{\partial t} + \frac{\partial \mathbf{f}(\mathbf{u})}{\partial x} - \frac{\partial}{\partial x}\left(\mathbf{g}(\mathbf{u})\frac{\partial \mathbf{u}}{\partial x}\right) - \mathbf{q}(\mathbf{u}). \tag{1.13}$$

The four terms of (1.2) correspond to accumulation, convection, diffusion and reaction, from left to right; however, it is not the case that all four terms are present in every model. Notice also that $\widetilde{\boldsymbol{\mathcal{E}}} : \mathbb{R}^M \to \mathbb{R}^M$, and therefore some terms of (1.2) may be present in some — but not all — of the M scalar equations $\widetilde{\mathcal{E}}$ represented by $\widetilde{\boldsymbol{\mathcal{E}}}$. Because of that, it may be required to provide different discretizations $\mathcal{E}$ for each equation $\widetilde{\mathcal{E}}$.

Models that do not include the pressure and the velocity in the state vector can be based on this equation and discretized over the simple grid described in Subsection 1.1.1. There are models for compressible flows based on this equation, but they employ simplifications so

that the pressure or the velocity are not part of the state vector, and are therefore treated as if the flow was incompressible.

1.4.1. Crank-Nicolson Scheme. We present a scheme which generalizes the first Crank-Nicolson scheme [**11**]; it is sufficient for most cases and has advantages such as quadratic convergence in time and space. This discretization is valid if $\mathbf{g}$ is non–zero, although it may be constant. The other three functions $\mathbf{h}$, $\mathbf{f}$ and $\mathbf{q}$ may be identically zero.

Following the previously introduced notation $\mathbf{u}(x_i, t^n) \equiv \mathbf{u}_i^n$, we denote $\mathbf{h}(\mathbf{u}(x_i, t^n)) = \mathbf{h}_i^n$, $\mathbf{f}(\mathbf{u}(x_i, t^n)) = \mathbf{f}_i^n$, $\mathbf{g}(\mathbf{u}(x_i, t^n)) = \mathbf{g}_i^n$ and $\mathbf{q}(\mathbf{u}(x_i, t^n)) = \mathbf{q}_i^n$. Using central differences and averages in time, i.e,

$$\frac{\partial}{\partial t}\mathbf{h}^{n+1/2} = \frac{\mathbf{h}^{n+1} - \mathbf{h}^n}{\delta t} + \mathcal{O}(\delta t^2) \quad \text{and} \quad \mathbf{f}^{n+\alpha} = \alpha \mathbf{f}^{n+1} + (1-\alpha)\mathbf{f}^n + \mathcal{O}\left(\left(\alpha - \frac{1}{2}\right)\delta t + \delta t^2\right), \tag{1.14}$$

equation (1.2) is approximated in time by

$$\begin{aligned} &\frac{\mathbf{h}^{n+1} - \mathbf{h}^n}{\delta t} + \left[\alpha\left(\frac{\partial \mathbf{f}(\mathbf{u})}{\partial x}\right)^{n+1} + (1-\alpha)\left(\frac{\partial \mathbf{f}(\mathbf{u})}{\partial x}\right)^{n}\right] = \\ &\frac{\partial}{\partial x}\left[\alpha\left(\mathbf{g}(\mathbf{u})\frac{\partial \mathbf{u}}{\partial x}\right)^{n+1} + (1-\alpha)\left(\mathbf{g}(\mathbf{u})\frac{\partial \mathbf{u}}{\partial x}\right)^{n}\right] + \mathbf{q}^{n+\alpha} + \mathcal{O}\left(\left(\alpha - \frac{1}{2}\right)\delta t + \delta t^2\right). \end{aligned} \tag{1.15}$$

If the weighing factor $\alpha \in [0, 1]$ is $1/2$, we obtain the Crank-Nicolson scheme, which is of second order in time. Other values $\alpha > 1/2$ yield similar schemes which are more diffusive and more stable, but of first order in time.

Applying the central differences operators for the first derivative in space, i.e.,

$$\frac{\partial}{\partial x}\mathbf{f}_i = \frac{\mathbf{f}_{i+1} - \mathbf{f}_{i-1}}{2\delta x} + \mathcal{O}(\delta x^2)$$

to the time discrete equation (1.15) yields, for the left hand side,

$$\frac{\mathbf{h}_i^{n+1} - \mathbf{h}_i^n}{\delta t} + \frac{\alpha\left(\mathbf{f}_{i+1}^{n+1} - \mathbf{f}_{i-1}^{n+1}\right) + (1-\alpha)\left(\mathbf{f}_{i+1}^{n} - \mathbf{f}_{i-1}^{n}\right)}{2\delta x} + \mathcal{O}(\delta x^2); \tag{1.16}$$

for the right hand side, we use virtual middle points of the discrete x grid, denoting their indices by $i \pm 1/2$, and obtain

$$\begin{aligned} \frac{\partial}{\partial x}\left(\mathbf{g}(\mathbf{u})\frac{\partial \mathbf{u}}{\partial x}\right) &= \frac{1}{\delta x}\left(\mathbf{g}_{i+1/2}\frac{\partial \mathbf{u}}{\partial x}\bigg|_{x=x_{i+1/2}} - \mathbf{g}_{i-1/2}\frac{\partial \mathbf{u}}{\partial x}\bigg|_{x=x_{i-1/2}}\right) + \mathcal{O}(\delta x^2) \\ &= \frac{1}{\delta x}\left(\mathbf{g}_{i+1/2}\frac{\mathbf{u}_{i+1} - \mathbf{u}_i}{\delta x} - \mathbf{g}_{i-1/2}\frac{\mathbf{u}_i - \mathbf{u}_{i-1}}{\delta x}\right) + \mathcal{O}(\delta x^2), \end{aligned} \tag{1.17}$$

where the value of $\mathbf{g}_{i\pm1/2}$ can be approximated by

$$\mathbf{g}_{i\pm1/2} = \frac{\mathbf{g}_{i\pm1} + \mathbf{g}_i}{2} + \mathcal{O}(\delta x^2),$$

so the final form of expression (1.17) is

$$\begin{aligned}\frac{\partial}{\partial x}\left(\mathbf{g}(\mathbf{u})\frac{\partial \mathbf{u}}{\partial x}\right) &= \frac{1}{\delta x^2}\left(\frac{\mathbf{g}_{i+1}+\mathbf{g}_i}{2}(\mathbf{u}_{i+1}-\mathbf{u}_i)-\frac{\mathbf{g}_{i-1}+\mathbf{g}_i}{2}(\mathbf{u}_i-\mathbf{u}_{i-1})\right)+\mathcal{O}(\delta x^2)\\ &= \frac{1}{2\delta x^2}\big((\mathbf{g}_{i+1}+\mathbf{g}_i)\mathbf{u}_{i+1}-(\mathbf{g}_{i+1}+2\mathbf{g}_i+\mathbf{g}_{i-1})\mathbf{u}_i+(\mathbf{g}_i+\mathbf{g}_{i-1})\mathbf{u}_{i-1}\big)+\mathcal{O}(\delta x^2).\end{aligned} \tag{1.18}$$

Putting together expressions (1.14) through (1.18), we reach the fully discrete equation

$$\begin{aligned}&\frac{\mathbf{h}_i^{n+1}-\mathbf{h}_i^n}{\delta t}+\frac{\alpha\big(\mathbf{f}_{i+1}^{n+1}-\mathbf{f}_{i-1}^{n+1}\big)+(1-\alpha)\big(\mathbf{f}_{i+1}^{n}-\mathbf{f}_{i-1}^{n}\big)}{2\delta x}=\\ &\quad\frac{1}{2\delta x^2}\Bigg(\alpha\big((\mathbf{g}_{i+1}^{n+1}+\mathbf{g}_i^{n+1})\mathbf{u}_{i+1}^{n+1}-(\mathbf{g}_{i+1}^{n+1}+2\mathbf{g}_i^{n+1}+\mathbf{g}_{i-1}^{n+1})\mathbf{u}_i^{n+1}+(\mathbf{g}_i^{n+1}+\mathbf{g}_{i-1}^{n+1})\mathbf{u}_{i-1}^{n+1}\big)+\\ &\qquad(1-\alpha)\big((\mathbf{g}_{i+1}^{n}+\mathbf{g}_i^{n})\mathbf{u}_{i+1}^{n}-(\mathbf{g}_{i+1}^{n}+2\mathbf{g}_i^{n}+\mathbf{g}_{i-1}^{n})\mathbf{u}_i^{n}+(\mathbf{g}_i^{n}+\mathbf{g}_{i-1}^{n})\mathbf{u}_{i-1}^{n}\big)\Bigg)+\\ &\quad+\alpha\mathbf{q}_i^{n+1}+(1-\alpha)\mathbf{q}_i^n+\mathcal{O}\left(\left(\alpha-\frac{1}{2}\right)\delta t+\delta t^2+\delta x^2\right).\end{aligned} \tag{1.19}$$

From equation (1.19) we obtain a numerical scheme for integrating in time: separating the terms involving the unknowns at time $n+1$ on the left hand side, we obtain

$$\begin{aligned}&\frac{\mathbf{h}_i^{n+1}}{\delta t}+\alpha\frac{\mathbf{f}_{i+1}^{n+1}-\mathbf{f}_{i-1}^{n+1}}{2\delta x}-\\ &\alpha\left(\frac{(\mathbf{g}_{i+1}^{n+1}+\mathbf{g}_i^{n+1})\mathbf{u}_{i+1}^{n+1}-(\mathbf{g}_{i+1}^{n+1}+2\mathbf{g}_i^{n+1}+\mathbf{g}_{i-1}^{n+1})\mathbf{u}_i^{n+1}+(\mathbf{g}_i^{n+1}+\mathbf{g}_{i-1}^{n+1})\mathbf{u}_{i-1}^{n+1}}{2\delta x^2}\right)-\alpha\mathbf{q}_i^{n+1}=\end{aligned} \tag{1.20}$$

$$\begin{aligned}&\frac{\mathbf{h}_i^{n}}{\delta t}-(1-\alpha)\frac{\mathbf{f}_{i+1}^{n}-\mathbf{f}_{i-1}^{n}}{2\delta x}\\ &+(1-\alpha)\left(\frac{(\mathbf{g}_{i+1}^{n}+\mathbf{g}_i^{n})\mathbf{u}_{i+1}^{n}-(\mathbf{g}_{i+1}^{n}+2\mathbf{g}_i^{n}+\mathbf{g}_{i-1}^{n})\mathbf{u}_i^{n}+(\mathbf{g}_i^{n}+\mathbf{g}_{i-1}^{n})\mathbf{u}_{i-1}^{n}}{2\delta x^2}\right)+(1-\alpha)\mathbf{q}_i^n,\end{aligned} \tag{1.21}$$

with an error of the order of $\mathcal{O}\big((\alpha-1/2)\delta t+\delta t^2+\delta x^2\big)$. If we put the right hand side (1.21) to the left, we obtain the complete discrete operator $\boldsymbol{\mathcal{E}}$ corresponding to (1.13). Differentiation of the left hand side (1.20) gives the formulæ for the expressions in (1.11):

$$\begin{aligned}\mathbf{B}_i &= \frac{\partial \boldsymbol{\Xi}_i}{\partial \mathbf{u}_{i-1}} &&= -\alpha\left(\frac{\partial \mathbf{f}_{i-1}}{2\delta x}+\frac{\partial \mathbf{g}_{i-1}(\mathbf{u}_{i-1}-\mathbf{u}_i)+\mathbf{g}_i+\mathbf{g}_{i-1}}{2\delta x^2}\right);\\ \mathbf{A}_i &= \frac{\partial \boldsymbol{\Xi}_i}{\partial \mathbf{u}_i} &&= \frac{\partial \mathbf{h}_i}{\delta t}-\alpha\left(\frac{\partial \mathbf{g}_i(\mathbf{u}_{i+1}-2\mathbf{u}_i+\mathbf{u}_{i-1})-\mathbf{g}_{i+1}-2\mathbf{g}_i-\mathbf{g}_{i-1}}{2\delta x^2}+\partial \mathbf{q}_i\right);\\ \mathbf{C}_i &= \frac{\partial \boldsymbol{\Xi}_i}{\partial \mathbf{u}_{i+1}} &&= \alpha\left(\frac{\partial \mathbf{f}_{i+1}}{2\delta x}-\frac{\partial \mathbf{g}_{i+1}(\mathbf{u}_{i+1}-\mathbf{u}_i)+\mathbf{g}_{i+1}+\mathbf{g}_i}{2\delta x^2}\right).\end{aligned} \tag{1.22}$$

1.4.2. Box Scheme. In the event of there being an equation in $\widetilde{\boldsymbol{\mathcal{E}}}$ for which $\mathbf{g} \equiv 0$, we apply the box scheme. It uses central differences in time in the same manner as Crank-Nicolson, but instead of using central differences in space around x_i, it does so around $x_{i+1/2}$ — or, alternatively, uses forward differences. Whatever interpretation is given, the derivation is simple. From the continuous equation

$$\frac{\partial \mathbf{h}}{\partial t}(\mathbf{u}) + \frac{\partial \mathbf{f}}{\partial x}(\mathbf{u}) = \mathbf{q}(\mathbf{u}),$$

using the same time discretization as in (1.14) yields

$$\frac{\mathbf{h}^{n+1} - \mathbf{h}^n}{\delta t} + \left[\alpha\left(\frac{\partial \mathbf{f}(\mathbf{u})}{\partial x}\right)^{n+1} + (1-\alpha)\left(\frac{\partial \mathbf{f}(\mathbf{u})}{\partial x}\right)^{n}\right] = \mathbf{q}^{n+\alpha} + \mathcal{O}\left(\left(\alpha - \frac{1}{2}\right)\delta t + \delta t^2\right),$$

to which we apply

$$\frac{\partial}{\partial x}\mathbf{f}_i = \frac{\mathbf{f}_{i+1} - \mathbf{f}_i}{\delta x} + \mathcal{O}(\delta x^2).$$

and averages in space analogous to (1.14). Separating the terms at $n+1$ on the left hand side, we arrive at

$$\begin{aligned}\frac{\mathbf{h}_{i+1}^{n+1} + \mathbf{h}_i^{n+1}}{2\delta t} + \alpha\left(\frac{\mathbf{f}_{i+1}^{n+1} - \mathbf{f}_i^{n+1}}{\delta x} - \frac{\mathbf{q}_{i+1}^{n+1} + \mathbf{q}_i^{n+1}}{2}\right) = \\ \frac{\mathbf{h}_{i+1}^{n} + \mathbf{h}_i^{n}}{2\delta t} - (1-\alpha)\left(\frac{\mathbf{f}_{i+1}^{n} - \mathbf{f}_i^{n}}{\delta x} - \frac{\mathbf{q}_{i+1}^{n} + \mathbf{q}_i^{n}}{2}\right),\end{aligned} \tag{1.23}$$

with an error of the order of $\mathcal{O}\big((\alpha - 1/2)\delta t + \delta t^2 + \delta x^2\big)$. Again, if we put all terms in the left hand side, we obtain the complete discrete operator $\boldsymbol{\mathcal{E}}$ corresponding to (1.13). Differentiation of the left hand side of (1.23) gives the formulæ for the expressions in (1.11):

$$\begin{aligned}\mathbf{B}_i &= \frac{\partial \boldsymbol{\Xi}_i}{\partial \mathbf{u}_{i-1}} = 0; \\ \mathbf{A}_i &= \frac{\partial \boldsymbol{\Xi}_i}{\partial \mathbf{u}_i} = \frac{\partial \mathbf{h}_i}{2\delta t} - \alpha\left(\frac{\partial \mathbf{f}_i}{\delta x} + \frac{\partial \mathbf{q}_i}{2}\right); \\ \mathbf{C}_i &= \frac{\partial \boldsymbol{\Xi}_i}{\partial \mathbf{u}_{i+1}} = \frac{\partial \mathbf{h}_{i+1}}{2\delta t} + \alpha\left(\frac{\partial \mathbf{f}_{i+1}}{\delta x} - \frac{\partial \mathbf{q}_{i+1}}{2}\right).\end{aligned} \tag{1.24}$$

1.5. Boundary Conditions for the Single Grid

In Section 1.2, we presented the complete solution procedure for the problem stated in equations (1.8) and (1.9), assuming Dirichlet boundary conditions, i.e., that the values of $\mathbf{u}_1 = \boldsymbol{\gamma}_a$ and $\mathbf{u}_N = \boldsymbol{\gamma}_b$ are given, so they can be excluded from the system presented in equation (1.8), which we solve only for the $N-2$ inner points of the physical domain.

Equation (1.8) was linearized as part of Newton's method, i.e., the solution of the nonlinear system (1.8) is found solving several linear systems (1.12). In order to introduce other boundary conditions, we first create an augmented version of (1.12). The system

$$\left[\begin{array}{c|ccccccc} \mathbf{I} & 0 & 0 & 0 & 0 & \cdots & \\ \hline \mathbf{B}_2 & \mathbf{A}_2 & \mathbf{C}_2 & 0 & 0 & 0 & \cdots \\ 0 & \mathbf{B}_3 & \mathbf{A}_3 & \mathbf{C}_3 & 0 & 0 & 0 \\ & & \ddots & \ddots & \ddots & & \\ 0 & 0 & 0 & \mathbf{B}_{N-2} & \mathbf{A}_{N-2} & \mathbf{C}_{N-2} & 0 \\ \cdots & 0 & 0 & 0 & \mathbf{B}_{N-1} & \mathbf{A}_{N-1} & \mathbf{C}_{N-1} \\ \hline & \cdots & 0 & 0 & 0 & 0 & \mathbf{I} \end{array}\right] \left[\begin{array}{c} \delta\mathbf{u}_1 \\ \hline \delta\mathbf{u}_2 \\ \delta\mathbf{u}_3 \\ \vdots \\ \delta\mathbf{u}_{N-2} \\ \delta\mathbf{u}_{N-1} \\ \hline \delta\mathbf{u}_N \end{array}\right] = \left[\begin{array}{c} \mathbf{0} \\ \hline -\boldsymbol{\Xi}_2 \\ -\boldsymbol{\Xi}_3 \\ \vdots \\ -\boldsymbol{\Xi}_{N-2} \\ -\boldsymbol{\Xi}_{N-1} \\ \hline \mathbf{0} \end{array}\right] \tag{1.25}$$

is completely equivalent to the one in (1.12): the two unnecessary lines — $\mathbf{I}\,\delta\mathbf{u}_i = \mathbf{0}$ for $i \in \{1, N\}$ — imply $\delta\mathbf{u}_i = 0$, i.e., the values of $\mathbf{u}_i$ do not change at the boundaries. However, adopting this formulation makes the solution procedure compatible with other boundary conditions, for which $\mathbf{u}_1$ and $\mathbf{u}_N$ do vary.

If the boundary conditions are given in terms of $\partial\mathbf{u}/\partial x$ or by a nonlinear equation in $\mathbf{u}$, the values of the $\mathbf{u}_1$ and $\mathbf{u}_N$ become unknowns. This increases the number of equations and unknowns to N: equations $\boldsymbol{\Xi}_1 \equiv \boldsymbol{\mathcal{E}}(\mathbf{u}_1, \mathbf{u}_2)$ and $\boldsymbol{\Xi}_N \equiv \boldsymbol{\mathcal{E}}(\mathbf{u}_{N-1}, \mathbf{u}_N)$ must be obtained from the discretization of the expression for the boundary condition. The corresponding modifications transform system (1.25) into

$$\left[\begin{array}{c|ccccccc} \mathbf{A}_1 & \mathbf{C}_1 & 0 & 0 & 0 & \cdots & \\ \hline \mathbf{B}_2 & \mathbf{A}_2 & \mathbf{C}_2 & 0 & 0 & 0 & \cdots \\ 0 & \mathbf{B}_3 & \mathbf{A}_3 & \mathbf{C}_3 & 0 & 0 & 0 \\ & & \ddots & \ddots & \ddots & & \\ 0 & 0 & 0 & \mathbf{B}_{N-2} & \mathbf{A}_{N-2} & \mathbf{C}_{N-2} & 0 \\ \cdots & 0 & 0 & 0 & \mathbf{B}_{N-1} & \mathbf{A}_{N-1} & \mathbf{C}_{N-1} \\ \hline & \cdots & 0 & 0 & 0 & \mathbf{B}_N & \mathbf{A}_N \end{array}\right] \left[\begin{array}{c} \delta\mathbf{u}_1 \\ \hline \delta\mathbf{u}_2 \\ \delta\mathbf{u}_3 \\ \vdots \\ \delta\mathbf{u}_{N-2} \\ \delta\mathbf{u}_{N-1} \\ \hline \delta\mathbf{u}_N \end{array}\right] = \left[\begin{array}{c} -\boldsymbol{\Xi}_1 \\ \hline -\boldsymbol{\Xi}_2 \\ -\boldsymbol{\Xi}_3 \\ \vdots \\ -\boldsymbol{\Xi}_{N-2} \\ -\boldsymbol{\Xi}_{N-1} \\ \hline -\boldsymbol{\Xi}_N \end{array}\right], \tag{1.26}$$

where the expressions for lines $i \in \{1, N\}$ are given in the subsections that follow for the two other types of boundary conditions we consider, namely the Neumann condition and the Robin condition.

So far we have assumed that the same type of boundary condition is imposed on all M state variables at both boundaries, which was appropriate and sufficient since we were considering only the Dirichlet condition. In general, one may impose a different type of boundary condition on each of the M scalar equations in $\widetilde{\boldsymbol{\mathcal{E}}} : \mathbb{R}^M \to \mathbb{R}^M$ (equation (1.1)); one may just as well impose a different condition for the same equation at either boundary. This amounts to an implementation problem only: all the formulæ presented hold for the scalar equations just as they hold for the block equations.

1.5.1. Neumann Boundary Condition. Instead of prescribing the value of $\mathbf{u}_1 \equiv \mathbf{u}_a$ at the left end, we prescribe

$$\frac{\partial \mathbf{u}}{\partial x}(x_a, t) = \boldsymbol{\gamma}_a(t). \tag{1.27}$$

If we apply forward differences, we obtain

$$\frac{\mathbf{u}_2 - \mathbf{u}_1}{\delta x} = \boldsymbol{\gamma}_a(t) \Rightarrow \mathbf{u}_2 = \mathbf{u}_1 + \boldsymbol{\gamma}_a(t)\delta x, \tag{1.28}$$

which gives an equation for $\boldsymbol{\Xi}_1(\mathbf{u}_1, \mathbf{u}_2)$, namely

$$\boldsymbol{\Xi}_1(\mathbf{u}_1, \mathbf{u}_2) = \mathbf{u}_1 - \mathbf{u}_2 + \boldsymbol{\gamma}_a(t^{n+1})\delta x, \tag{1.29}$$

so $\mathbf{A}_1 = \mathbf{I}$ and $\mathbf{C}_1 = -\mathbf{I}$.

The expressions at the right end of the domain are analogous: backward differences between x_{N-1} and x_N are used to discretize the value of

$$\frac{\partial \mathbf{u}}{\partial x}(x_b, t) = \boldsymbol{\gamma}_b(t) \approx \frac{\mathbf{u}_N - \mathbf{u}_{N-1}}{\delta x} = \boldsymbol{\gamma}_b(t^{n+1}) \Rightarrow \mathbf{u}_N = \mathbf{u}_{N-1} + \boldsymbol{\gamma}_b(t^{n+1})\delta x \tag{1.30}$$

yielding

$$\boldsymbol{\Xi}_N(\mathbf{u}_{N-1}, \mathbf{u}_N) = \mathbf{u}_N - \mathbf{u}_{N-1} - \boldsymbol{\gamma}_b(t^{n+1})\delta x \tag{1.31}$$

and $\mathbf{A}_N = \mathbf{I}$ and $\mathbf{B}_N = -\mathbf{I}$.

1.5.2. Robin Boundary Condition. The treatment of a prescribed value for a linear combination $\mathbf{u}_a$ and $\partial \mathbf{u}_a / \partial x$, i.e, at the left end

$$\alpha \mathbf{u}(x_a, t) + \beta \frac{\partial \mathbf{u}}{\partial x}(x_a, t) = \boldsymbol{\gamma}_a(t), \tag{1.32}$$

is very similar to the Neumann case. The use of forward differences gives an equation for $\boldsymbol{\Xi}_1(\mathbf{u}_1, \mathbf{u}_2)$, namely

$$\alpha \mathbf{u}_1 + \beta \frac{\mathbf{u}_2 - \mathbf{u}_1}{\delta x} = \boldsymbol{\gamma}_a(t) \Rightarrow \boldsymbol{\Xi}_1(\mathbf{u}_1, \mathbf{u}_2) = (\alpha \delta x - \beta)\mathbf{u}_1 + \beta \mathbf{u}_2 - \boldsymbol{\gamma}_a(t)\delta x, \tag{1.33}$$

so $\mathbf{A}_1 = (\alpha \delta x - \beta)\mathbf{I}$ and $\mathbf{C}_1 = \beta \mathbf{I}$.

At the right end, the use of backward differences leads to an equation for $\boldsymbol{\Xi}_N(\mathbf{u}_{N-1}, \mathbf{u}_N)$:

$$\alpha \mathbf{u}_N + \beta \frac{\mathbf{u}_N - \mathbf{u}_{N-1}}{\delta x} = \boldsymbol{\gamma}_b(t) \Rightarrow \boldsymbol{\Xi}_N(\mathbf{u}_{N-1}, \mathbf{u}_N) = (\alpha \delta x + \beta)\mathbf{u}_N - \beta \mathbf{u}_{N-1} - \boldsymbol{\gamma}_b(t)\delta x, \tag{1.34}$$

so $\mathbf{A}_N = (\alpha \delta x + \beta)\mathbf{I}$ and $\mathbf{B}_N = -\beta \mathbf{I}$

1.6. General Reaction–Convection–Diffusion Equation for Compressible Flows

At the time of this draft, this discretization was fully implemented, but not tested.

Models in which the coupling of the p and v fields over x is taken into account are governed by equations of the form (1.3)

$$\frac{\partial \mathbf{h}(\mathbf{u})}{\partial t} + \frac{\partial v\mathbf{f}(\mathbf{u})}{\partial x} = \frac{\partial}{\partial x}\left(\mathbf{g}(\mathbf{u})\frac{\partial \mathbf{u}}{\partial x}\right) + \mathbf{q}(\mathbf{u}),$$

where the pressure $p \equiv u^{(M)}$ is coupled to the velocity v through Darcy's Law (1.4),

$$v = -k(\mathbf{u})\frac{\partial p}{\partial x} + r(\mathbf{u}).$$

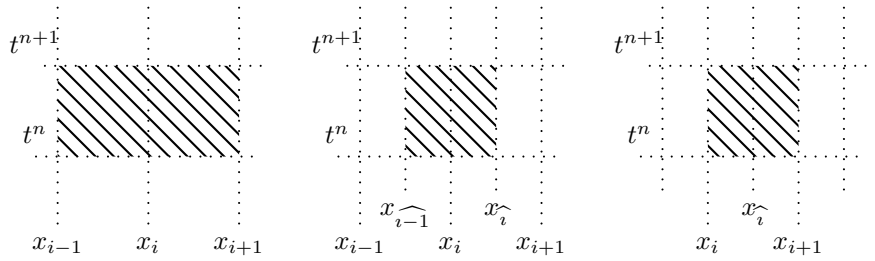
Equation (1.3) is the general convection–diffusion–reaction equation for compressible flows, where $\{\mathbf{u}; v\}$ with $\mathbf{u} \in \mathbb{R}^M$ is the state vector, $\mathbf{f}, \mathbf{h}, \mathbf{q} : \mathbb{R}^M \to \mathbb{R}^M$ and $\mathbf{g} : \mathbb{R}^M \to \mathbb{R}^{M\times M}$ are differentiable, the physical domain is $x \in [x_a, x_b]$ and $t \geq 0$. The differential operator $\widetilde{\boldsymbol{\mathcal{E}}} : \mathbb{R}^{M+1} \to \mathbb{R}^{M+1}$ corresponding to (1.3) and Darcy's Law is

$$\begin{cases} \widetilde{\mathcal{E}}^{(m)}(\mathbf{u}; v) \equiv \dfrac{\partial h^{(m)}(\mathbf{u})}{\partial t} + \dfrac{\partial v\, f^{(m)}(\mathbf{u})}{\partial x} - \dfrac{\partial}{\partial x}\left(g^{(m)}(\mathbf{u})\dfrac{\partial \mathbf{u}}{\partial x}\right) - q^{(m)}(\mathbf{u}), \quad m = 1, \ldots, M \\ \widetilde{\mathcal{E}}^{(M+1)}(\mathbf{u}; v) \equiv v + k(\mathbf{u})\dfrac{\partial p}{\partial x} - r(\mathbf{u}), \quad \text{with } p \in \mathbf{u} \end{cases}, \tag{1.35}$$

and the observations made for the simplified version of the equation on page 2 still apply.

1.6.1. Control Volume Discretization. The scheme derived in 1.4.1 was obtained using the traditional approach of substituting the continuous differential operators with finite difference approximations centered on each point x_i. The same scheme can be derived integrating equation (1.2) in time and space over the control volume $x \in [x_{i-1}, x_{i+1}]$, $t \in [t^n, t^{n+1}]$, depicted in Figure 1.3a. That scheme is valid if $\mathbf{g}$ is not null: it becomes less reliable the more the convection term dominantes the diffusive term.

In this section we derive the scheme using integration in the control volume shown in Figure 1.3b for the convection term, and the control volume depicted in Figure 1.3c for Darcy's Law (1.4). Only the discretization of the convection term and Darcy's Law are derived in this subsection: the discrete versions of the other terms in equation (1.3) are identical to the corresponding terms in (1.2). This scheme gives better results for problems in which convection dominates diffusion: it is valid even if $\mathbf{g}$ is null. This is because the discrete expression of the convection term, although centered and symmetric, takes all three spatial nodes of the stencil for x_i into account, whereas the expressions used in Section 1.4 used only two.



(a) (b) (c)

FIGURE 1.3. Control volumes used for integrating terms of equation (1.3). The control volume in (a) is used to integrate terms that do not involve the velocity, the reduced control volume in (b) is used to integrate the convection term, and the control volume in (c), centered at $x_{\widehat{i}}$ instead of x_i, is used for Darcy's law.

For the convection term, we wish to find the integral of $\partial(v\mathbf{f})/\partial x$ over the $\delta x \times \delta t$ volume in Figure 1.3b,

$$\int_{x_{\widehat{i-1}}}^{x_{\widehat{i}}} \int_{t^n}^{t^{n+1}} \frac{\partial(v\mathbf{f}(\mathbf{u}))}{\partial x}\, dt\, dx.$$

There is no time derivative involved, so we take a wheighted average just like in (1.14) and (1.15); it remains to integrate

$$\delta t \int_{x_{\widehat{i-1}}}^{x_{\widehat{i}}} \alpha \frac{\partial(v\mathbf{f}(\mathbf{u}))}{\partial x}\bigg|_{t^{n+1}} + (1-\alpha)\frac{\partial(v\mathbf{f}(\mathbf{u}))}{\partial x}\bigg|_{t^n} dx.$$

This integral represents the difference between the fluxes going through the two boundaries of the volume depicted in Figure 1.3b, i.e.,

$$\int_{x_{\widehat{i-1}}}^{x_{\widehat{i}}} \frac{\partial(v\mathbf{f}(\mathbf{u}))}{\partial x}\, dx = v\mathbf{f}(\mathbf{u})\bigg|_{x_{\widehat{i}}} - v\mathbf{f}(\mathbf{u})\bigg|_{x_{\widehat{i-1}}}.$$

The values of v are defined at the secondary locations $x_{\widehat{i}}$ and $x_{\widehat{i-1}}$, but the values of $\mathbf{u}$ (and therefore of $\mathbf{f}(\mathbf{u})$) are not, so we take averages:

$$v\mathbf{f}(\mathbf{u})\bigg|_{x_{\widehat{i}}} = v_{\widehat{i}}\frac{\mathbf{f}_i + \mathbf{f}_{i+1}}{2}.$$

Putting all the preceding expressions together, we derive

$$\begin{aligned}\int_{x_{\widehat{i-1}}}^{x_{\widehat{i}}}\int_{t^n}^{t^{n+1}} \frac{\partial(v\mathbf{f}(\mathbf{u}))}{\partial x}\,dt\,dx &= \delta t\int_{x_{\widehat{i-1}}}^{x_{\widehat{i}}} \alpha\frac{\partial(v\mathbf{f}(\mathbf{u}))}{\partial x}\bigg|_{t^{n+1}} + (1-\alpha)\frac{\partial(v\mathbf{f}(\mathbf{u}))}{\partial x}\bigg|_{t^n} dx \\ &= \frac{\delta t}{2}\bigg(\alpha\big(v_{\widehat{i}}^{n+1}(\mathbf{f}_i^{n+1}+\mathbf{f}_{i+1}^{n+1}) - v_{\widehat{i-1}}^{n+1}(\mathbf{f}_{i-1}^{n+1}+\mathbf{f}_i^{n+1})\big)+ \\ &\qquad (1-\alpha)\big(v_{\widehat{i}}^{n}(\mathbf{f}_i^{n}+\mathbf{f}_{i+1}^{n}) - v_{\widehat{i-1}}^{n}(\mathbf{f}_{i-1}^{n}+\mathbf{f}_i^{n})\big)\bigg).\end{aligned} \tag{1.36}$$

Dividing this by $\delta x \times \delta t$ — the area of the control volume in Figure 1.3b — we arrive at the expression equivalent to (1.16) on the simple grid:

$$\begin{aligned}\frac{\mathbf{h}_i^{n+1}-\mathbf{h}_i^n}{\delta t} &+\frac{1}{2\delta x}\bigg(\alpha\big(v_{\widehat{i}}^{n+1}(\mathbf{f}_i^{n+1}+\mathbf{f}_{i+1}^{n+1}) - v_{\widehat{i-1}}^{n+1}(\mathbf{f}_{i-1}^{n+1}+\mathbf{f}_i^{n+1})\big)+ \\ &\qquad (1-\alpha)\big(v_{\widehat{i}}^{n}(\mathbf{f}_i^{n}+\mathbf{f}_{i+1}^{n}) - v_{\widehat{i-1}}^{n}(\mathbf{f}_{i-1}^{n}+\mathbf{f}_i^{n})\big)\bigg) + \mathcal{O}(\delta x^2).\end{aligned} \tag{1.37}$$

Using (1.37) instead of (1.16) gives the full expressions corresponding to (1.19) (see page 9), which we skip in this derivation for begin too lengthy. The equation used in the numerical method, corresponding to the expression (1.20)–(1.21), is

$$\begin{aligned}&\frac{\mathbf{h}_i^{n+1}}{\delta t} + \frac{\alpha}{2\delta x}\big(v_{\widehat{i}}^{n+1}(\mathbf{f}_i^{n+1}+\mathbf{f}_{i+1}^{n+1}) - v_{\widehat{i-1}}^{n+1}(\mathbf{f}_{i-1}^{n+1}+\mathbf{f}_i^{n+1})\big)- \\ &\alpha\bigg(\frac{(\mathbf{g}_{i+1}^{n+1}+\mathbf{g}_i^{n+1})\mathbf{u}_{i+1}^{n+1} - (\mathbf{g}_{i+1}^{n+1}+2\mathbf{g}_i^{n+1}+\mathbf{g}_{i-1}^{n+1})\mathbf{u}_i^{n+1} + (\mathbf{g}_i^{n+1}+\mathbf{g}_{i-1}^{n+1})\mathbf{u}_{i-1}^{n+1}}{2\delta x^2}\bigg) - \alpha\mathbf{q}_i^{n+1} =\end{aligned} \tag{1.38}$$

$$\begin{aligned}&\frac{\mathbf{h}_i^{n}}{\delta t} - \frac{(1-\alpha)}{2\delta x}\big(v_{\widehat{i}}^{n}(\mathbf{f}_i^{n}+\mathbf{f}_{i+1}^{n}) - v_{\widehat{i-1}}^{n}(\mathbf{f}_{i-1}^{n}+\mathbf{f}_i^{n})\big) \\ &+(1-\alpha)\bigg(\frac{(\mathbf{g}_{i+1}^{n}+\mathbf{g}_i^{n})\mathbf{u}_{i+1}^{n} - (\mathbf{g}_{i+1}^{n}+2\mathbf{g}_i^{n}+\mathbf{g}_{i-1}^{n})\mathbf{u}_i^{n} + (\mathbf{g}_i^{n}+\mathbf{g}_{i-1}^{n})\mathbf{u}_{i-1}^{n}}{2\delta x^2}\bigg) + (1-\alpha)\mathbf{q}_i^{n}.\end{aligned} \tag{1.39}$$

Differentiation of the left hand side (1.38) produces the first M lines of each block of the Jacobian of $\boldsymbol{\Xi}$: the last line is obtained from the discretization of the operator $\Xi^{(M+1)} \equiv \widetilde{\mathcal{E}}^{(M+1)}$ in (1.35). We discretize Darcy's Law using the control volume depicted in Figure 1.3c, centered in space at $x_{\widehat{i}}$. Taking averages in time and applying central differences in space to p, we derive from $\mathcal{E}^{(M+1)} = 0$

$$\alpha\bigg(v_{\widehat{i}}^{n+1} - \frac{k_{\widehat{i}}^{n+1}}{\delta x}(p_{i+1}^{n+1}-p_i^{n+1}) + r_{\widehat{i}}^{n+1}\bigg) = -(1-\alpha)\bigg(v_{\widehat{i}}^{n} - \frac{k_{\widehat{i}}^{n}}{\delta x}(p_{i+1}^{n}-p_i^{n}) + r_{\widehat{i}}^{n}\bigg), \tag{1.40}$$

and averages in space for k and r yield

$$\begin{aligned}\alpha\left(v_{\widehat{i}}^{n+1} - \frac{(k_i^{n+1}+k_{i+1}^{n+1})(p_{i+1}^{n+1}-p_i^{n+1})}{2\delta x} + \frac{r_i^{n+1}+r_{i+1}^{n+1}}{2}\right) = \\ -(1-\alpha)\left(v_{\widehat{i}}^{n} - \frac{(k_i^{n}+k_{i+1}^{n})(p_{i+1}^{n}-p_i^{n})}{2\delta x} + \frac{r_i^{n}+r_{i+1}^{n}}{2}\right).\end{aligned} \tag{1.41}$$

In the preceeding expression, k is a function of $\mathbf{u}$; because it appears multiplied by $p \in \mathbf{u}$, the differentiation of (1.41) with respect to the other components of $\mathbf{u}$ differs from the differentiation with respect to p.

Before presenting the Jacobian of the $\boldsymbol{\Xi}(\mathbf{u}; v)$ as given by the discrete version of operator $\widetilde{\boldsymbol{\mathcal{E}}}$ (1.35) in page 13, equivalent to (1.22) for the simple grid, we recall the following: the system

$$\boldsymbol{\Xi}(\mathbf{u}) = 0 \quad \text{with} \quad \boldsymbol{\Xi}_i \equiv \boldsymbol{\mathcal{E}}(\mathbf{u}_{i-1}, \mathbf{u}_i, \mathbf{u}_{i+1}; v_{\widehat{i-1}}, v_{\widehat{i}}) \quad i = 2, \ldots, N-1,$$

is defined for the state vector $(\mathbf{u}; v) \in \mathbb{R}^{M+1}$ with $\mathbf{u} \in \mathbb{R}^M$; for each x_i, it comprises M scalar equations $\Xi_i^{(m)}$ of the form $\mathcal{E} : \mathbb{R}^{M+1} \to \mathbb{R}$ — and we have taken $\widetilde{\boldsymbol{\mathcal{E}}}$ of the form (1.3) — plus a discretization (1.41) of Darcy's Law (1.4) as $\Xi_i^{(M+1)}$. For computational purposes, the velocity at position $x_{\widehat{i}}$ is compounded with the state vector $\mathbf{u}_i$, and the resulting Jacobian matrix is still block tridiagonal. The $M+1 \times M+1$ blocks of $\mathbf{J}$ have themselves a block structure: we denote the Jacobian blocks $\mathbf{J}_{i,j}$ as

$$\mathbf{J}_{i,j} = \frac{\partial \boldsymbol{\Xi}_i}{\partial(\mathbf{u}; v)_j} = \left|\begin{array}{c:c} \dfrac{\partial \boldsymbol{\Xi}_i^{(1:M)}}{\partial \mathbf{u}_j^{(1:M)}} \equiv \mathbf{J}_{i,j;1} & \dfrac{\partial \boldsymbol{\Xi}_i^{(1:M)}}{\partial v_{\widehat{\jmath}}} \equiv \mathbf{J}_{i,j;2} \\ \hdashline \partial \Xi_i^{(M+1)}/\partial \mathbf{u}_j^{(1:M)} \equiv \mathbf{J}_{i,j;3} & \partial \Xi_i^{(M+1)}/\partial v_{\widehat{\jmath}} \equiv \mathbf{J}_{i,j;4} \end{array}\right|, \tag{1.42}$$

and $\mathbf{A}_i \equiv \mathbf{J}_{i,i}$, $\mathbf{B}_i \equiv \mathbf{J}_{i,i-1}$, and $\mathbf{C}_i \equiv \mathbf{J}_{i,i+1}$, as before.

The upper left blocks of (1.42) are $M \times M$, and correspond to those given in (1.22); but with the different discretization of the convection term, they are given by

$$\begin{aligned}\mathbf{B}_{i;1} &= -\alpha\left(\frac{v_{\widehat{i-1}}\partial \mathbf{f}_{i-1}}{2\delta x} + \frac{\partial \mathbf{g}_{i-1}(\mathbf{u}_{i-1}-\mathbf{u}_i) + \mathbf{g}_i + \mathbf{g}_{i-1}}{2\delta x^2}\right); \\ \mathbf{A}_{i;1} &= \frac{\partial \mathbf{h}_i}{\delta t} + \alpha\left(\frac{(v_{\widehat{i}} - v_{\widehat{i-1}})\partial \mathbf{f}_i}{2\delta x} - \frac{\partial \mathbf{g}_i(\mathbf{u}_{i+1}-2\mathbf{u}_i+\mathbf{u}_{i-1}) - \mathbf{g}_{i+1} - 2\mathbf{g}_i - \mathbf{g}_{i-1}}{2\delta x^2} - \partial \mathbf{q}_i\right); \\ \mathbf{C}_{i;1} &= \alpha\left(\frac{v_{\widehat{i}}\partial \mathbf{f}_{i+1}}{2\delta x} - \frac{\partial \mathbf{g}_{i+1}(\mathbf{u}_{i+1}-\mathbf{u}_i) + \mathbf{g}_{i+1} + \mathbf{g}_i}{2\delta x^2}\right).\end{aligned} \tag{1.43}$$

The upper right $M \times 1$ blocks are obtained from the same equations, now differentiated with respect to $v_{\widehat{j}}$ instead:

$$\begin{aligned} \mathbf{B}_{i;2} &= -\alpha \frac{(\mathbf{f}_i + \mathbf{f}_{i-1})}{2\delta x}; \\ \mathbf{A}_{i;2} &= \alpha \frac{\mathbf{f}_{i+1} + \mathbf{f}_i}{2\delta x}; \\ \mathbf{C}_{i;2} &= \mathbf{0}. \end{aligned} \tag{1.44}$$

The last line of each block $\mathbf{J}_{i,j}$ is obtained from (1.41):

$$\begin{aligned} \mathbf{B}_{i;3} &= \mathbf{0}, \\ \mathbf{A}_{i;3}^{(m)} &= -\frac{\alpha}{\delta x}\left(\frac{\partial k_i}{\partial u^{(m)}}\,(p_{i+1} - p_i)\right) + \frac{\alpha}{2}\,\frac{\partial r_i}{\partial u^{(m)}}, \quad \text{except} \\ \mathbf{A}_{i;3}^{(M)} &= -\frac{\alpha}{2\delta x}\left(\frac{\partial k_i}{\partial u^{(m)}}\,(p_{i+1} - p_i) - k_i - k_{i+1}\right) + \frac{\alpha}{2}\,\frac{\partial r_i}{\partial u^{(m)}}, \\ \mathbf{C}_{i;3}^{(m)} &= -\frac{\alpha}{\delta x}\left(\frac{\partial k_{i+1}}{\partial u^{(m)}}\,(p_{i+1} - p_i)\right) + \frac{\alpha}{2}\,\frac{\partial r_{i+1}}{\partial u^{(m)}}, \quad \text{except} \\ \mathbf{C}_{i;3}^{(M)} &= -\frac{\alpha}{2\delta x}\left(\frac{\partial k_{i+1}}{\partial u^{(m)}}\,(p_{i+1} - p_i) + k_i + k_{i+1}\right) + \frac{\alpha}{2}\,\frac{\partial r_{i+1}}{\partial u^{(m)}}, \\ \mathbf{B}_{i;4} &= 0, \quad \mathbf{A}_{i;4} = \alpha, \quad \mathbf{C}_{i;4} = 0. \end{aligned} \tag{1.45}$$

1.7. Boundary Conditions for the Staggered Grid

At the time of this draft, these features were partially implemented, but not tested.

In the staggered grid depicted in Figure 1.2, the boundary positions x_a and x_b lie in secondary grid locations. Because of this, $\mathbf{u}_1$ and $\mathbf{u}_N$ are unknowns regardless of the boundary conditions imposed. Including the ghost points, we always have a system of N (vector) unknowns on the primary grid. The secondary grid, where velocities are defined, contains $N-1$ (scalar) unknowns: the velocity $v_{\widehat{N}}$ is beyond the ghost point, and does not appear in any equation.

Boundary conditions for p and v cannot be independently specified, just like the initial data $p(x,0)$ and $v(x,0)$ — or even the states $p(x,t)$ and $v(x,t)$ at any given time. The two variables are coupled through Darcy's Law (1.4). We first present expressions for the treatment of boundary conditions imposed on variables defined on the primary grid, analogous to those given in the previous section; then we address the specifics of pressure and velocity.

1.7.1. Boundary Conditions for Variables on the Primary Grid. We assume that the value of $\mathbf{u}$ varies linearly around the boundary, where some condition is prescribed, towards the two nodes that enclose it — one being inside the physical domain and the other being a ghost node. At the left end, this allows the definition of the equation $\boldsymbol{\Xi}_1(\mathbf{u}_1, \mathbf{u}_2)$ from a Dirichlet boundary condition as

$$\frac{\mathbf{u}_1 + \mathbf{u}_2}{2} = \boldsymbol{\gamma}_a(t) \Rightarrow \boldsymbol{\Xi}_1(\mathbf{u}_1, \mathbf{u}_2) = \mathbf{u}_1 + \mathbf{u}_2 - 2\boldsymbol{\gamma}_a(t), \tag{1.46}$$

so the coefficients for (1.26) are $\mathbf{A}_1 = \mathbf{I}$ and $\mathbf{C}_1 = \mathbf{I}$. At the right end,

$$\frac{\mathbf{u}_{N-1} + \mathbf{u}_N}{2} = \boldsymbol{\gamma}_b(t) \Rightarrow \boldsymbol{\Xi}_N(\mathbf{u}_{N-1}, \mathbf{u}_N) = \mathbf{u}_{N-1} + \mathbf{u}_N - 2\boldsymbol{\gamma}_b(t), \tag{1.47}$$

so $\mathbf{A}_N = \mathbf{I}$ and $\mathbf{B}_N = \mathbf{I}$ too.

For the Neumann case, since the boundary position x_a is equidistant from the x_2 and the ghost point x_1, the value of $\partial\mathbf{u}/\partial x$ at x_a can be written using central differences as

$$\frac{\mathbf{u}_2 - \mathbf{u}_1}{\delta x} = \boldsymbol{\gamma}_a(t), \Rightarrow \boldsymbol{\Xi}_1(\mathbf{u}_1, \mathbf{u}_2) = \mathbf{u}_1 - \mathbf{u}_2 + \delta x \boldsymbol{\gamma}_a(t^{n+1}), \tag{1.48}$$

so the coefficients for (1.26) are $\mathbf{A}_1 = \mathbf{I}$ and $\mathbf{C}_1 = -\mathbf{I}$. At the right end

$$\frac{\mathbf{u}_N - \mathbf{u}_{N-1}}{\delta x} = \boldsymbol{\gamma}_b(t), \Rightarrow \boldsymbol{\Xi}_N(\mathbf{u}_{N-1}, \mathbf{u}_N) = \mathbf{u}_N - \mathbf{u}_{N-1} - \delta x \boldsymbol{\gamma}_b(t^{n+1}), \tag{1.49}$$

and the coefficients for (1.26) are $\mathbf{A}_N = \mathbf{I}$ and $\mathbf{B}_N = -\mathbf{I}$ as well.

The expression for $\boldsymbol{\Xi}_\mathbf{1}(\mathbf{u_1}, \mathbf{u_2})$ in the Robin case is derived from the combination of (1.46) and (1.48):

$$\begin{aligned} &\alpha\frac{\mathbf{u}_1 + \mathbf{u}_2}{2} + \beta\frac{\mathbf{u}_2 - \mathbf{u}_1}{\delta x} = \boldsymbol{\gamma}_a(t) \quad \text{yields} \\ &\boldsymbol{\Xi}_1(\mathbf{u}_1, \mathbf{u}_2) = (\alpha\delta x - 2\beta)\mathbf{u}_1 + (\alpha\delta x + 2\beta)\mathbf{u}_2 - 2\delta x\boldsymbol{\gamma}_a(t), \end{aligned} \tag{1.50}$$

so $\mathbf{A}_1 = (\alpha\delta x - 2\beta)\mathbf{I}$ and $\mathbf{C}_1 = (\alpha\delta x + 2\beta)\mathbf{I}$. At the right end, from (1.47) and (1.49),

$$\begin{aligned} &\alpha\frac{\mathbf{u}_{N-1} + \mathbf{u}_N}{2} + \beta\frac{\mathbf{u}_N - \mathbf{u}_{N-1}}{\delta x} = \boldsymbol{\gamma}_b(t) \quad \text{yields} \\ &\boldsymbol{\Xi}_N(\mathbf{u}_{N-1}, \mathbf{u}_N) = (\alpha\delta x - 2\beta)\mathbf{u}_{N-1} + (\alpha\delta x + 2\beta)\mathbf{u}_N - 2\delta x\boldsymbol{\gamma}_b(t), \end{aligned} \tag{1.51}$$

so $\mathbf{A}_N = (\alpha\delta x + 2\beta)\mathbf{I}$ and $\mathbf{B}_N = (\alpha\delta x - 2\beta)\mathbf{I}$.

1.7.2. Boundary Conditions for the Velocity. The values for velocity and pressure are not independent anywhere in the physical domain, including the boundaries. Therefore, only two of the four values p_a, p_b, v_a and v_b can (and must) be specified as boundary conditions. These conditions are always of Dirichlet type. Specifing a Neumann condition for the pressure is equivalent to specifing Dirichlet condition for the velocity (see (1.4)), and specifing a Neumann condition for the velocity is of no physical interest.

Because velocity and pressure must be compatible through Darcy's Law, their initial boundary conditions are not independent from the initial state inside the spatial domain. Other state variables in $\mathbf{u}$ may be initialized with arbitrary non-smooth profiles and even jumps at the boundaries, but the pressure and velocity profiles are initialized solely from their boundary information and the initial state of the other variables in $\mathbf{u}$.

This initialization is performed solving a steady state problem on the variables $p \in \mathbf{u}$ and v, considering all other variables in $\mathbf{u}$ to be constant in time. The nonlinear system to be solved is formed by equation (1.4) and the total flow equation $\widetilde{\mathcal{E}}^{(M)} : \mathbb{R}^{M+1} \to \mathbb{R}$ stripped from the accumulation term. This is a system of ordinary differential equations in x.

CHAPTER 2

Physical models

Each physical phenomenon is modeled through a different set of equations stemming from general conservation laws and specific constitutive equations. We attempt to group different phenomena into classes, both at the mathematical formulation and computer implementation levels. In this chapter, we present the former, i.e., the mathematical models, the implementations of which are described in Chapter 3.

2.1. Three–Phase Flow Models

Models for the study of three–phase flow of water, oil and gas in porous media arise in the context of secondary oil recovery, and are presented in [**9**] and references therein. The flow is described by a pressure equation, expressing Darcy's law of force, coupled to two saturation equations which generalize the classical Buckley–Leverett equation and express the conservation of the three quantities, the masses of water, oil and gas. Darcy's law for incompressible one–dimensional flows implies that the total fluid velocity is position independent, so if it is given as a time independent boundary condition then it is constant in the whole domain, and the nondimensional system can be written as

$$\frac{\partial s_w}{\partial t} + \frac{\partial}{\partial x} f_w(s_w, s_g) = D_w \quad \text{and} \quad \frac{\partial s_g}{\partial t} + \frac{\partial}{\partial x} f_g(s_w, s_g) = D_g, \tag{2.1}$$

where s_w and s_g denote water and gas saturations, and the oil saturation is $s_o = 1 - s_w - s_g$, so the state of the fluid is defined by $\mathbf{u} = \{s_w, s_g\}$. The generalization of the classical model introduced by Buckley–Leverett in [**4**] adds the fractional flow functions f_w and f_g which are given in terms of fixed fluid viscosities μ_w, μ_g and μ_o, and of relative permeability functions k_w, k_g and k_o, i.e.,

$$f_w = \frac{k_w/\mu_w}{k_w/\mu_w + k_g/\mu_g + k_o/\mu_w} \quad \text{and} \quad f_g = \frac{k_g/\mu_g}{k_w/\mu_w + k_g/\mu_g + k_o/\mu_w}. \tag{2.2}$$

Following [**3**], the diffusive terms due to capillary pressure effects

$$D_w = \frac{\partial}{\partial x}\left(B_{ww}\frac{\partial s_w}{\partial x}\right) + \frac{\partial}{\partial x}\left(B_{wg}\frac{\partial s_g}{\partial x}\right) \quad \text{and} \quad D_g = \frac{\partial}{\partial x}\left(B_{gw}\frac{\partial s_w}{\partial x}\right) + \frac{\partial}{\partial x}\left(B_{gg}\frac{\partial s_g}{\partial x}\right) \tag{2.3}$$

represent the effect of capillary pressure differences among fluids, with the four B values taken from the matrix

$$\mathbf{B} = \begin{bmatrix} B_{ww} & B_{wg} \\ B_{gw} & B_{gg} \end{bmatrix} = \begin{bmatrix} \frac{k_w}{\mu_w}(1-f_w) & -\frac{k_w}{\mu_w}f_g \\ \frac{k_g}{\mu_g}f_w & \frac{k_g}{\mu_g}(1-f_g) \end{bmatrix} \begin{bmatrix} \frac{\partial p_{c,wo}}{\partial s_w} & \frac{\partial p_{c,wo}}{\partial s_g} \\ \frac{\partial p_{c,go}}{\partial s_w} & \frac{\partial p_{c,go}}{\partial s_g} \end{bmatrix},$$

which depends also on the capillary pressures

$$p_{c,wo} = p_w - p_o \quad \text{and} \quad p_{c,go} = p_g - p_o.$$

The model is usually simplified ([**7**]) by assuming that the relative permeability functions k depend on both fluid saturations and nothing else. The capillary pressure differences $p_{c,wo}$ and $p_{c,go}$ depend only on s_w and s_g respectively. All in all, equation (2.1) can be written as

$$\frac{\partial \mathbf{u}}{\partial t} + \frac{\partial \mathbf{f}(\mathbf{u})}{\partial x} = \frac{\partial}{\partial x}\left(\mathbf{g}(\mathbf{u})\frac{\partial \mathbf{u}}{\partial x}\right), \tag{2.4}$$

with the state vector $\mathbf{u} = \{s_w, s_g\}$, the vector function $\mathbf{f}$ given by (2.2) and the diffusion function $\mathbf{g}$ given by the matrix $\mathbf{B}$ in (2.3).

2.1.1. Quadratic Model. The simplest model we implement is derived from the three–phase flow model just introduced. It is useful for the analysis of singularities in the phase space of the solution, even if it does not capture as many features of the physical phenomenon as the more detailed model given by equations (2.1)–(2.3). In this simplified quadratic model, $\mathbf{h}(\mathbf{u}) \equiv \mathbf{u}$, $\mathbf{f}(\mathbf{u}) : \mathbb{R}^2 \to \mathbb{R}^2$ given in equation (2.2) is replaced by a quadratic homogeneous function, i.e.,

$$\begin{aligned} \mathbf{f}_{s_w}(s_w, s_o) &= \frac{1}{2}\,a_1 {s_w}^2 + b_1 s_w s_o + \frac{1}{2}\,c_1 {s_o}^2 + d_1 s_w + e_1 s_o \\ \mathbf{f}_{s_o}(s_w, s_o) &= \frac{1}{2}\,a_2 {s_w}^2 + b_2 s_w s_o + \frac{1}{2}\,c_2 {s_o}^2 + d_2 s_w + e_2 s_o, \end{aligned} \tag{2.5}$$

the viscosity matrix $\mathbf{g}(\mathbf{u}) : \mathbb{R}^2 \to \mathbb{R}^{2\times 2}$ is taken as a constant matrix $\mathbf{M}$ — often the identity $\mathbf{I}$ — multiplied by a constant ϵ, i.e.,

$$\mathbf{g}(\mathbf{u}) = \epsilon \mathbf{M}, \tag{2.6}$$

and $\mathbf{q}(\mathbf{u}) \equiv 0$. For these particular forms of $\mathbf{f}$ and $\mathbf{g}$, the expression for $\mathbf{f}'$ is

$$\frac{\partial \mathbf{f}}{\partial \mathbf{u}} = \begin{bmatrix} \frac{\partial \mathbf{f}_{s_w}}{\partial s_w} & \frac{\partial \mathbf{f}_{s_w}}{\partial s_o} \\ \frac{\partial \mathbf{f}_{s_o}}{\partial s_w} & \frac{\partial \mathbf{f}_{s_o}}{\partial s_o} \end{bmatrix} = \begin{bmatrix} a_1 s_w + b_1 s_o + d_1 & b_1 s_w + c_1 s_o + e_1 \\ a_2 s_w + b_2 s_o + d_2 & b_2 s_w + c_2 s_o + e_2 \end{bmatrix}, \tag{2.7}$$

and $\mathbf{g}' = 0$. Since $\mathbf{h}$ and $\mathbf{g}$ are constant and $\mathbf{q} \equiv 0$, the numerical scheme given by expression (1.20)–(1.21) simplifies to

$$\frac{\mathbf{u}_i^{n+1}}{k} + \frac{\mathbf{f}_{i+1}^{n+1} - \mathbf{f}_{i-1}^{n+1}}{4h} - \frac{\mathbf{g}}{2h^2}(\mathbf{u}_{i+1}^{n+1} - 2\mathbf{u}_i^{n+1} + \mathbf{u}_{i-1}^{n+1}) = \tag{2.8}$$

$$\frac{\mathbf{u}_i^{n}}{k} - \frac{\mathbf{f}_{i+1}^{n} - \mathbf{f}_{i-1}^{n}}{4h} + \frac{\mathbf{g}}{2h^2}(\mathbf{u}_{i+1}^{n} - 2\mathbf{u}_i^{n} + \mathbf{u}_{i-1}^{n}), \tag{2.9}$$

and the expressions for the iterative solution in (1.22) become

$$\begin{aligned} \mathbf{F}(\mathbf{u}_i^{n+1}) &= \frac{\mathbf{u}_i^{n+1}}{k} + \frac{\mathbf{f}_{i+1}^{n+1} - \mathbf{f}_{i-1}^{n+1}}{4h} - \frac{\mathbf{g}}{2h^2}(\mathbf{u}_{i+1}^{n+1} - 2\mathbf{u}_i^{n+1} + \mathbf{u}_{i-1}^{n+1}) \\ \mathbf{y}(\mathbf{u}_i^{n}) &= \frac{\mathbf{u}_i^{n}}{k} - \frac{\mathbf{f}_{i+1}^{n} - \mathbf{f}_{i-1}^{n}}{4h} + \frac{\mathbf{g}}{2h^2}(\mathbf{u}_{i+1}^{n} - 2\mathbf{u}_i^{n} + \mathbf{u}_{i-1}^{n}) \end{aligned} \tag{2.10}$$

and

$$\begin{aligned} \mathbf{C}_i &= \frac{1}{4h}\frac{\partial \mathbf{f}}{\partial \mathbf{u}_{i+1}} - \frac{\mathbf{g}_{i+1}}{2h^2} \\ \mathbf{A}_i &= \frac{1}{k}\mathbf{I} + \frac{\mathbf{g}_i}{\delta x^2} \\ \mathbf{B}_i &= -\frac{1}{4h}\frac{\partial \mathbf{f}}{\partial \mathbf{u}_{i-1}} - \frac{\mathbf{g}_{i-1}}{2h^2}, \end{aligned} \tag{2.11}$$

with $\mathbf{f}$ and $\partial\mathbf{f}/\partial\mathbf{u}$ given by (2.5) and (2.7).

2.1.2. Corey's Model. At the time of this draft, this model was implemented and compiled, but not validated.

This model differs from the previous one only in its flow functions. Following [**7**], the permeabilities are defined as

$$k_w = s_w^2, \quad k_g = s_g^2, \quad \text{and} \quad k_o = s_o^2; \quad \text{with} \quad s_o = 1 - s_w - s_g; \tag{2.12}$$

substituting the above in equation (2.2), we obtain the flow function

$$\begin{aligned} f_w(s_w, s_g) &= \frac{s_w^2/\mu_w}{\dfrac{s_w^2}{\mu_w} + \dfrac{s_g^2}{\mu_g} + \dfrac{(1 - s_w - s_g)^2}{\mu_o}} \\ f_g(s_w, s_g) &= \frac{s_g^2/\mu_g}{\dfrac{s_w^2}{\mu_w} + \dfrac{s_g^2}{\mu_g} + \dfrac{(1 - s_w - s_g)^2}{\mu_o}} \end{aligned} \tag{2.13}$$

with derivatives

$$\frac{\partial \mathbf{f}}{\partial \mathbf{u}} = \begin{bmatrix} \dfrac{\partial f_w}{\partial s_w} & \dfrac{\partial f_w}{\partial s_g} \\ \dfrac{\partial f_g}{\partial s_w} & \dfrac{\partial f_g}{\partial s_g} \end{bmatrix} = \frac{2}{\left(\dfrac{s_w^2}{\mu_w} + \dfrac{s_g^2}{\mu_g} + \dfrac{(1-s_w-s_g)^2}{\mu_o}\right)^2} \times$$

$$\begin{bmatrix} \dfrac{s_w}{\mu_w}\left(\dfrac{s_g^2}{\mu_g} + \dfrac{(1-s_w-s_g)^2}{\mu_o} + s_w\dfrac{1-s_w-s_g}{\mu_o}\right) & \dfrac{s_w^2}{\mu_w}\left(\dfrac{1-s_w-s_g}{\mu_o} - \dfrac{s_g}{\mu_g}\right) \\ \dfrac{s_g^2}{\mu_g}\left(\dfrac{1-s_w-s_g}{\mu_o} - \dfrac{s_w}{\mu_w}\right) & \dfrac{s_g}{\mu_g}\left(\dfrac{s_w^2}{\mu_w} + \dfrac{(1-s_w-s_g)^2}{\mu_o} + s_g\dfrac{1-s_w-s_g}{\mu_o}\right) \end{bmatrix}. \tag{2.14}$$

2.2. Dry Combustion Models

At the time of this draft, only the models in 2.2.3 and 2.2.4 were implemented and tested.

The study of combustion waves has many applications in different areas. One of them is in-situ combustion — a technique for heavy oil recovery. Models for this phenomenon consist of balance equations for both mass and energy. We initially present a combustion model from [**2**], and then present increasingly simplified versions of it [**5, 6, 10**]. The model takes into account coal distribution, allows moving fronts and uses physical formulæ for the combustion rate. The differences from the base formulation will be highlighted at the beginning of each subsection.

2.2.1. Complete Akkutlu's Model. The model comprises five dependent variables, $\mathbf{u} = \{\theta, Y, \eta, p\}$ and v: it requires the use of the staggered grid and the formulation from Section 1.6.

In the following, tildes indicate dimensional values; subscripts o and i mean "original state of the reservoir" and "state at the injection end", respectively. The dimensional independent variables $\tilde{x}$ and $\tilde{t}$ are scaled using reference values $x^* = a_s/v_i$ and $t^* = a_s/v_i^2$, where a_s is the effective thermal diffusivity and v_i is the injection velocity. The dimensionless temperature θ is the temperature $\tilde{T}(x,t)$ divided by the reference/initial reservoir temperature $\tilde{T}_o$; the dimensionless oxygen fraction Y is the oxygen mass fraction distribution of the gas $\tilde{Y}(x,t)$ divided by the oxygen mass fraction of the injected gas $\tilde{Y}_i$, which is assumed to be constant in time. The fuel conversion depth $\eta = 1 - \rho_f/\rho_{fo}$ represents the consumption of the fuel: the fuel mass distribution is $\rho_f(x,t)$, and ρ_{fo} is the initial fuel distribution, which is assumed to be spatially homogenous. The dimensionless gas pressure p is the pressure distribution

$\tilde{p}(x,t)$ divided by some reference pressure value $\tilde{p}_o$, and v is the Darcy velocity, i.e., the volumetric flow of gas per unit area.

The equation for ideal gases is given by

$$\tilde{p}M_g = \tilde{\rho}_g R\tilde{T},$$

where M_g and $\tilde{\rho}_g$ are the effective molecular weight and effective density of the gas, respectively, and R is the universal constant of gases. The already introduced reference values $\tilde{p}_o$ and $\tilde{T}_o$ satisfy this equation along with the gas density at the inlet $\tilde{\rho}_{gi}$. Therefore, with dimensionless p, ρ and θ, the equation for ideal gases becomes simply

$$\rho\,\theta = p, \tag{2.15}$$

and the non–dimensionalization just described is sumarized by

$$x = \frac{v_i\tilde{x}}{a_s}; \quad t = \frac{v_i^2\tilde{t}}{a_s}; \quad \theta = \frac{\tilde{T}}{\tilde{T}_o}; \quad Y = \frac{\tilde{Y}}{\tilde{Y}_i}; \quad p = \frac{\tilde{p}}{\tilde{p}_o}; \quad \rho = \frac{\tilde{\rho}_g}{\tilde{\rho}_{gi}}. \tag{2.16}$$

The model formulated in [**2**] considers a linear flow with heat losses, and gives a different non–dimensionalization for the equation of ideal gases. Removing the heat loss term, the system presented therein becomes

$$\begin{aligned}
\frac{\partial\theta}{\partial t} + \frac{\partial(a\rho v\theta)}{\partial x} &= \frac{\partial^2\theta}{\partial x^2} + q\Phi \\
\phi\frac{\partial(\rho Y)}{\partial t} + \frac{\partial(\rho v Y)}{\partial x} &= \frac{1}{L_e}\frac{\partial}{\partial x}\left(\rho\frac{\partial Y}{\partial x}\right) - \mu\Phi \\
\phi\frac{\partial\rho}{\partial t} + \frac{\partial(\rho v)}{\partial x} &= \mu_g\Phi \\
\frac{\partial\eta}{\partial t} &= \Phi, \quad \text{and} \\
\frac{\partial p}{\partial x} &= -\kappa v;
\end{aligned} \tag{2.17}$$

The reaction rate $\Phi(\theta, Y, \eta)$, based on the Arrhenius' law, is given by

$$\begin{cases} \Phi = Y(1-\eta)\Upsilon, \quad \text{with} \quad \Upsilon = \alpha e^{-\frac{\gamma}{\theta}}, \text{ for } \theta > 1 \\ \Phi = 0, \text{ for } \theta \leq 1. \end{cases} \tag{2.18}$$

We use (2.15) to remove $\rho(\theta, p)$, and write

$$\frac{\partial \theta}{\partial t} + \frac{\partial (apv)}{\partial x} = \frac{\partial^2 \theta}{\partial x^2} + q\Phi \tag{2.19}$$

$$\phi \frac{\partial}{\partial t}\left(\frac{pY}{\theta}\right) + \frac{\partial}{\partial x}\left(\frac{pvY}{\theta}\right) = \frac{1}{L_e}\frac{\partial}{\partial x}\left(\frac{p}{\theta}\frac{\partial Y}{\partial x}\right) - \mu\Phi \tag{2.20}$$

$$\phi \frac{\partial}{\partial t}\left(\frac{p}{\theta}\right) + \frac{\partial}{\partial x}\left(\frac{pv}{\theta}\right) = \mu_g \Phi \tag{2.21}$$

$$\frac{\partial \eta}{\partial t} = \Phi, \quad \text{and} \tag{2.22}$$

$$\frac{\partial p}{\partial x} = -\kappa v. \tag{2.23}$$

The dimensionless parameters used in the model, along with typical values [**1**], are given in Table 2.2.1.

Physical quantity	*Symbol*	*Value*
Total heat content of the porous medium	q	1.0121
Dimensionless stoichiometric coefficients for oxygen	μ	205.8
Dimensionless stoichiometric coefficients for gaseous products	μ_g	68.19
Lewis number (ratio of thermal and molecular diffusion)	L_e	0.214
Arrhenius number (dimensionless activation energy)	γ	23.69
Dimensionless reaction coefficient	α	0.027
Volumetric heat capacity ratio of the filtrating gas and matrix	a	$6.13 \cdot 10^{-4}$
Porosity of the medium	ϕ	0.3

TABLE 2.1. Typical values of dimensionless parameters.

The physical domains of the dependent variables are given by

$$\theta \geq 0, \quad 0 \leq Y \leq 1, \quad 0 \leq \eta \leq 1, \quad p > 0 \quad \text{and} \quad v \in \mathbb{R}. \tag{2.24}$$

The reference temperature $\theta = 1$ corresponds to initial reservoir temperature; the oxygen values $Y = 1$ and $Y = 0$ correspond to the fraction in the injected gas (i.e. maximum availability) and no remaining oxygen (i.e. complete consumption), respectively; similarly, the values $\eta = 0$ and $\eta = 1$ correspond to no fuel consumption (initial state) and complete consumption, respectively; pressure is non–negative, and v may have any value.

From the system (2.19)–(2.22) we have

$$\mathbf{u} = \begin{bmatrix} \theta \\ Y \\ \eta \\ p \end{bmatrix}, \ \mathbf{h} = \begin{bmatrix} \theta \\ \phi pY/\theta \\ \phi p/\theta \\ \eta \end{bmatrix}, \ \mathbf{f} = \begin{bmatrix} ap \\ pY/\theta \\ p/\theta \\ 0 \end{bmatrix}, \ \mathbf{g} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{p}{L_e\theta} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \ \mathbf{q} = \begin{bmatrix} q \\ -\mu \\ \mu_g \\ 1 \end{bmatrix} \Phi. \tag{2.25}$$

with Φ given in (2.18). For simplicity, we assume $\mathbf{g}$ to be constant, taking $\mathbf{g}_{2,2} = L_e^{-1}$. The derivatives of the other three functions are given by

$$\mathbf{h}' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\phi pY/\theta^2 & \phi p/\theta & 0 & \phi Y/\theta \\ -\phi p/\theta^2 & 0 & 0 & \phi/\theta \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad \mathbf{f}' = \begin{bmatrix} 0 & 0 & 0 & a \\ -pY/\theta^2 & p/\theta & 0 & Y/\theta \\ -p/\theta^2 & 0 & 0 & 1/\theta \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.26}$$

and

$$\mathbf{q}' = \begin{bmatrix} q\gamma(1-\eta)Y/\theta^2 & q(1-\eta) & -qY & 0 \\ -\mu\gamma(1-\eta)Y/\theta^2 & -\mu(1-\eta) & \mu Y & 0 \\ \mu_g\gamma(1-\eta)Y/\theta^2 & \mu_g(1-\eta) & -\mu_g Y & 0 \\ \gamma(1-\eta)Y/\theta^2 & (1-\eta) & -Y & 0 \end{bmatrix} \Upsilon. \tag{2.27}$$

Finally, for Darcy's Law as denoted in (1.4), we have $k(\mathbf{u}) = 1/\kappa$ and $r(\mathbf{u}) = 0$.

2.2.2. Simplified Akkutlu's Model. We use the simplified version for the nondimensional gas equation of state as given in [**10**], i.e., we replace equation (2.15) with

$$\rho\,\theta = 1, \tag{2.28}$$

so the density ρ is a function of temperature θ alone; system (2.17) becomes

$$\frac{\partial\theta}{\partial t} + \frac{\partial(av)}{\partial x} = \frac{\partial^2\theta}{\partial x^2} + q\Phi \tag{2.29}$$

$$\phi\frac{\partial}{\partial t}\left(\frac{Y}{\theta}\right) + \frac{\partial}{\partial x}\left(\frac{vY}{\theta}\right) = \frac{1}{L_e}\frac{\partial}{\partial x}\left(\frac{1}{\theta}\frac{\partial Y}{\partial x}\right) - \mu\Phi \tag{2.30}$$

$$\phi\frac{\partial}{\partial t}\left(\frac{1}{\theta}\right) + \frac{\partial}{\partial x}\left(\frac{1}{\theta}\right) = \mu_g\Phi \tag{2.31}$$

$$\frac{\partial\eta}{\partial t} = \Phi, \quad \text{and} \tag{2.32}$$

$$\frac{\partial p}{\partial x} = -\kappa v. \tag{2.33}$$

From the system (2.29)–(2.32) we have expressions similar to (2.25)–(2.27):

$$\mathbf{u} = \begin{bmatrix} \theta \\ Y \\ \eta \\ p \end{bmatrix}, \ \mathbf{h} = \begin{bmatrix} \theta \\ \phi Y/\theta \\ \phi/\theta \\ \eta \end{bmatrix}, \ \mathbf{f} = \begin{bmatrix} a \\ Y/\theta \\ 1/\theta \\ 0 \end{bmatrix}, \ \mathbf{g} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{L_e\theta} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \ \mathbf{q} = \begin{bmatrix} q \\ -\mu \\ \mu_g \\ 1 \end{bmatrix} \Phi; \tag{2.34}$$

$$\mathbf{h}' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\phi Y/\theta^2 & \phi/\theta & 0 & 0 \\ -\phi/\theta^2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad \mathbf{f}' = \begin{bmatrix} 0 & 0 & 0 & 0 \\ -Y/\theta^2 & 1/\theta & 0 & 0 \\ -1/\theta^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.35}$$

and

$$\mathbf{q}' = \begin{bmatrix} q\gamma(1-\eta)Y/\theta^2 & q(1-\eta) & -qY & 0 \\ -\mu\gamma(1-\eta)Y/\theta^2 & -\mu(1-\eta) & \mu Y & 0 \\ \mu_g\gamma(1-\eta)Y/\theta^2 & \mu_g(1-\eta) & -\mu_g Y & 0 \\ \gamma(1-\eta)Y/\theta^2 & (1-\eta) & -Y & 0 \end{bmatrix} \Upsilon. \tag{2.36}$$

In the current implementation, the matrix g is assumed constant.

2.2.3. Box Akkutlu's Model. We further simplify the model by removing Darcy's law and the pressure altogether, leaving v as a dependent variable in $\mathbf{u}$ and using the discretization over the simple grid from Section 1.4. System (2.17) becomes

$$\frac{\partial\theta}{\partial t} + \frac{\partial(av)}{\partial x} = \frac{\partial^2\theta}{\partial x^2} + q\Phi \tag{2.37}$$

$$\phi\frac{\partial}{\partial t}\left(\frac{Y}{\theta}\right) + \frac{\partial}{\partial x}\left(\frac{vY}{\theta}\right) = \frac{1}{L_e}\frac{\partial}{\partial x}\left(\frac{1}{\theta}\frac{\partial Y}{\partial x}\right) - \mu\Phi \tag{2.38}$$

$$\frac{\partial\eta}{\partial t} = \Phi, \quad \text{and} \tag{2.39}$$

$$\phi\frac{\partial}{\partial t}\left(\frac{1}{\theta}\right) + \frac{\partial}{\partial x}\left(\frac{v}{\theta}\right) = \mu_g\Phi. \tag{2.40}$$

We changed the order of the last two equations in the system above, because $g^{(4)}(\mathbf{u}) \equiv 0$ in equation (2.40) (corresponding to (2.31)). This equation cannot be solved using the Crank–Nicolson discretization: we use the box scheme to solve it. As a side effect, in this model we consider only the forward combustion wave with speed $v > 0$, i.e., combustion advances from left to right. The physical domains of the dependent variables are given by

$$\theta \geq 0, \quad 0 \leq Y \leq 1, \quad 0 \leq \eta \leq 1, \quad v > 0. \tag{2.41}$$

From the system (2.37)-(2.40) we have

$$\mathbf{u} = \begin{bmatrix} \theta \\ Y \\ \eta \\ v \end{bmatrix}, \ \mathbf{h} = \begin{bmatrix} \theta \\ \phi Y/\theta \\ \eta \\ \phi/\theta \end{bmatrix}, \ \mathbf{f} = \begin{bmatrix} av \\ vY/\theta \\ 0 \\ v/\theta \end{bmatrix}, \ \mathbf{g} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{L_e\theta} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \ \mathbf{q} = \begin{bmatrix} q \\ -\mu \\ 1 \\ \mu_g \end{bmatrix} \Phi, \tag{2.42}$$

and

$$\mathbf{h}' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\phi Y/\theta^2 & \phi/\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\phi/\theta^2 & 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{f}' = \begin{bmatrix} 0 & 0 & 0 & a \\ -vY/\theta^2 & v/\theta & 0 & Y/\theta \\ 0 & 0 & 0 & 0 \\ -v/\theta^2 & 0 & 0 & 1/\theta \end{bmatrix} \tag{2.43}$$

$$\mathbf{q}' = \begin{bmatrix} q\gamma(1-\eta)Y/\theta^2 & q(1-\eta) & -qY & 0 \\ -\mu\gamma(1-\eta)Y/\theta^2 & -\mu(1-\eta) & \mu Y & 0 \\ \gamma(1-\eta)Y/\theta^2 & (1-\eta) & -Y & 0 \\ \mu_g\gamma(1-\eta)Y/\theta^2 & \mu_g(1-\eta) & -\mu_g Y & 0 \end{bmatrix} \Upsilon, \tag{2.44}$$

with $\Upsilon = \alpha\exp(-\gamma/\theta)$ so $\Phi = Y(1-\eta)\Upsilon$.

Because there is no combustion ahead or behind this wave, the reaction rate Φ must vanish, and therefore at least one of the following conditions must be satisfied: $\theta = 1$, $Y = 0$ or $\eta = 1$. In [**10, 6**] it was shown that there is only one admissible possibility for this to happen: the domain is fuel-deficient behind the wave, and is oxygen deficient ahead of it. Denoting these *burned* and *unburned* states by the superscripts b and u, we write these conditions as

$$\theta^b > 0, \quad Y^b = 1, \quad \eta^b = 1, \quad v^b > 0, \tag{2.45}$$

$$\theta^u = 1, \quad Y^u = 0, \quad \eta^u = 0, \quad v^u > 0, \tag{2.46}$$

2.2.4. Chapiro's Model. ASK GRIGORI ABOUT REFERENCE BELOW

A greatly simplified version of the model in 2.2.3 was derived in [], taking the velocity as a constant v instead of as dependent variable. Therefore, the variables in this model are temperature θ, oxygen fraction Y and fuel η, but unlike in the previous model, $\eta = 0$ represents fuel deficiency (instead of $\eta = 1$), i.e., in this model η maps to what in the previous model was $1 - \eta$. The simplified governing equations are

$$\frac{\partial \theta}{\partial t} + a\frac{\partial \theta}{\partial x} = \frac{\partial^2 \theta}{\partial x^2} + q\Phi \tag{2.47}$$

$$\frac{\partial Y}{\partial t} + v\frac{\partial Y}{\partial x} = -\mu\Phi \tag{2.48}$$

$$\frac{\partial \eta}{\partial t} = -\Phi. \tag{2.49}$$

The combustion rate is described by Arrhenius' law

$$\Phi = KY\eta e^{-\gamma/\theta},$$

and the boundary conditions for the Riemann problem are:

$$\text{Left:} \quad T_L, \quad Y_L = 1, \quad \eta_L = 0;$$

$$\text{Right:} \quad T_R, \quad Y_R = 0, \quad \eta_R = 1.$$

CHAPTER 3

Computer Implementation

In this chapter we describe the computer implementation of the models previously presented. It is not the goal of this document to repeat what is already written in the code (in the form of code), but rather to provide the rationales and motivations behind it. Those not interested in the internals may skip directly to 3.6, where the compilation, execution and output visualization are described.

Computational efficiency is the second architectural constraint, the first one being of course the mathematical validity and accuracy of the solution. In our attempt to satisfy both simultaneously, we have chosen to employ various techniques that permeate the code, in detriment of others which we deliberately avoided. The program is written in ANSI C, and parallelism is implemented either through OpenMP or MPI. These choices are mainly with portability in mind, although efficiency was a major factor as well. There is no enforced encapsulation, i.e., globally used data is declared as globally visible static data. The code uses compile–time resolution whenever possible, either through conditional compilation, code inlining or other hard optimizations such as constant–based loop unrolling and condition evaluation.

A third architectural constrain is that, in writing a source code where these optimization opportunities are clear for the compiler's optimizer, we must not hinder the code any less clear for ourselves. The trade–offs between efficiency and readability are inevitable, however. Thus is this document further justified.

For brevity, we do not reproduce the source code here. Instead, we provide references to it, which can be looked for in the source files. Such references will take the form `[file]:DOC_<mnemonic>`. For instance, the reference `sourcemain.c:DOC_foobar` means that in the file `sourcemain.c` there is a comment containing the string `DOC_foobar` to which the current explanation is related. The `file` may be absent in consecutive references to a single source file, or when otherwise implicit.

Data objects and functions will have their identifiers reproduced verbatim in this document, but regular expression syntax will be use for brevity, e.g., variables `u_now`, and `bc_left_now` and `bc_right_now` may be referred to simply as `*_now`, and variables `u_now` and `u_past` may be referred to as `u_{now,past}`. Functions will be distinguished from variables by the `()` following the identifier. This does not imply anything about the parameters of a

function: the formal parameter list is given in the source code where the function is declared, and the actual parameters where it is called. An optional `[file]:` may be given for data objects and functions as well.

3.1. Parallelism as an Option

The use of parallelism, either with OpenMP or MPI, is optional — not only at execution as with any OpenMP/MPI parallel program, but at compile time as well, i.e., the program can be compiled and linked in a development environment that does not provide these parallelism facilities. This is achieved through rather extensive use of conditional compilation in the source code for the solver itself, which renders it particularly complex.

All parallelism related constructs are isolated inside `#if`/`#ifdef` blocks conditioned by the macros `WITH_MPI` and `WITH_OPENMP`, and all of these conditionally compiled statements are found inside the main source file (`solvermain.c`). Compiling the program without defining either of these macros does not require either parallel library. Some OpenMP specific `#pragma`-directives are found in source files other than `solvermain.c`, but they should be ignored by the compiler in the case of OpenMP–disabled compilation.

PARALLEL: *In the following three sections, we describe initialization, computation and output, focusing on the serial implementation. Parallelism details are provided as complementary notes in italic like this one, and should be skipped on a first reading or depending on the information being sought.*

3.2. Initialization

In describing the initialization of the simulator, we consider only the time evolution problems. A steady state problem can be seen as a time evolution problem with one single time step to be taken. The concept of time step should be clear from the previous chapter, as well as the basic data objects that are required to perform all computations.

3.2.1. Data Objects. Data objects are static variables, usually in global scope. This requires careful and judicious naming conventions and choice of where to declare and define each variable. Some of these data objects are declared/defined in `solvermain.c`, lumped under the following tags:

`DOC_file_vars` File descriptors for input and input echo (text files) and output (binary file). See 3.2.2 and 3.4 for further information on input and output, respectively.

`DOC_timing_vars` The program uses three wall–clock timers for the total running time and the time spent in parallel and serial execution, named

and enumerated accordingly. The distributed (MPI) parallel implementation keeps one such timer on each process, thus the size of the `times` array.

`DOC_step_vars` These variables store the current simulation time, the current time step, control when the simulator stops and when it produces output. Further explanations are given in 3.3.1 and 3.4.1.

`DOC_plot_vars` Preferred/initial visualization interval. These only bridge post–processing information from the input file to the output, i.e., they are not used by the simulator.

`DOC_ghost_vars` These variables store the ghost points for implementing boundary conditions.

PARALLEL: *The coupling lines between data blocks allocated to different processes (see Chapter 4) are implemented as Dirichlet boundary conditions, i.e., in parallel execution, they hold states at inner (non–boundary) nodes which are "boundaries" between regions allocated to separate processes.*

`DOC_newton_vars` These variables store quantities related to Newton's method stopping criteria, see 1.2.1 and 3.3.1.

`DOC_proc_vars` These variables are set to 1 in serial compilation and/or execution. PARALLEL: *In parallel execution,* `P` *holds the number of processes and* `I` *holds the process identity (rank), so* $0 \leq$ `I` $<$ `P`.

`DOC_state_vars` The state vectors at two time levels `u_now` and `u_past`, and a collection `time_stats` of physics–dependent values which are computed in physics–dependent routines at each time step. Every physics must provide at least the next time step size in this array.

Physics provide an access routine that fills `time_stats`, but have private storage for these data. This encapsulation is required in serial execution in order to keep the globally visible storage constant while the private storage is updated: the former is associated to the data to be output, while the latter is required for the next time step. PARALLEL: *This double storage is further required in the case of parallel execution, because the new values are computed locally and synchronized at each Newton iteration.*

`DOC_solver_vars` These variables store the many data objects used by the solver described in 4.1. The nomenclature is in direct correspondence. The `IPIV*` arrays are LAPACK–related.

`DOC_parallel_vars` These auxiliary variables are necessary only in parallel execution and are described in Chapter 4.

PARALLEL: *The* `#pragma omp threadprivate` *directives indicate some variables are to have one copy per thread in the OpenMP runtime. Without this directive, one single variable is shared by all threads. These directives are ignored by the compiler in the alternative compilations.*

In the MPI compilation, processes have their own variable space and therefore one copy of each variable. The problem of concurrency/synchronization manifests itself in other ways: look in `solvermain.c` *for identifiers containing* `mpi` *or* `MPI`*. Further explanations are given in Chapter 4.*

Other globally required/available variables and constants are declared elsewhere, such as the physics–dependent ones. The most pervading of these is the number of state variables M, defined as a physics–dependent macro `M`, and there are many physics–dependent variables that are read from the input file. Being physics–specific, these are explained in 3.5.

In {`grid,data`}`.h`, under the tags `DOC_`{`grid,data`}`_vars`, the following data objects map to the discretization definitions given in Chapter 1 as in Tables 3.1 and 3.2.

Variable(s)	Corresponding object(s)
`double *x, h`	A vector of size N containing the x_i positions of the grid points (boundaries included), $i = 1, \ldots, N$, and the constant spacing value $\delta x = x_i - x_{i-1}$. Note that arrays in C start at index 0.
`double grid_lim[2]`	The extremes of the physical domain, i.e., $[x_a, x_b]$.
`int N, n_cells, K[MAXP], Ko[MAXP]`	Respectively N, $N-1$, $K_p \approx N/P$ and (roughly) $\sum K_{p-1}$. Recall that `P` is nonunit only in parallel execution, so `K[0]=N` and `Ko[0] = 0` in serial compilation and/or execution. See chapter 4 for further information on K_p.

TABLE 3.1. Data objects associated with spatial grid, in `grid.h`.

PARALLEL: *The variables* `K` *and* `Ko` *hold the information regarding how the N grid points are distributed across the P processes. This will be explained in Chapter 4. For now, it suffices to keep in mind the identities shown for the serial case in Table 3.1.*

Boundary data may vary greatly, depending not only on the type of boundary condition imposed but also on how the data associated with it is provided in the input file. Therefore, most of the boundary–related data, either read from the input file or post–processed from these, are private to the code in `bc.c`, and a few interface functions perform all required

Variable(s)	Corresponding object(s)

`double *data`
`int n_data, data_type`
`enum {DATA_UNKNOWN,`
`DATA_PIECEWISE_LINEAR,`
`DATA_PIECEWISE_CONSTANT}`
	The initial data $\mathbf{u}^0(x)$ (see 3.2.2 below).

TABLE 3.2. Data objects associated with initial values $\mathbf{u}_0$.

operations. The only two globally visible variables under `bc.h:DOC_bc_vars` are `bc_type` and `bc_data_type`, which represent, for each variable on each boundary, the type of boundary condition imposed and the type of data provided for it, respectively.

3.2.2. Input Data Loading and Processing. All data objects are read from the input file at the beginning of execution. The current implementation uses an input subsystem that reads a text file containing variable declarations in standard MATLAB syntax and stores them in variables with the same name. Array indexing in the input file is assumed to start from 1, as in MATLAB: thus the components of any M-vector or matrix will be refered by an index in the range $\{1, M\}$. This subsystem is fully contained in `mfile.[hc]`, and is peripheral to the simulator. We outline its interface here. **The implementation of the input subsystem is not documented.**

3.2.2.1. *Input File Parsing.* The input file is read at `solvermain.c:DOC_input_file`. Calls to macro / routines `[mfile.h]:mfile_*` cause the name given as a parameter to be stored and associated with the data object, i.e., the data object is *registered* for loading from the input file. The calls to `{grid,data,bc,phys}_load()`, localize this registration to the files where the data objects are actually declared/defined.

The various `mfile_` methods currently implemented allow for integers, doubles, and arrays and matrices of doubles to be read: the alternatives available are those sufficient for the current necessities, and can be readily added to.

As in MATLAB, N–vectors are $1 \times N$ matrices. Any matrix input data object may have its size known beforehand or not. If the size of a matrix is known *a priori*, the pointer passed as a parameter to `mfile_double_fix_mat()` should be valid, and the matrix is read row–wise from the file and stored column–wise in the memory. If the size of the matrix is not known, the pointer passed to `mfile_double_var_mat()` will be overwritten to point to dynamically allocated storage, and the matrix will be read and stored row–wise in memory.

The actual reading and parsing of the input file is done with the call to `mfile_read()`. It performs the following tasks:

- All registered variables are read from the input file, and stored in the data objects with the corresponding identifier.

- An echo of the input file is generated, with the same filename appended by `.echo`, containing exactly the same text, including comments.
- If a registered variable is not found in the input file, but its `mfile_*` statement included a default value, the corresponding data object is initialized to that default. A warning is issued on the terminal, and the missing variable is appended to the echo file with a comment stating it was not found.
- If a registered variable is not found in the input file, and its `mfile_*` statement did not include a default value, an error message is issued to the terminal and the program aborts.
- If there is some size inconsistency for a vector or matrix variable of fixed size, e.g. wrong number of elements at some given row in the input file and/or wrong number of rows, the corresponding pre-allocated buffer is filled up to the point where the data was correctly provided: in the case of a statically allocated buffer, this means the rest of the vector/matrix will be left at the compiler initialized status, i.e., with zero values. A warning is issued to the terminal.
- If an unregistered variable is found in the input file, or in the case of a syntax error from which the parser cannot recover, the program aborts.

3.2.2.2. *Grid Initialization and Memory Allocation.* The subsequent call to `grid_init()` at `solvermain.c:DOC_grid_init` initializes the `x` array and constants related to N presented in Table 3.1. These determine the sizes of the memory blocks to be allocated: notice that these sizes depend on the number of processes too.

Most storage and buffers are allocated to the required space in the following calls under `DOC_memory_allocation`. The wrapper macros `checkmalloc.h:*CHECKMALLOC` and related functions in `checkmalloc.[hc]` do basic bookkeeping of the amount of memory allocated, which is displayed at runtime. This code is peripheral to the simulator, and mostly self–explanatory.

Many of the variables allocated at this point of the source code are only used in parallel execution, and their relevance will be made clear in Chapter 4. In fact, from this point on, most of the code in `solvermain.c` is not compiled into the serial program. The next relevant point in the serial compilation is under `DOC_data_init`, where the calls to `{bc,data,phys}_init()` are found.

3.2.2.3. *Boundary Data.* The first call under `DOC_data_init` initializes the boundary data. Boundary data varies greatly: for a model with $M = 2$, the simplest way to specify the left boundary condition is with the line `bc_left = [0,1];` in the input file, which means constant Dirichlet conditions are to be applied to both state variables, with values 0 and 1. This is equivalent to `bc_left=[0,1; 0,0];` or `bc_left=[0,1; 0,0; 0,0];`: the second row

of the matrix `bc_left` specifies boundary type Dirichlet, and the third specifies the data is constant–in–time, as provided in the first row. The second and third rows default to zero.

Recall from Section 1.5 that boundary data is given as a $\gamma_\alpha^m(t)$ function, $\alpha \in \{a, b\}, m = 1, \ldots, M$. The values in the first row of `bc_{left,right}` are constant values for γ, which is the most common case. If γ is not constant, a separate entry must be given in the input file to provide the necessary data.

Summarizing, the boundary conditions imposed on each state variable are described by three values given in each column of the matrices `bc_{left,right}`. For $m = 1, \ldots, M$, these values have the following meaning.

- `bc_{left,right}(1,m)` **Constant value for $\boldsymbol{\gamma_\alpha^m}$.**
  This value is ignored if `bc_{left,right}(3,m)` is non–zero, i.e., if the value of γ_α^m varies in time.
- `bc_{left,right}(2,m)` **Boundary condition type**.
  Either 0 for Dirichlet, 1 for Neumann or 2 for Robin, respectively. Defaults to Dirichlet. If set to Robin, extra data (the α and β coefficients of equation (1.32)) must be provided for this variable (see below).
- `bc_{left,right}(3,m)` **Boundary data type**.
  Either 0, 1 or 2 for constant–in–time, transient time series or periodic time series, respectively. Defaults to constant–in–time. If non–constant data are selected, they must be provided separately (see below).

Some extra information may be required, in addition to that contained in `bc_{left,right}`: if Robin conditions are imposed anywhere, the two coefficients must be given; and if the data is non–constant anywhere, a time series must be given. If such extra information is required, for example, for state variable $m = 2$ at the left boundary, then a two–column matrix named `bc_left_data_2` must be present in the input file, containing a series of $(t, \gamma_a^2(t))$ pairs, with increasing t. If the boundary type is Robin, the first row of this matrix contains α and β, and the following rows contain the data series. If the boundary type is Robin with constant γ, then the matrix has only one row and γ is given by `bc_left(1,2)`. This is MATLAB indexing: notice that the corresponding indexes inside the C program would be `bc_left[0][1]`.

The increasing time values of a time series cover some range $[t_{MIN}, t_{MAX}]$. It is required that $t_{MIN} = 0$. Depending of the series being transient of periodic, evaluations for $t > t_{MAX}$ will either yield the value of $\gamma(t_{MAX})$, for the transient case, or wrap around to $\gamma\big(t - t_{MAX} \times \lfloor t/t_{MAX} \rfloor\big)$, for the periodic case.

3.2.2.4. *Initial Data.* The routine `data_init()` interpolates the state values in `data` into the storage for the current (and now initial) state vector `u_now`. Initial data is provided in `data` as an array of size `n_data` of $(M+1)$-vectors, i.e., it contains `(M+1)*n_data` floating point values: the first component of each $(M+1)$-vector is some x, and the next M components are the state variables. The x values should be in increasing order. If two or more consecutive $(M+1)$-vectors are given for a same x position, the first is used as the value at x^- and the last is used as the value at x^+: this allows discontinuities to be introduced in linearly interpolated data.

How these data are interpreted depends on the value of `data_type`. The size of `data` (i.e., the value of `n_data`) need not match N, just as the values of x stored in `data` need not match actual grid points. The current implementation provides two mappings:

- `data_type = 1` → **Piecewise Linear:** Each $(M+1)$-vector corresponds to a *nodal* value, and the values are linearly interpolated between the given x positions. The x values in the array must cover the whole domain $[x_a, x_b]$, unless Dirichlet conditions are imposed. In this latter case, the boundary conditions suplement the initial data (see below). On the other hand, if the minimum and/or maximum values of x in `data` are outside the domain, the extra information is ignored.
- `data_type = 2` → **Piecewise Constant:** Each $(M+1)$-vector corresponds to a constant value over a *segment*, and the x value given in the $(M+1)$-vector is the *right* end of the segment to which that value corresponds. It is implicit that the left end of each segment is the right end of the *previous* segment. From that, it is implicit that the left end of the *first* segment is the left end x_a of the domain. The entries in `data` for which $x < x_a$ are ignored; the first entry for which $x > x_b$ has x set to x_b, and any others are ignored.

The preceding call to `data_make_compatible()` fills in missing initial piecewise linear data using Dirichlet boundary conditions. If the initial data are missing at some boundary for which Dirichlet conditions are given, these are applied. If initial data is still missing, the program aborts. It is always the case that the boundary data take precedence.

PARALLEL: *The* `ghost_*_now` *values are also filled in by* `data_init()`*, but only in parallel execution. See Chapter 4.*

This completes the physics–independent data initialization: the call to `phys_init()` sets up physics–dependent variables, which are discussed in 3.5.

Last, statistics generated during each time step are gathered in the `time_stats` array. This first call to `get_local_time_stats()` retrieves a suggested initial time step size that was computed inside `phys_init()`: how this is done is described in 3.5.

3.3. Time Evolution

At the point `solvermain.c:DOC_time_evolution`, the initial/boundary value problem presented in Chapter 1 is fully defined, i.e., all the required information has been read and processed for the time evolution solution described in Section 1.2 to be carried out. The last initializations concerning the simulation time and the time step counter are found under the tag `DOC_time_evolution`, immediately preceding the core of the simulator. The wall–clock timers and output file are also initialized at this point: the output format is described in 3.4.

Two nested iterative procedures are performed. The outermost is the time stepping algorithm, which solves a nonlinear system; to do so, it has an embedded iterative Newton solver. The outermost loop starts at `DOC_time_loop_start` and ends at `DOC_time_loop_end`, and inside it the Newton loop is marked by the tags `DOC_newton_loop_start` and `DOC_newton_loop_end`. Of the approximately three hundred lines of code that make up this solver, less than one quarter is included in the serial compilation. The other three quarters are parallelization code. In this section we describe the serial implementation: it is futile effort to try to understand the parallel implementation without first having a good grasp on the serial one.

The first action taken in the time evolution loop is to truncate the value of `time_step`, as in `DOC_truncate`: the simulator may advance up to `sim_time+time_step` as long as this value does not exceed the values of `next_print_time`, when output should be generated (see 3.4) or `stop_time`. Once the time step is defined, the call to `phys_set_dt()` initializes physics–dependent variables that depend on the time step size (if any). The value of `time_step` is assumed constant until the beginning of the next time iteration.

In the process of truncating the time step size, the target time of the current time step may be the requested `stop_time`, in which case `stop_flag` is set. This will cause the time loop to break at the end of the current iteration. It may also happen that `stop_flag` is set if the number of time steps has reached the maximum allowed number stored in `stop_step`.

Under `DOC_newton_init`, the current state variables stored in the `*_now` arrays is copied into the `*_past` arrays, and used to compute the part of the $\boldsymbol{\Xi}(\mathbf{u})$ which is related to the previous time level and therefore constant throughout each time step. Recall from equation (1.8) that, in order to compute the state at time level $n+1$ from the known state at time level n, we seek the solution of $\boldsymbol{\Xi}(\mathbf{u}) = \mathbf{0}$: but $\mathbf{u}$ is actually $\mathbf{u}^{n+1}$, and $\boldsymbol{\Xi}(\mathbf{u})$ is just a shorthand notation for $\boldsymbol{\Xi}(\mathbf{u}^{n+1}; \mathbf{u}^n)$. If all additive terms that can be computed exclusively from the previous time level are moved to a separate object, i.e., if we denote $\boldsymbol{\Xi}(\mathbf{u}) \equiv \widetilde{\boldsymbol{\Xi}}(\mathbf{u}^{n+1}; \mathbf{u}^n) + \mathbf{y}(\mathbf{u}^n)$, we can at this point compute values of $\mathbf{y}$, and then use them through the whole Newton process. These two tasks are performed here, by the call to `build_y()`, and `build_tri()`, which is called a bit further on.

At the beginning of each Newton iteration, first the buffers associated to `time_stats` private to the physics are reset. They will be written during each Newton iteration with

tentative values: when some stopping criterion is reached, the values contained therein are those to be used in the next time step. Next, the boundary ghost points are filled in, and the block tridiagonal matrix is built with the call to `build_tri()`, whose parameters up to and including `y` are input, and the rest are output. The names of the routines and the parameters are obvious — they build the block tridiagonal system from Equation (1.12). The code for these `build_*()` functions is in `tridiag.[hc]`.

From this point until `DOC_newton_loop_end`, most of the code is relevant only if parallelism is used. We highlight the code segments which are executed in the serial compilation, leaving the rest to be described in Chapter 4:

- At `DOC_residual_reset`, the local array for residuals is initialized to zero. At `DOC_local_residual_Xi`, its first half is filled with the norm values of the right hand side.
- At `DOC_local_tridiag`, the linear system local to this process is solved: in the serial case there is only one process. Computationally, this replaces the values of the right hand side $\boldsymbol{\Xi}(\mathbf{u}^{(l)})$ of (1.12) stored in `Xi` with the value of $\delta\mathbf{u}$. The algorithm used is a block version of standard LU factorization, described in Appendix A.
- At `DOC_newton_update`, the `u_now` array is updated, completing the iteration as in Equation (1.10), and the local residuals array second half is filled with the norms of the solution for this Newton iteration.
- At `DOC_newton_loop_end`, the iteration counter `newton_counter` is incremented and the stopping criteria for Newton's method are checked, based on the current residuals, perhaps ending this inner loop.

Finally, at `DOC_time_loop_end`, the time step number `step` and the simulation time `sim_time` are incremented, output is generated if appropriate (see 3.4.1), and the time loop breaks if `stop_flag` is set. Under `DOC_wrapup`, the simulators prints final diagnostics information to the terminal, regarding memory allocation and total runtime, and closes the output file.

3.3.1. Stopping Criteria. At the end of each time loop, the function `maybe_stop_newton()` in `solvermain.c` is called: it returns nonzero if the Newton convergence criteria in 1.2.1 are satisfied by the residues of the last Newton iteration, stored in the $2M$-array `newton_residues`.

In `maybe_stop_newton()`, the the first M values in `newton_residues` are tested against `newton_tol_Xi`, and the latter M values are tested against `newton_tol_u`, corresponding to the two convergence criteria in 1.2.1. The residual accumulation can be found in the Newton loop, looking for the updates of these variables.

3.4. Output

The simulator generates interactive text output to the terminal, and binary output to a file. The output to the terminal consists of diagnostics information: it is generated at constant real–time intervals, and includes the elapsed simulation time, predicted remaining and total time, and some qualitative information about the current state of the simulation. This work is performed by the `*_diagnostics()` routines in `solvermain.c`, which in turn call physics specific `phys_diagnostics()`.

Writing to the binary file is performed in the routine `maybe_print_state()`. The routine is partly described in the following subsection, along with the file format, and partly in section 4.4 because of MPI related delayed output issues.

The periodicity with which binary output is generated is controlled by the `print_by_step`, `print_time_interval` and `print_step_interval` variables given in the input file. These variables have default values 1, 0 and 1, respectively, indicating that output is to be generated at the end of every time step.

Conceptually, output first is sent to the binary file just after initialization, i.e., the initial data $\mathbf{u}_0$ corresponding to time level $n = 0$ and simulation time $T = 0$, stored respectively in `u_now`, `step` and `sim_time`, is always the first record stored in the output file. The output of subsequent records is generated at the end of each time step, with the call to `maybe_print_state()` under `DOC_time_loop_end`, according to the following rule:

- If the flag `print_by_step` is zero, output is generated at constant intervals in the simulation time. If `sim_time` is a multiple of `print_time_interval`, output is generated. Because these are floating point values, the truncation and tracking performed under `DOC_truncate` and in the function `maybe_print_state()` is required.
- If the flag `print_by_step` is nonzero, output is generated at constant intervals in the time level. If `step` is a multiple of `print_step_interval`, output is generated. These are integer values, and the tracking of when to generate output is done in `maybe_print_state()`.

The final state of the simulation is sent to the output file regardless of the two criteria above.

3.4.1. Binary File Format. The first call to `maybe_print_state()`, after data initialization and before the first time step is taken, prints a header containing information that is not related to the time level. The contents of this header are those dumped under `DOC_output_header`, and are described in Table 3.3. It also dumps a binary version of the input file, so that restart files can be generating solely from postprocessing the output.

Each subsequent record (including the initial data) is dumped under `DOC_output_record`, and has the format described in Table 3.4. Last, the next simulation time when output

should be generated is determined under `DOC_output_tracking`. The rest of the code in this function is explained in 4.4.

Record type[size]	Data
`char[8]`	String identifying the simulator version, e.g. `"NITRO001"`.
`char[64]`	String identifying the physics version, e.g. `"QUAD001"`.
`int32`	Number of grid points N.
`int32`	Number of state variables M.
`int32`	Number `_Nts` of values in `time_stats`.
`char[16*M]`	Strings identifying the components in the state vector.
`char[8*_Nts]`	Array of `_Nts` strings identifying the values in `time_stats`.
`double[2]`	Physical domain boundaries x_a and x_b.
`double[N]`	Spatial grid x.
`double[2]`	Initial visualization interval in physical domain.
`double[M*2]`	Initial visualization interval for each state component.

TABLE 3.3. Output file header.

Record type[size]	Data
`double`	Simulation time
`double`	Time level (saved as double, albeit integer).
`double[_Nts]`	The contents of `time_stats`.
`double[N*M]`	The contents of `u_now`.

TABLE 3.4. Output file header.

3.5. Physics Implementation

A particular "physics" comprises one particular physical model and one particular discretization. These two define the specific case of the general problem solved by the procedure presented so far in this chapter. In the implementation, we wish to maintain the same level of abstraction we have kept so far, i.e., the code that performs the bulk work of initialization, time evolution and output is physics–independent. In this section, we first describe the general (physics–independent) interface with the physics–dependent code, and then the implementation of each physics that was presented in Chapter 2.

3.5.1. Public Interfaces. The actual physics to be compiled is determined in the `makefile`, and physics–dependent routines are made visible in the physics–independent code through declarations in `phys.h` and `discrete.h`. In the `makefile`, a macro that identifies the physics is defined and added to the compilation flags. For example, the quadratic model presented in 2.1.1 is labeled `quad`. If the makefile variable `PHYS` is set to `quad`, the macro

QUAD is defined while the code is compiled, i.e., -DQUAD is added to the compilation flags, and the files containing the implementation of this physics — meaning both the equations in 2.1.1 and the discretization in 1.4.1 — are added to the list of objects to be linked: in this case, the objects rcdphys.o and rcddiscrete.o.

The macro defining the physics — QUAD in the example — is used in the {phys,discrete}.h files to include the proper physics–dependent headers. The physics–independent headers are included in physics–independent sources {solvermain,grid,tridiag}.c.

3.5.1.1. *The* *phys.h *files.* First and foremost, each physics provides constants M, M2 and _Nts, which are the size of the state vector and its square, i.e., M and M^2, and the number of objects made available through phys_get_local_time_stats(), i.e., the size of the time_stats vector. It is always the case that _Nts > 0, and the first value of time_stats is a suggestion for the next time step size, based on the current state of the simulation.

Physics also provide some standardized descriptive information:

`char phys_version[64]` The name and version of the physics, e.g. QUAD001.

`char phys_comp_names[M][16]` An array of strings of length sixteen, containing the names of the M state variables.

`char time_stat_names[_Nts][8]` An array of strings of length eight, describing the contents of the _Nts values in the time_stats array.

The following functions account for all physics dependent operations:

`phys_load()` This is called under solvermain.c:DOC_input_file. Physics dependent parameters are registered for loading by this function.

`phys_init()` Initializes physics private data, after the input file had been loaded and the global data initialized, including the initial state vector.

`phys_*_time_stats()` The phys_get_local_time_stats() function fills in the time_stats vector passed by the main solver at the end of each time step; because the computation of the values returned by phys_get_local_time_stat is cumulative in nature, i.e., maxima/minima over the state array, phys_reset_local_time_stats() is called in solvermain.c to reset the accumulation buffers at the beginning of each time step. PARALLEL: *The quantities stored in the* time_stats *array by* phys_get_local_time_stats *are, as suggested by the name,* local *to the current process. In order to get* globally *valid quantities,* phys_reduce_time_stats() *takes the values generated at each process, i.e., corresponding to each segment of the solution,*

and performs the applicable computations, e.g., maximize, minimize or average. Put simply, it takes P sets as input, each representative of a K sized segment of the state vector, and returns one set as output, representative of the whole.

`phys_set_dt()` After the actual time step size is determined in `solvermain.c` under `DOC_truncate`, this function allows the physics to precompute any temporaries it may have related to the value of `time_step`.

`phys_begin_diagnostics()` Outputs physics–dependent information to the terminal at the beginning of a run.

`phys_f()` This is where the equations that define the physics are computed. It is heavily bound to the functions in the `*discrete.c` files, which are discussed next.

3.5.1.2. *The* `*discrete.h` *files.* The two functions in this file, namely `discrete_{tri,y}()`, together with `tridiag.h:build_{tri,y}()`, build the tridiagonal system (1.12). Some complications arise in parallel execution, however: these are discussed in Chapter 4.

The functions in `tridiag.c` loop over the spatial grid and compute the values of the operator $\boldsymbol{\mathcal{E}}$ and its derivatives for each node, i.e., it loops over the lines of the system (1.12), building them sequentially. Recall from equation (1.7) that we use three grid points and two time levels to evaluate each block equation of the system. Some of this computation is done per grid point, i.e., some functions of $\mathbf{u}$ are evaluated independently at each point involved in the discretization. This is accomplished by the function `phys_f()`. Next, the values associated with each of the three grid points in the stencil for a given node, both the M state components and the functions evaluated in `phys_f()`, are given as input to `discrete_{tri,y}()`, which compute the block equation associated to that grid node.

The architecture implied by the interface of `phys_f()` serves all three goals presented in the beginning of this chapter, i.e., mathematical validity, computational efficiency, and source code clarity. Even so, implementing all mathematical formulæ that are deemed necessary, i.e, fully satisfying the requirement of mathematical validity, may render the code cumbersome and the computation costly. The implementations of `phys_f()` are therefore the foremost object of scrutiny in writing light and fast code, so as to satisfy the requirement of computational efficiency.

Further savings come from the buffering and reuse of all the values computed in `phys_f()`, which is called only once per grid node, without wasting either processor time (if, when computing the blocks corresponding to a given node, `phys_f()` was called for the said node and its two neighbours, the amount of computation would be threefold) or memory (as it would be if the buffers were not recycled).

The key point in the implementation of each physics is the implementation of `phys_f()` and `discrete_{tri,y()}`. The concepts behind these functions are:

- The three functions are expanded inline: they must be, because they are "called" for each grid node, at each time step, at least once. The proper use of the constants described in the following items ensures the code expansion is minimal, that all loops can be fully unrolled and that all conditional statements are optimized out at compile time.
- The first actual parameter `u` passed to `phys_f()` is an array of size `M`, containing the state vector, as indexed by `enum State`. The second actual parameter `jet` is an array of size `_JETSIZE`, defined as the last symbol of `enum Jet`. The symbols in `enum Jet` identify the positions in `jet` with the functions of the state variables to be computed in `phys_f()`.
- The third formal parameter is a binary mask with three relevant flags. The first flag indicates whether the mathematical functions relevant for computing the right hand side of system (1.12) must be computed, and is always true. The second flag is analogous for the left hand side of (1.12), and is not always true. Another way of interpreting these flags is that the first requests the zero–order derivatives, and the second flag requests the first–order derivatives.

  The third flag indicates whether `phys_f()` must write values which are state independent, e.g., whether a 4×4 matrix with only two nonzero values must be fully written or only the nonzero ($\mathbf{u}$–dependent) values must be written. Recall that the buffers are reused, so these constant values need to be set only once.
- In the calls to `phys_f()`, the third actual parameter is always a constant, so all conditional statements in `phys_f()` are resolved at compile–time by the optimizer.
- After buffers are passed to `phys_f()`, they contain all information required by `discrete_{tri,y}()`. When passed to `phys_f()`, the `u` buffer is input only, and the `jet` buffer output, and the actual grid position associated with the state stored in `u` and `jet` is not relevant. When the same buffers are passed to `discrete_{tri,y}()`, they are both input only, and the grid position is relevant.
- The symbols in `enum State` and `enum Jet` must be used consistently in all three functions to access the values stored in these buffers.

This architecture provides maximum efficiency if the actual implementation of these three methods is in full accordance with the concepts above. Nonoptimal or improper implementations will have correspondingly poor performance. Likewise, the referencing through enumerations, the separation of the computations into node–dependent (`phys_f()`) and stencil–dependent (`discrete_{tri,y}()`), provide for good readability without degrading the object code. Other points in coding style contribute as well: the members of the enumerations

`State`, `Jet` and `Stats` are prepended by underscores, and hard typecasts are used to allow for matrix notation access in portions of the `jet` vector corresponding to `dfdu`, `dhdu` and `dqdu`.

3.5.2. RCD Physics. All the models in Chapter 2 use the discretization given in 1.4, and are lumped into the sourcefiles `rcd*.*`. The code in `rcddiscrete.c` is largely shared by all these physics, and the code in `rcdphys.h` is very similar between any two physics. Because the models have so much in common, we discuss most implementation details using the quadratic model from 2.1.1 as an example: the other models are discussed in comparison to this one.

3.5.2.1. *General macros and conventions.* The physics described in Chapter 2 are identified by the following macros:

`QUAD`	Quadratic model from 2.1.1.
`COREY`	Corey's model from 2.1.2.
`SOLID_AKKUTLU`	Combustion model from 2.2.3.
`SOLID_GRIGORI`	Combustion model from 2.2.4.

The variable `PHYS`, defined at the beginning of the `makefile` to the lower-case versions of these strings, selects which physics should be compiled, by defining the appropriate macro and compiling the appropriate files.

Below `rcdphys.h:DOC_phys_vars`, there are disjoint sections conditioned by these macros for compilation. Each section defines the `M` and `M2` constants, and the `State`, `Jet` and `Stats` enumerations, along with the declarations of the physics dependent parameters. Some other macros are defined only in some physics, as explained below:

- If a model has $\mathbf{h}(\mathbf{u})$ other than $\mathbf{h}(\mathbf{u}) = \mathbf{u}$, then it must provide the implementation of $\mathbf{h}(\mathbf{u})$ (see below), and define the macro `NON_LINEAR_ACCUM`.
- If a model has $\mathbf{q}(\mathbf{u})$ other than $\mathbf{q}(\mathbf{u}) = 0$, then it must define the macro `HAS_SOURCE` and provide the implementation of $\mathbf{q}(\mathbf{u})$ (see below).
- If the model contains an equation that must be discretized using the Box scheme from 1.4.2, then it must define the macro `NBOX` with the number of equations to be discretized differently. In this case, the first `M-NBOX` equations are discretized using the Crank–Nicolson scheme, and the last `NBOX` of the total M equations are discretized with the Box scheme.

At the time of this draft, no stationary problem was implemented, neither any physics that uses the staggered grid. The latter are sketched in rcd2discrete.[hc]. Also, the discretizations in rcddiscrete.[hc] are for constant g (viscosity) matrices only. Non-constant g would be implemented in the same lines of NON_LINEAR_ACCUM

and HAS_SOURCE. Some code has been marked by the macro VARIABLE_G, which is not defined anywhere.

If any or both of the functions $\mathbf{h}(\mathbf{u})$ and $\mathbf{q}(\mathbf{u})$ are nonzero in the model, the macros `NON_LINEAR_ACCUM` and `HAS_SOURCE` should be defined accordingly. The symbols in each `enum Jet` are mnemonics for the values of the functions $\mathbf{f}$, (which is present in all RCD models), and $\mathbf{h}$, $\mathbf{q}$, if these should be defined. Compare the two declarations of `enum Jet` in the `QUAD` and `SOLID_AKKUTLU` sections of the code: the latter has symbols for both $\mathbf{h}$ and $\mathbf{q}$, in accordance with having the two associated macros defined.

After the definitions specific to each physics, come a number of variables used by all of them, seen in Table 3.5.

Variable		Corresponding object(s)
`double`	`visc[M2], visc_mult`	The viscosity matrix and its multiplier, as in equation (2.6).
`double`	`one_k, one_2k, h2, one_h, one_2h, one_h2`	Temporaries for $1/\delta x$, $1/2\delta x$, δx^2, $1/\delta t$, $1/2\delta t$ and $1/\delta t^2$.
`double`	`g[M2], g_h2[M2]`	The matrices $\mathbf{g}$ as in (2.6) and $\mathbf{g}/\delta t^2$.
`double`	`max_c_rate, max_r_rate, max_d_rate`	Maxima of convection, reaction and diffusion rate, reduced during the computation of `phys_f()` over the physical domain during the last Newton iteration.
`double`	`alpha, onemalpha, onemalpha_2, onemalpha_h, onemalpha_2h, onemalpha_h2, alpha_2, alpha_h, alpha_2h, alpha_h2, twoalphag_h2[M2]`	The `alpha` variable can be present in the input, but defaults to 0.5: it is the interpolation weight between the two time levels, as given in equation (1.15). The rest are temporaries similar to those above, but including the weight α.
`double`	`fixed_dt, cfl, reaction, diffusion, interp, reynolds`	**Most of these are legacy from the previous software: in particular,** `interp` **has been replaced by** `alpha`**.** `fixed_dt` is used in case the adaptive time step computation is overridden (see below).
`int`	`use_fixed, use_parabolic, visc_is_scalar_times_eye`	**The** `use_parabolic` **parameter is legacy from the previous software.** Nonnull `use_fixed` overrides the adaptive adaptive time step computation. `visc_is_scalar_times_eye` should be true if the off–diagonal entries of $\mathbf{g}$ are zero.

TABLE 3.5. RCD physics variables.

TODO: The "legacy" parameters above should eventually be removed from the input file, once the time adaptive time step calculation is properly formalized.

PARALLEL: *The variable* `max_c_rate` *is declared as* `threadprivate` *because only the convection rate is actually computed along each Newton iteration. The adaptive time stepping mechanism is incomplete. One would expect all three variables to be updated alongside, and thus all three should be local to each thread.*

3.5.2.2. *The actual physics equations.* Still in `rcdphys.h`, the definition of `phys_f()` for the `quad` physics illustrates syntax mapping to the equations in 2.1.1. Here, a few shorthands are defined, namely `f` and `dfdu`, the latter with a hard typecast to allow for matrix notation in the ensuing lines.

IMPORTANT: Notice that matrices in C are indexed contrary to Fortran and Matlab, so the symbols / indexes used to index the typecasted `dfdu` correspont to column and row, instead of the other way around.

TODO:

At the time of this draft, there is an implemented but only partially tested feature, defined by macros `STATS_*_FLUX`. The idea is to gather along with the time stats the values of the flux function at the boundaries.

Also, `phys_begin_diagnostics()` does not display any information particular to any physics other than `quad`, and there is an ad-hoc implementation of velocity initialization for the `solid_akkutlu` model in `phys_init()`.

3.5.2.3. *The discretization.* The discretizations outlined in 1.4.1 and 1.4.2 are overlaped in `rcddiscrete.h`, with most statements conditioned by the macros defined in `rcdphys.h`, differently for each physics. Because of this, the code is syntatically fragile, and should be modified with extreme caution. These functions are called for each grid point (i.e., for each block equation) and the correspondence between their formal parameters and mathematical objects is given at table 3.6: the mnemonics used are commonplace and should be self-explanatory.

3.5.2.4. *The system builders.* The functions `build_{y,tri}()` in `tridiag.[hc]` build the **whole** system (1.12), calling `discrete_{y,tri}` for each grid point successively. The function `build_tri()` is struturally identical to `build_y()`, therefore we describe the rationale of the latter only.

PARALLEL: *The function works over a partition of size $K \approx N/P$, i.e., the workload of this function is split in the case of parallel execution: in this section, we focus on the behaviour of the serial case, for which the formal parameters described in table 3.7 have unique values.*

The function is straightforward with a couple of noteworthy quirks: under `DOC_time_stat`, notice the call to zero out the buffer relative to the state at the ghost point. This is required to ensure that the accumulations in `phys_post()` will not be contaminated by data from the previous time level during parallel execution.

Parameter	Corresponding object(s)
`uim, ui, uip`	The state vector $\mathbf{u}$ at the positions x_{i-1}, x_i and x_{i+1}, respectively. In `build_y()`, these are at time t^{n-1}. In `build_tri()`, these are at time t^n.
`uim_jet, ui_jet, uip_jet`	The `jet` vectors filled in by calls to `phys_f`, corresponding to the state vectors `u*` above.
`upim, upi, upip`	The state vector $\mathbf{u}$ at the positions x_{i-1}, x_i and x_{i+1}, at time t^{n-1} (only in `build_tri()`).
`upim_jet, upi_jet, upip_jet`	The `jet` vectors filled in by calls to `phys_f`, corresponding to the state vectors `u*` above.
`y`	Part of the right hand side $\Xi(\mathbf{u}_i^n, \mathbf{u}_i^{n-1}) = \mathbf{F}(\mathbf{u}_i^n) + y(\mathbf{u}_i^{n-1})$ of system (1.12); it is output in `build_y()`, and input in `build_tri()`.
`A, B, C, Xi`	Right and left hand sides ($\mathbf{A}_i$, $\mathbf{B}_i$, $\mathbf{C}_i$ and Ξ_i of system (1.12).

TABLE 3.6. Formal parameters for the discretization functions.

Parameter	Corresponding object(s)	
`int`	`K, I, P`	Partition size, process rank and number of processes, respectively. In serial execution, `K = N`, `I = 0` and `P = 1`.
`double`	`sim_time, time_step`	Simulation time t^{n-1} and current time step δt — target time is $t^n = t^{n-1} + \delta t$.
`const double*`	`ghost_left_past, u_past, ghost_right_past`	Only `u_past` is used in serial execution: it contains the state vector $\mathbf{u}^{n-1}$.
`double*`	`y`	In serial execution, the whole vector y corresponding to part of the right hand side $\Xi(\mathbf{u}^n, \mathbf{u}^{n-1}) = \mathbf{F}(\mathbf{u}^n) + y(\mathbf{u}^{n-1})$.

TABLE 3.7. Formal parameters for the `build_y()` function.

PARALLEL: *In parallel execution, the ghost nodes for each partition are updated out of sync with the rest of the state vector. The correct data cannot be retrieved from them until the coupling system is solved.*

Also, the loop structure appears broken, as can be seen from the `for` and `if` statements. This is forced by the presence of (computational/index related) boundary conditions and the buffer recycling.

The recycled buffers passed to `phys_f()` are declared under `DOC_recycled`, initialized under `DOC_init`, and rotated under `DOC_rotate`; the next value for `uip` is determined under `DOC_move` as either a position inside the state vector, or the ghost point.

As previously stated, the function `build_tri()` is structurally identical. The only relevant difference is the `TWOLEVEL` macro, which conditions the compilation of statements that implement the usage of the previous time level in the computation of the left hand side. **At**

the time of this draft, no model implemented uses this feature. It implies the computation of `phys_f()` ONE EXTRA TIME for each grid point.

3.6. Using the program

Understanding the implementation, as described so far in this chapter, is not required in order to run the program, although it does make it clearer by exposing all inner mechanisms. In the following subsections, the compilation, execution and output visualization are explained, with proper references to the internals previously detailed.

3.6.1. Compiling the program. The whole NITRO package, including source code and documentation, should be installed on a GNU/GCC compatible platform: any UNIX brand or clone should work, including the Cygwin environment for Win32. The root directory contains a makefile, which is simple as far as makefiles go: it determines the host architecture from the environement variable `RPHOSTTYPE`, which can be set to `mingw32` for Cygwin - any other value defaults to UNIX/GCC.

The first line of `makefile` defines the `PHYS` variable, which should be set to lower case strings corresponding to the name of the physics to be compiled (e.g. `quad` or `solid_akkutlu`).

The `make` command should be given with a target name parameter — typing simply `make` with give an error message prompting for one such parameter. Valid targets are `clean`, `gdb`, `serial`, `mpi`, `openmp`. The `serial` target generates a serial executable, whereas the `mpi` and `openmp` targets generate the corresponding parallel versions. The `gdb` target is the same as `serial`, but with debug information. All executables are built into the `bin` directory, and object files are left in the NITRO root directory. The `clean` target removes temporary objects, but does not remove any executables.

Non-experts should always run `make clean` before building an executable.

3.6.2. Running the program. The program takes one parameter from the command line: the name of the input file. Sample input files for each implemented physics are given in the `test` directory.

`PHYS` variable in `makefile`	Macro in source code	Input file in `test` directory
`quad`	`QUAD`	`quadtest.m`
`solid_akkutlu`	`SOLID_AKKUTLU`	`satest.m`
`solid_grigori`	`SOLID_GRIGORI`	`sotest.m`

TABLE 3.8. Compile and run information for the physics currently implemented.

For completion, the file `quadtest.m` is reproduced below and crossreferenced to previous sections of this document.

```
a1 = 3; b1 = 0; c1 = 1; d1 = 0; e1 = 2;
a2 = 0; b2 = 1; c2 = 0; d2 = 0; e2 = 0;
```

Coefficients of the flux functions (2.5) on page 22.

```
visc_is_scalar_times_eye = 1;
visc = [ 1 0 ; 0 1];
visc_mult = 3;
```

Viscosity matrix (2.6) from page 22 as given on table 3.5 on page 47. Notice that `visc` and `visc_is_scalar_times_eye` are consistent.

```
stop_time = 500;
stop_step = 10000;
print_by_step = 0;
print_time_interval = 10;
print_step_interval = 10;
```

Simulation advances to time $t = 500$; a maximum of 10000 time steps is allowed. Output is generated at constant time intervals of $\delta t = 10$ (the value of `print_step_interval` is ignored). See sections 3.3 and 3.4.

```
bc_left = [0 -2];
bc_right = [-.2 -2];
```

Boundary conditions are Dirichlet, constant in time, with $\mathbf{u}_a = \{0, -2\}$ and $\mathbf{u}_b = \{-0.2, -2\}$. See 3.2.2.3 on page 36.

```
grid_lim = [-1000 1000];
n_cells = 1000;
```

The physical domain is $-1000 \leq x \leq 1000$, discretized into 1000 cells. See subsection 3.2.2.2 on page 36.

```
data_type = 1;
data = [20 0 -2; 20 -.2 -2];
```

Data is piecewise linear, with a jump point at $x = 20$: the inital state vector is $\mathbf{u} = \{0, 2\}$ for $x \leq 20$ and $\mathbf{u} = \{-0.2, -2\}$ for $x > 20$. The Dirichlet boundary conditions are used to fill in, i.e., the data is linearly interpolated from the jump condition at $x = 20$ to the boundary values. See subsection 3.2.2.4 on page 38, and table 3.2 on page 35.

```
use_fixed = 0;
fixed_dt = 0;
use_parabolic = 0;
cfl = 0.8;
reaction = 0.1;
diffusion = 1;
```

```
interp = 0.5;
```

Time step is adaptive (`use_fixed` is zero), so `fixed_dt` is ignored. See table 3.5 on page 47.

```
plot_x = [-1000 1000];
plot_lim = [-2 1; -5 0];
```

When visualizing the output (see next section), the initial ranges will be $x \in [-1000, 1000]$, $s_w \in [-2, 1]$ and $s_o \in [-5, 0]$. These do not affect the calculations, see subsection 3.2.1 on page 32.

3.7. Post–processing

The binary output generated by the simulator is suitable for efficient loading by any independent application. The `result.m` MATLAB script, located in the root of the NITRO package, provides basic data visualization and comparison facilities.

The script runs on MATLAB 7.0 or higher. Its invocation creates an interface window with the following contents:

- **Load 1st Set...** This button opens a dialog for choosing an output file from the simulator, from which the data set is loaded. The rest of the interface is disabled until one such data set is loaded. Using this button again discards the previously loaded data set or sets (see below).
- **Load 2nd Set...** A second data set may be opened afterwards for comparison with the first. (Re)using this function does NOT discard the primary data set, which is used as reference for comparison. The second data set must match the previous regarding physics and program version, but may differ in all other aspects.
- **Tile views** Tries to resize and position all open graphics windows to fill the screen.
- **Limits** Opens a dialog where the user can set the plot limits for x and each component in $\mathbf{u}$. The dialog also allows the user to automatically set these limits, based on maxima and minima values of the actual data. The current x range is used when generating restart files (see **Save...** below).
- **Prompt** Gives the user the MATLAB debug prompt, where he can inspect the data using MATLAB commands. The data is available inside the `D` structure: the data sets are stored in `D.A` and `D.B`. Most of the contents of these structures correspond to those in tables 3.3 and 3.4, and some are postprocessed from those: all are described in table 3.9.
- **Save...** Opens a dialog for choosing a file name (`.m`) for writing a new input (restart) file, based on the input file of the primary data currently loaded, but using the state for the currently visible time (see **Time level** below) as initial data, and the currently visible x range for grid limits. **At the time of this draft, this feature was partially implemeted and tested, in particular regarding non-constant / non-Dirichlet boundary conditions**.

- **Popup view for:** This list the available data views, namely each state component against x and against each other. Double clicking opens the figure window.
- **Time level** This slider controls for which time level the data is displayed, on the graphical plots and on the text output below. The currently displayed data is used when creating a restart file (see **Save...** above).

Record type[size]	Data
`version`	String identifying the simulator version, e.g. `"NITRO001"`.
`phys_version`	String identifying the physics version, e.g. `"QUAD001"`.
`N`	Number of grid points N.
`M`	Number of state variables M.
`Nts`	Number `_Nts` of values in `time_stats`.
`phys_comp_names`	Strings identifying the components in the state vector.
`time_stat_names`	Strings identifying the values in `time_stats`.
`grid_lim`	Physical domain boundaries x_a and x_b.
`x`	Spatial grid x.
`orig_plot_x`	Initial visualization interval in physical domain.
`orig_plot_lim`	Initial visualization interval for each state component.
`plot_x`	Current visualization interval in physical domain.
`plot_lim`	Current visualization interval for each state component.
`input_file`	The original input file, without comments.
`T`	The simulation times.
`step`	The corresponding time levels.
`time_stats`	The corresponding `time_stats` for each time level.
`nT`	The total number of time levels in this file.
`u`	The state vectors for each position and time ($M \times N \times$`nT`).
`lims`	The actual maxima and minima of the state components over the domain, used for automatically fitting the plot intervals to the data.
`filename`	The file name from which this data structure was read.

TABLE 3.9. MATLAB data structure.

CHAPTER 4

Parallel Implementation

We now discuss how the operations previously described are implemented in a parallel environment with P processing elements. Put simply, data parallelism is employied in the Newton solver. The state vector is split over the processing elements, in equal or near equal contiguous shares, i.e., the physical domain is partitioned in P shares of size K_p such that $K_p \approx N/P$. The computation of the coefficients of the linear system (1.12) at each Newton iteration is distributed accordingly, and so is its solution.

The distribution of the computation of the coefficients of (1.12) is a natural consequence of the distribution of the state vector, described in 4.2.1: each processing element computes the block equations corresponding to its partition: some information must be duplicated at the boundaries between partitions, i.e., the two equations at the boundary between neighbour partitions both depend on the value of the state at the boundary node(s). On the other hand, the solution of the linear system implies global exchange of information, and a completely different approach to its solution is required.

Each Newton iteration consists of these two steps, i.e., computing the system's coefficients and solving the system, so they are performed alternatingly in the Newton loop. The two steps are comparable in terms of computational load. In the description of the serial implementation, we deliberately overlooked the solution of each linear system, which is done with the sequential algorithm presented in Appendix A. It is quite trivial to understand, and fully encapsulated by two functions in `block_lu.h`, namely `block_lu_{decomp,solve}()`.

Performing an efficient parallel solution requires embedding the linear solver into the Newton loop: in the next section 4.1, we present the parallel algorithm for the solution of block tridiagonal systems that is implemented in `solvermain.c`. We introduce the algorithm before we describe its implementation, as it poses constraints on many decisions in the implementation of the simulator.

4.1. Parallel Solution of Block Tridiagonal Systems

The parallel algorithm for the solution of block tridiagonal systems is a two level construct based on the sequential algorithm presented in Appendix A, i.e., we assume that the sequential solution of a block linear system is known procedure.

Consider the problem of solving (1.12) with $N = KP + (P-1)$. We can group the equations into P blocks of size $KM \times KM$ as follows:

$$\mathbf{A} = \left[\begin{array}{ccccc:ccc} A_1 & C_1 & & & & & & \\ B_2 & A_2 & C_2 & & & & & \\ & \ddots & \ddots & \ddots & & & & \\ & & B_{K-1} & A_{K-1} & C_{K-1} & & & \\ & & & B_K & A_K & C_K & & \\ \hdashline & & & & B_{K+1} & A_{K+1} & C_{K+1} & \\ & & & & & B_{K+2} & A_{K+2} & C_{K+2} \\ & & & & & & \ddots & \ddots & \ddots \end{array}\right]; \tag{4.1}$$

in the ensuing discussion, we write this partitioning as

$$\left[\begin{array}{ccccccccccc} \mathbf{A}_1 & & & & & & & & & & \\ & \mathbf{\Theta}_1 & & & & & & & & & \\ & \mathbf{\Gamma}_1 \quad \mathbf{\Lambda}_1 \quad \mathbf{\Phi}_1 & & & & & & & & & \\ & \mathbf{\Omega}_1 & & & & & & & & & \\ & & \mathbf{A}_2 & & & & & & & & \\ & & & \ddots & & & & & & & \\ & & & & \mathbf{A}_{P-1} & & & & & & \\ & & & & & & \mathbf{\Theta}_{P-1} & & & & \\ & & & & & \mathbf{\Gamma}_{P-1} & \mathbf{\Lambda}_{P-1} & \mathbf{\Phi}_{P-1} & & & \\ & & & & & & \mathbf{\Omega}_{P-1} & & & & \\ & & & & & & & & & \mathbf{A}_P & \end{array}\right] \left[\begin{array}{c} \mathbf{x}_1 \\ \\ \boldsymbol{\xi}_1 \\ \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_{P-1} \\ \\ \boldsymbol{\xi}_{P-1} \\ \\ \mathbf{x}_P \end{array}\right] = \left[\begin{array}{c} \mathbf{y}_1 \\ \\ \boldsymbol{\psi}_1 \\ \\ \mathbf{y}_2 \\ \vdots \\ \mathbf{y}_{P-1} \\ \\ \boldsymbol{\psi}_{P-1} \\ \\ \mathbf{y}_P \end{array}\right], \tag{4.2}$$

with

$$\begin{array}{l} \mathbf{A}_p \in \mathbb{R}^{KM \times KM}, \\ \mathbf{x}_p,\ \mathbf{y}_p \in \mathbb{R}^{KM}, \\ \mathbf{\Lambda}_p,\ \mathbf{\Theta}_p,\ \mathbf{\Gamma}_p,\ \mathbf{\Phi}_p,\ \mathbf{\Omega}_p \in \mathbb{R}^{M \times M}, \text{ and} \\ \boldsymbol{\xi}_p,\ \boldsymbol{\psi}_p \in \mathbb{R}^{M}. \end{array}$$

The full system given in equation (4.2) is solved in three stages: first, each of the P systems of K block equations $\mathbf{A}_p\mathbf{x}_p = \mathbf{y}_p$ is solved independently, and the solutions obtained are functions of the unknowns $\boldsymbol{\xi}_p$; second, we construct from these solutions a new, smaller system of $P-1$ block equations, and determine the $\boldsymbol{\xi}_p$ unknowns; finally, we assemble the solutions of the two previous stages to find the solution of the full system.

We now formalize the first step: introducing the $KM \times M$ matrices

$$\mathbf{U} = \begin{bmatrix} \mathbf{I} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \end{bmatrix} \quad \text{and} \quad \mathbf{V} = \begin{bmatrix} \mathbf{0} \\ \vdots \\ \mathbf{0} \\ \mathbf{I} \end{bmatrix},$$

we write

$$\begin{array}{ll}
\mathbf{A}_1\mathbf{x}_1 + \boldsymbol{\xi}_1\boldsymbol{\Theta}_1\mathbf{V} = \mathbf{y}_1 & \rightarrow \mathbf{x}_1 = \mathbf{A}_1^{-1}(\mathbf{y}_1 - \boldsymbol{\xi}_1\boldsymbol{\Theta}_1\mathbf{V}), \\
\boldsymbol{\xi}_{p-1}\boldsymbol{\Omega}_{p-1}\mathbf{U} + \mathbf{A}_p\mathbf{x}_p + \boldsymbol{\xi}_p\boldsymbol{\Theta}_p\mathbf{V} = \mathbf{y}_p & \rightarrow \mathbf{x}_p = \mathbf{A}_p^{-1}(\mathbf{y}_p - \boldsymbol{\xi}_{p-1}\boldsymbol{\Omega}_{p-1}\mathbf{U} - \boldsymbol{\xi}_p\boldsymbol{\Theta}_p\mathbf{V}), \quad p = 2, \ldots, P-1, \\
\boldsymbol{\xi}_{P-1}\boldsymbol{\Omega}_{P-1}\mathbf{U} + \mathbf{A}_P\mathbf{x}_P = \mathbf{y}_P & \rightarrow \mathbf{x}_P = \mathbf{A}_P^{-1}(\mathbf{y}_P - \boldsymbol{\xi}_{P-1}\boldsymbol{\Omega}_{P-1}\mathbf{U}),
\end{array} \tag{4.3}$$

and define the objects $\overline{\mathbf{y}}_p$, $\overline{\mathbf{U}}_p$, $\overline{\mathbf{V}}_p$ as

$$\mathbf{A}_p\overline{\mathbf{y}}_p = \mathbf{y}_p, \qquad \mathbf{A}_p\overline{\mathbf{U}}_p = \mathbf{U}\boldsymbol{\Omega}_{p-1} \qquad \text{and} \qquad \mathbf{A}_p\overline{\mathbf{V}}_p = \mathbf{V}\boldsymbol{\Theta}_p, \tag{4.4}$$

so they can be computed using any method such as LU factorization of $\mathbf{A}_p$ and back substitution for each $\mathbf{y}_p$, $\mathbf{U}$ and $\mathbf{V}$ right hand side.

Now the solutions for $\mathbf{x}_p$ in (4.3) can be written as functions of $\boldsymbol{\xi}_p$, i.e.,

$$\begin{array}{l}
\mathbf{x}_1 = \overline{\mathbf{y}}_1 - \overline{\mathbf{V}}_1\boldsymbol{\xi}_1, \\
\mathbf{x}_p = \overline{\mathbf{y}}_p - \overline{\mathbf{U}}_p\boldsymbol{\xi}_{p-1} - \overline{\mathbf{V}}_p\boldsymbol{\xi}_p, \quad p = 2, \ldots, P-1, \\
\mathbf{x}_P = \overline{\mathbf{y}}_P - \overline{\mathbf{U}}_P\boldsymbol{\xi}_{P-1},
\end{array} \tag{4.5}$$

The second step is to use these solutions to determine the unknowns $\boldsymbol{\xi}_p$. For each row of (4.2) corresponding to a $\boldsymbol{\psi}_p$, we use the $\mathbf{x}_p$ defined in (4.5) to obtain an equation of the form

$$\boldsymbol{\Gamma}_p(\mathbf{V}^{\mathrm{T}}\mathbf{x}_p) + \boldsymbol{\Lambda}_p\boldsymbol{\xi}_p + \boldsymbol{\Phi}_p(\mathbf{U}^{\mathrm{T}}\mathbf{x}_{p+1}) = \boldsymbol{\psi}_p, \quad p = 1, ..., P-1, \tag{4.6}$$

Each solution $\mathbf{x}_p$ is a function of $\boldsymbol{\xi}_p$ and $\boldsymbol{\xi}_{p-1}$, except for $\mathbf{x}_1$ and $\mathbf{x}_P$. Applying the expressions for $\mathbf{x}_p$ from (4.5) to (4.6), we obtain

$$\begin{array}{l}
\boldsymbol{\Gamma}_1\big[\mathbf{V}^{\mathrm{T}}(\overline{\mathbf{y}}_1 - \overline{\mathbf{V}}_1\boldsymbol{\xi}_1)\big] + \boldsymbol{\Lambda}_1\boldsymbol{\xi}_1 + \boldsymbol{\Phi}_1\big[\mathbf{U}^{\mathrm{T}}(\overline{\mathbf{y}}_2 - \overline{\mathbf{U}}_2\boldsymbol{\xi}_1 - \overline{\mathbf{V}}_2\boldsymbol{\xi}_2)\big] = \boldsymbol{\psi}_1, \\
\\
\boldsymbol{\Gamma}_p\big[\mathbf{V}^{\mathrm{T}}(\overline{\mathbf{y}}_p - \overline{\mathbf{U}}_p\boldsymbol{\xi}_{p-1} - \overline{\mathbf{V}}_p\boldsymbol{\xi}_p)\big] + \boldsymbol{\Lambda}_p\boldsymbol{\xi}_p \\
\qquad + \boldsymbol{\Phi}_p\big[\mathbf{U}^{\mathrm{T}}(\overline{\mathbf{y}}_{p+1} - \overline{\mathbf{U}}_{p+1}\boldsymbol{\xi}_p - \overline{\mathbf{V}}_{p+1}\boldsymbol{\xi}_{p+1})\big] = \boldsymbol{\psi}_p, \quad p = 2, \ldots, P-2, \\
\\
\boldsymbol{\Gamma}_{P-1}\big[\mathbf{V}^{\mathrm{T}}(\overline{\mathbf{y}}_{P-1} - \overline{\mathbf{U}}_{P-1}\boldsymbol{\xi}_{P-2} - \overline{\mathbf{V}}_{P-1}\boldsymbol{\xi}_{P-1})\big] + \boldsymbol{\Lambda}_{P-1}\boldsymbol{\xi}_{P-1} \\
\qquad + \boldsymbol{\Phi}_{P-1}\big[\mathbf{U}^{\mathrm{T}}(\overline{\mathbf{y}}_P - \overline{\mathbf{U}}_P\boldsymbol{\xi}_{P-1})\big] = \boldsymbol{\psi}_{P-1}.
\end{array} \tag{4.7}$$

from which we derive the coefficients of a new tridiagonal system of $P-1$ equations

$$\begin{bmatrix}
\overline{\boldsymbol{\Lambda}}_1 & \overline{\boldsymbol{\Phi}}_1 & & & \\
\overline{\boldsymbol{\Gamma}}_2 & \overline{\boldsymbol{\Lambda}}_2 & \overline{\boldsymbol{\Phi}}_2 & & \\
& \ddots & \ddots & \ddots & \\
& & \overline{\boldsymbol{\Gamma}}_{P-2} & \overline{\boldsymbol{\Lambda}}_{P-2} & \overline{\boldsymbol{\Phi}}_{P-2} \\
& & & \overline{\boldsymbol{\Gamma}}_{P-1} & \overline{\boldsymbol{\Lambda}}_{P-1}
\end{bmatrix}
\begin{bmatrix} \boldsymbol{\xi}_1 \\ \boldsymbol{\xi}_2 \\ \vdots \\ \boldsymbol{\xi}_{P-2} \\ \boldsymbol{\xi}_{P-1} \end{bmatrix}
=
\begin{bmatrix} \overline{\boldsymbol{\psi}}_1 \\ \overline{\boldsymbol{\psi}}_2 \\ \vdots \\ \overline{\boldsymbol{\psi}}_{P-2} \\ \overline{\boldsymbol{\psi}}_{P-1} \end{bmatrix}, \tag{4.8}$$

where

$$\begin{array}{ll}
\overline{\boldsymbol{\Lambda}}_p = \boldsymbol{\Lambda}_p - \boldsymbol{\Gamma}_p(\mathbf{V}^{\mathrm{T}}\overline{\mathbf{V}}_p) - \boldsymbol{\Phi}_p(\mathbf{U}^{\mathrm{T}}\overline{\mathbf{U}}_{p+1}), & p = 1, \ldots, P-1, \\
\overline{\boldsymbol{\Gamma}}_p = -\boldsymbol{\Gamma}_p(\mathbf{V}^{\mathrm{T}}\overline{\mathbf{U}}_p), & p = 2, \ldots, P-1, \\
\overline{\boldsymbol{\Phi}}_p = -\boldsymbol{\Phi}_p(\mathbf{U}^{\mathrm{T}}\overline{\mathbf{V}}_{p+1}), & p = 1, \ldots, P-2, \\
\overline{\boldsymbol{\psi}}_p = \boldsymbol{\psi}_p - \boldsymbol{\Gamma}_p(\mathbf{V}^{\mathrm{T}}\overline{\mathbf{y}}_p) - \boldsymbol{\Phi}_p(\mathbf{U}^{\mathrm{T}}\overline{\mathbf{y}}_{p+1}), & p = 1, \ldots, P-1.
\end{array} \tag{4.9}$$

The multiplications involving $\mathbf{U}$ and $\mathbf{V}$ in equation (4.9) are trivial: they only indicate that the first or last block of the vectors $\overline{\mathbf{U}}_p$ or $\overline{\mathbf{V}}_p$ must be isolated, respectively. The third step

is trivial as well: once the system (4.8) is solved, the solution of the full system is computed from the expressions in (4.5).

4.2. Parallel Iterative Newton Solver

In this section we fill in the gaps left in Chapter 3, regarding parallelism–specific data structures and procedures: we replicate the layout previously employed.

The code can be compiled in two ways: one is using shared–memory/thread–based parallelism through OpenMP, the other is using distributed–memory/cluster–based parallelism through MPI. This choice is controled by defining one of the two (mutually exclusive) macros `WITH_OPENMP` and `WITH_MPI`. Large sections of the code in `solvermain.c` are conditioned by these, and very little of the parallel specific code is shared by both models. It essence, `solvermain.c` contains three different programs overlapped, one serial and two parallel. They are, however, completely equivalent: all three generate the exact same results when executed on a single processor, and the two parallel codes generate the exact same results if executed on the same number of processors. As far as the operations in the algorithm described in the previous section go, all are performed in the same sequence, on the same data.

It is important to point out that the code included in the MPI version is executed on all processes, so if a section of code must be executed only (or differently) by process `I==0`, there is a `if (I) {...` or `if (!I){...` conditioning it. The code included in the OpenMP version, on the other hand, is executed only by process `I==0` unless there is a `#pragma omp parallel` directive preceding a block–statement, in which case the program forks. The two programming models are fundamentally different, and we have used each of them in the most natural way, which happen to be diametrically opposite: in MPI, parallel execution takes place unless otherwise enforced; in OpenMP, serial execution takes place unless otherwise enforced.

4.2.1. Data Objects. In the case of parallel solution, each process solves a local linear system with multiple right hand sides, as in equation (4.4). The next solution step is performed by a single process, which uses the solutions obtained by every other processor to solve the coupling system (4.8).

All the data objects used in the serial code are used in the parallel versions as well; however, their size and contents differ as can be seen in the memory allocation statements under `DOC_memory_allocation`. In particular, the objects declared under `DOC_state_vars` and `DOC_solver_vars` have sizes involving `M` and `N` in the serial code, while in the parallel versions the sizes are functions of `M` and `K`.

Many other data objects exist only in the parallel versions. The coupling system is obviously common to the two parallel implementations, and the data objects associated

with it are in the first `#if` block under `DOC_parallel_vars`. These are required both in the MPI and OpenMP versions. Other variables are specific to either programming model.

To setup systems (4.4) and make the solutions available to each process is trivial in the OpenMP version, because all processes share the same memory space. In the MPI version, the objects computed from (4.4) must be explicitly aggregated on one node, which builds and solves system (4.8) and then broadcasts the solution to all other processes. To do so, it uses the data structures and buffers declared in the MPI–specific `#ifdef` block, namely `mpi_{bcast,agg}_buf`. Other MPI–specific variables are required by the output routine, and are explained in 4.4.

There are two points of the Newton iteration where data must be reduced from all processes. The first is after the whole Jacobian matrix has been evaluated, i.e., when the global values `time_stats` can be determined from those local to each process. The second is at the end of each Newton iteration, when the norms of the residues are evaluated and tested against the stopping criteria. Both implementations use the space in `reduce_buf` for these operations.

4.2.2. Input Data Loading and Processing. The allocation of the MPI–specific objects in calls under `DOC_memory_allocation` is conditioned by the value of `I`, i.e., the process `I==0`, which performs all serial operations, requires extra storage for aggregating and reducing data, as well as managing comunication will all other processes. Some objects are allocated with different size on worker processes, others are not allocated at all. The OpenMP version is not executing in parallel at this point, so the memory is allocated by process `I==0`.

The input file is parsed and loaded by all processes in the MPI version, so in both parallel programs the raw input data is available to all processes. In 3.2.2 we singled out the lines under the tag `solvermain.c:DOC_data_init`, responsible for initializing the simulation data in the serial version. These calls are the ones included in the MPI version as well, only the values of `K` and `I` will determine which part of the data to process (in the serial version, `I==0` and `K==N`).

In the OpenMP version some extra initialization is necessary, which is perfomed by the code in the `#ifdef WITH_OPENMP` preceding tag `DOC_data_init`. For each thread, there is one copy of each global data object declared as `threadprivate`, i.e., although global in scope, they are not shared. These variables are pointers to (shared) memory, and their initialization is done in two steps. The first was memory allocation, already described, which is done by the master thread (`I==0`) before any `parallel` construct appears. The second part of the initialization is at the first `parallel` fork, immediately after the call to `omp_set_dynamic()`: this call is required for `threadprivate` static data to be coherent. The

`copyin` clause initializes each thread's copy to the value of the master thread, then offsets are added based on `I`. Now the pointer in each thread points to the beginning of the local section of the shared arrays.

The next calls are equivalent to those under `DOC_data_init`, with two small differences. First, threads other than the master copy the value of `ghost_left_now` into the correct positions of `u_now` (recall that the pointers of all threads point to the same arrays, only at different offsets). Second, while the MPI and serial versions use `time_stats` directly as buffer for `phys_get_local_time_stats`, OpenMP uses the reduction buffer and reduces it. Likewise, the `#ifdef WITH_MPI` block that follows performs aggregation, reduction and broadcast of `time_stats`. At this point, all processes have the same contents in `time_stats`, in both MPI and OpenMP versions.

4.3. Parallel Time Evolution

In the serial version, the system was fully solved by the two lines under `DOC_local_tridiag`. Now, the mathematical objects and operations of the algorithm presented in 4.1 can be seen in the lines before and after this tag.

The first step of the parallel solution, i.e., the computation of the objects in (4.4), is done in both parallel versions; however, the MPI program copies the results into the `mpi_agg_buf` for later aggregation. Notice that local `time_stats` are collected for reduction in both versions. This section executes in parallel.

The second step of the parallel solution is the serial step, i.e., setting up and solving the system (4.8). This requires data from all processes to be available to the `I==0` process. The MPI code does this with explicitly, directly under `DOC_coupling_sys`.

After the `ifdef WITH_MPI` conditioned aggregation, there is a serial section of code (no `#pragma parallel`, but a `if(!I)` on top of it). The whole second step of the algorithm in 4.1 is performed between the calls to `{start,end}_timer(SERIAL)`.

Once (4.8) is solved, all grid points have been processed and the local values of `time_stats` can be reduced. This is done slightly differently in each of the three alternative compilations, as seen in the code under `DOC_reduce`. In the MPI version, these values and the solution in `psibar` have to be broadcast to all processes, just like the data had to be aggregated before, so this opportunity is used to broadcast the `stop_flag` and (reduced) `time_stats`. Once `time_stats` is updated, the local representations are reset for the next Newton iteration.

The current implementation sets `stop_flag` consistently in each process, so this broadcasting is redundant. It may become useful if one some stopping criterion based on local data is adopted, or if interactive (but graceful) exit is implemented.

The values in `psibar` are the solution of the coupling lines of the full system, so these values contain the $\delta\mathbf{u}$ corresponding to the internal "ghost" points stored in `ghost_{left,right}`.

The code following `DOC_residual_reset` initializes the residual accumulators, and applied the Newton update for the coupling lines, which is replicated on neighbor processors. Next, under `DOC_full_solution`, the local systems are solved; under `DOC_newton_update` the rest of the state vector `u_now` is updated, and then the Newton residues collected at each process are aggregated, reduced and broadcast, so the call to `maybe_stop_newton()` is performed by each process on consistent data, i.e., `newton_residues` contains the same values for all processes.

4.4. Delayed Output

Output is performed by process `I==0` alone, which owns the output file. In the serial and OpenMP versions, the state vector is fully available to process `I==0` at any given moment, so the code in `maybe_print_state()` included in these compilations is straightforward and simple. However, in the MPI version, the variable `u_now` does not contain the whole state vector at any moment: it does not even have storage space to do so.

The MPI version performs delayed output: the first call to `maybe_print_state()` sets up message passing to transfer the initial values of `u_now` local to each process to the `u_gather` buffer of process `I==0`; it does NOT write a time record to the output file, only the header. Subsequent calls to `maybe_print_state()` write the time record that was setup for output in the PREVIOUS call, and sets up message passing to gather the current values of `u_now`, local to each process, into the `u_gather` buffer.

This gathering and flushing can be seen clearly under the tags `DOC_output_gather` and `DOC_delayed_output`, but one important point is rather obscure. At the end of the simulation, the final state must be printed, and printed only once. The big clause of the `if` at the function's beginning ensures this condition in the serial and OpenMP versions, as explained in 3.4. Notice that one term is dropped from the logical expression in the MPI compilation, namely the one that checks whether the current step was the one printed in the last call. This is because the function will be called twice at the end of the simulation recursively: the call takes place at the end of the `I==0` branch under `DOC_output_gather`. The call is done with the actual argument `stop_flag` set to two, to force the function to return at the point under `DOC_force_return`.

Bibliography

[1] I. Y. Akkutlu and Y. C. Yortsos. The effect of heterogeneity on in-situ combustion: The propagation of combustion fronts in layered porous media. *J. Pet. Tech.*, 54(6):56–56, 2002.

[2] I.Y. Akkutlu and Y.C. Yortsos. The dynamics of in-situ combustion fronts in porous media. *Combustion and Flame*, 134:229–247, 2003.

[3] A. Azevedo, D. Marchesin, B. J. Plohr, and K. Zumbrun. Capillary instability in models for three-phase flow. *Zeitschrift fur Angewandte Mathematik und Physik*, 53:713–746, 2002.

[4] S. Buckley and M. Leverett. Mechanisms of fluid displacement in sands. *Trans. AIME*, 146:187–196, 1942.

[5] g. chapiro. *Singular Perturbation Applied to Combustion Waves in Porous Media (in Portuguese)*. PhD thesis, IMPA, 2005.

[6] g. chapiro, a. a. mailybaev, d. marchesin, and a.j. souza. Singular perturbation in combustion waves for gaseous flow in porous media. In *XXVI CILAMCE*, 2005.

[7] A. Corey, C. Rathjens, J. Henderson, and M. Wyllie. Three–phase relative permeability. *Trans. AIME*, 207:349–351, 1956.

[8] E. Isaacson and H.B. Keller. *Analysis of Numerical Methods*. Dover Publications, 1994.

[9] D. Marchesin and B. J. Plohr. Wave structure in WAG recovery. *SPE Journal*, 6:209–219, 2001.

[10] a. j. souza, d. marchesin, and i. y. akkutlu. Wave sequences for solid fuel adiabatic in-situ combustion in porous media. *Comp. Applied Math*, 25(1):27–54, 2006.

[11] J. C. Strikwerda. *Finite difference schemes and partial differential equations*. Wardsworth & Brooks/Cole, 1989.

APPENDIX A

Serial Solution of Block Tridiagonal Systems

Consider a system of N block equations with coefficient matrix $\mathbf{A}$, unknown vector $\mathbf{x}$ and given right hand side $\mathbf{y}$, where the $\mathbf{A}_{ij}$ entries of $\mathbf{A} = [\mathbf{A}_{ij}]$ are $M \times M$ matrices themselves, so the total number of (scalar) equations in the system is NM, i.e.,

$$\mathbf{A}\mathbf{x} = \mathbf{y}, \quad \mathbf{A} \in \mathbb{R}^{NM \times NM}, \quad \mathbf{x}, \mathbf{y} \in \mathbb{R}^{NM}, \tag{A.1}$$

Recall that the only non–zero entries of $\mathbf{A}$ are those of the main diagonal and its two neighbors. Following the notation of [**8**], the matrix $\mathbf{A}$ is of the form

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 & \mathbf{C}_1 & & & \\ \mathbf{B}_2 & \mathbf{A}_2 & \mathbf{C}_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \mathbf{B}_{N-1} & \mathbf{A}_{N-1} & \mathbf{C}_{N-1} \\ & & & \mathbf{B}_N & \mathbf{A}_N \end{bmatrix}. \tag{A.2}$$

The LU factorization, when applied to this type of matrix, is much simplified: it becomes a linear algorithm, whereas it is quadratic for a general matrix. Trivially, one obtains

$$\mathbf{A} = \mathbf{L}\mathbf{U} = \begin{bmatrix} \overline{\mathbf{A}}_1 & & & & \\ \mathbf{B}_2 & \overline{\mathbf{A}}_2 & & & \\ & \ddots & \ddots & & \\ & & \mathbf{B}_{N-1} & \overline{\mathbf{A}}_{N-1} & \\ & & & \mathbf{B}_N & \overline{\mathbf{A}}_N \end{bmatrix} \begin{bmatrix} \mathbf{I} & \overline{\mathbf{C}}_1 & & & \\ & \mathbf{I} & \overline{\mathbf{C}}_2 & & \\ & & \ddots & \ddots & \\ & & & \mathbf{I} & \overline{\mathbf{C}}_{N-1} \\ & & & & \mathbf{I} \end{bmatrix},$$

with

$$\begin{array}{l} \overline{\mathbf{A}}_1 = \mathbf{A}_1, \ \overline{\mathbf{C}}_1 = \overline{\mathbf{A}}_1^{-1}\mathbf{C}_1 \\ \left\{ \begin{array}{l} \overline{\mathbf{A}}_i = \mathbf{A}_i - \mathbf{B}_i\overline{\mathbf{C}}_{i-1} \\ \overline{\mathbf{C}}_i = \overline{\mathbf{A}}_i^{-1}\mathbf{C}_i \end{array} \right. , \qquad i = 2, \ldots, N-1, \\ \overline{\mathbf{A}}_N = \mathbf{A}_N - \mathbf{B}_N\overline{\mathbf{C}}_{N-1}. \end{array} \tag{A.3}$$

The solution of system (A.1) can be computed through standard back substitution, solving

$$\mathbf{L}\mathbf{z} = \mathbf{y} \quad \text{and then} \quad \mathbf{U}\mathbf{x} = \mathbf{z},$$

so

$$\begin{array}{rcll} \mathbf{z}_1 & = & \overline{\mathbf{A}}_1^{-1}\mathbf{y}_1 & \\ \mathbf{z}_i & = & \overline{\mathbf{A}}_i^{-1}(\mathbf{y}_i - \mathbf{B}_i\mathbf{z}_{i-1}), & i = 2, \ldots, N \\ \text{and} & & & \\ \mathbf{x}_N & = & \mathbf{z}_N & \\ \mathbf{x}_i & = & \mathbf{z}_i - \overline{\mathbf{C}}_i\mathbf{x}_{i+1}, & i = N-1, \ldots, 1, \end{array} \tag{A.4}$$

where the $\mathbf{x}_i$, $\mathbf{y}_i$ and $\mathbf{z}_i$, with $i = 1, \ldots, N$, are M-vectors that correspond to each blocks equation of the full system.