6

Partial Differential Equations

Preview

In this chapter, we shall deal with method–of–lines solutions to models that are described by individual partial differential equations, by sets of coupled partial differential equations, or possibly by sets of mixed partial and ordinary differential equations.

Emphasis will be placed on the process of converting partial differential equations to equivalent sets of ordinary differential equations, and particular attention will be devoted to the problem of converting boundary conditions. To this end, we shall again consult our –meanwhile well–understood– Newton–Gregory polynomials.

We shall then spend some time analyzing the particular difficulties that await us when numerically solving the sets of resulting differential equations in the cases of parabolic, hyperbolic, and elliptic partial differential equations. It turns out that each class of partial differential equations exhibits its own particular and peculiar types of difficulties.

6.1 Introduction

Partial differential equation (PDE) modeling and simulation are certainly among the more difficult topics to deal with. PDE modeling is still in its infancy. You hardly ever encounter models of coupled PDEs that contain more than three or four PDEs at a time. This situation is comparable with ordinary differential equation (ODE) modeling some 30 years ago. At that time, researchers were content to analyze simple ODE models consisting of three or four coupled ODEs. No special software tools were needed to help the modeler organize his or her models. The modeling process was utterly trivial. What was difficult was the process of converting these ODEs to a form such that a numerical differential equation solver could tackle them, and then the process of simulation itself.

This way of looking at simulation still prevails in large portions of the simulation literature. However, reality of ODE modeling has changed drastically over the years. Today, continuous system modelers frequently deal with models containing hundreds or even thousands of coupled differential and algebraic equations, and the process of first deriving and then maintaining these ODE models has become the truly difficult part.

This was the focus point of the companion book to this text *Continuous*

System Modeling [6.5]. In that book, PDEs weren't mentioned with even one word. The reason for this is obvious. No special software tools or modeling methodologies are needed yet to derive or maintain PDE models, since PDE models are still very simple. You don't encounter models containing hundreds or even only tens of PDEs. It just isn't done. If you end up with three or four coupled PDEs, this is a lot. So, from a modeling perspective, PDE modeling is still a fairly trivial undertaking.

On the other hand, the numerical solution of PDE models is by no means trivial. Whereas we have learnt meanwhile pretty well how to numerically handle large classes of ODE models, the numerical solution of PDE models still presents a challenge.

Many different approaches to simulating PDE models have been described in the literature, partly purely numerical, such as the finite element methods used mostly to tackle elliptic PDE problems, and partly semi–analytical, such as the method–of–characteristics approach to solving hyperbolic systems of equations. It is not the aim of this chapter at all to duplicate or compete with that literature.

Among all the techniques that are known for tackling PDE models, only one specific technique shall be dealt with in this book, namely the method–of–lines (MOL) approach to numerically solving PDE models. The MOL methodology converts PDEs into (large) sets of (in some way equivalent) ODEs that are then solved by standard ODE solvers. Since this book deals explicitly and extensively with ODE solvers, the MOL approach to PDE solving fits well within the overall framework of this book methodologically. This is the only reason why this text focuses on MOL solutions. It is not our intention to convey the impression that MOL solutions are, in each and every case, the most suitable way of dealing with PDE problems. PDE problems are notoriously difficult to tackle, and the MOL approach is only one, among many, techniques that can provide a partial answer to these challenges.

6.2 The Method of Lines

The Method of Lines (MOL) is a technique that enables us to convert partial differential equations (PDEs) into sets of ordinary differential equations (ODEs) that, in some sense, are equivalent to the former PDEs.

The basic idea behind the MOL methodology is straightforward. Let us look at the simple *heat equation* or *diffusion equation* in a single space variable:

$$\frac{\partial u}{\partial t} = \sigma \cdot \frac{\partial^2 u}{\partial x^2} \tag{6.1}$$

Rather than looking at the solution $u(x, t)$ everywhere in the two–dimensional space spanned by the spatial variable x and the temporal variable t, we can discretize the spatial variable, and look at the solutions $u_i(t)$ where the index i denotes a particular point x_i in space. To this end, we replace the second–order partial derivative of u with respect to x by a finite difference, such as:

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{x=x_i} \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{\delta x^2} \tag{6.2}$$

where δx is the (here equidistantly chosen) distance between two neighboring discretization points in space, i.e., the so–called *grid width* of the discretization.

Plugging Eq.(6.2) into Eq.(6.1), we find:

$$\frac{du_i}{dt} \approx \sigma \cdot \frac{u_{i+1} - 2u_i + u_{i-1}}{\delta x^2} \tag{6.3}$$

and we have already converted the former PDE in u into a set of ODEs in u_i.

The principal idea behind the MOL methodology is thus utterly trivial. However, the devil is in the detail.

It is reasonable to use the same order of approximation accuracy for the discretization in space as for the discretization in time achieved by the numerical integration algorithm. Thus, if we plan to integrate the set of ODEs with a fourth–order method, we should better find a discretization formula for $\partial^2 u/\partial x^2$ that is also fourth–order accurate.

This can be accomplished by use of our old friends, the *Newton–Gregory polynomials*. A fourth–order polynomial needs to be fitted through five points. Since we prefer *central differences* over *biased differences*, we fit the polynomial through the five points x_{i-2}, x_{i-1}, x_i, x_{i+1}, and x_{i+2}. Using Newton–Gregory backward polynomials, we will have to write the polynomial around the point that is located most to the right, in our case, the point x_{i+2}. Thus, we write:

$$u(x) = u_{i+2} + s\nabla u_{i+2} + \left(\frac{s^2}{2} + \frac{s}{2}\right)\nabla^2 u_{i+2} + \left(\frac{s^3}{6} + \frac{s^2}{2} + \frac{s}{3}\right)\nabla^3 u_{i+2} + \dots \tag{6.4}$$

Notice that we write the approximation polynomial as $u(x)$ rather than as $u(t)$, since we want to discretize along the spatial axis.

Consequently, the second derivative can be written as:

$$\frac{\partial^2 u}{\partial x^2} = \frac{1}{\delta x^2}\left[\nabla^2 u_{i+2} + (s+1)\nabla^3 u_{i+2} + \left(\frac{s^2}{2} + \frac{3s}{2} + \frac{11}{12}\right)\nabla^4 u_{i+2} + \dots\right] \tag{6.5}$$

Eq.(6.5) needs to be evaluated at $x = x_i$, corresponding to $s = -2$. Truncating after the quartic term and expanding the ∇–operators, we find:

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{x=x_i} \approx \frac{1}{12\delta x^2}\left(-u_{i+2} + 16u_{i+1} - 30u_i + 16u_{i-1} - u_{i-2}\right) \qquad (6.6)$$

which is the fourth–order central difference approximation to the second partial derivative of $u(x,t)$ with respect to x evaluated at $x = x_i$.

We could have obtained the same result using the Newton–Gregory forward polynomial written around the point x_{i-2}, evaluating it for $s = +2$.

Had we decided that we wish to integrate with a second–order algorithm, we would have developed the Newton–Gregory backward polynomial around the point x_{i+1}, truncating Eq.(6.5) after the quadratic term, and evaluating for $s = -1$. This would have led to:

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{x=x_i} \approx \frac{1}{\delta x^2}\left(u_{i+1} - 2u_i + u_{i-1}\right) \qquad (6.7)$$

which is the second–order central difference formula for $\partial^2 u/\partial x^2$, the one that had been used in Eq.(6.2).

The third–order case is again a little different. For geometric reasons, it is obviously impossible to fit a central difference approximation of an odd order around x_i using only x_i and its nearest three neighbors. Thus, we can choose between a biased formula using the points x_{i-2} up to x_{i+1}, i.e., develop the Newton–Gregory backward polynomial around the point x_{i+1} and evaluate it for $s = -1$, and another biased formula using the points x_{i-1} up to x_{i+2}, i.e., develop the Newton–Gregory backward polynomial around the point x_{i+2} and evaluate it for $s = -2$.

It turns out that both cases lead to exactly the same formula, namely Eq.(6.7). Just by accident, a lot of terms drop out, and Eq.(6.7) turns out to be third–order accurate.

Looking more deeply into the matter, we find that the "lucky accident" is no accident at all, but has to do with the symmetry conditions. Every central difference approximation is one order more accurate than the number of points fitted by it would make us believe. Consequently, Eq.(6.6) is in fact fifth–order accurate.

The next difficulty arises as we approach the spatial domain boundary. Let us assume the heat equation applies to the temperature distribution along a rod of length $\ell = 1$ m. Let us assume we cut the rod into segments of a length of $\delta\ell = 10$ cm. Thus, we get 10 segments. If the left end of the rod corresponds to index $i = 1$, the right end corresponds to index $i = 11$. Let us further assume that we wish to integrate using a fourth–order algorithm. Thus, we shall apply Eq.(6.6) to the points x_3 up to x_9. However, for the remaining points, we need biased formulae, since we cannot use points outside the range where the solution $u(x,t)$ is defined.

In order to find a biased formula for x_2, we shall have to write the Newton–Gregory backward polynomial around the point u_5 and evaluate for $s = -3$, or alternatively, we can write a Newton–Gregory forward polynomial around the point u_1 and evaluate for $s = +1$. In order to find a biased formula for x_1, we shall have to write the Newton–Gregory backward polynomial around the point u_5 and evaluate for $s = -4$, or alternatively, we can write a Newton–Gregory forward polynomial around the point u_1 and evaluate for $s = 0$. Similarly for the points x_{10} and x_{11}.

Using the above example, we obtain the following biased approximation formulae:

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{x=x_1} = \frac{1}{12\delta x^2}\left(11u_5 - 56u_4 + 114u_3 - 104u_2 + 35u_1\right) \tag{6.8a}$$

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{x=x_2} = \frac{1}{12\delta x^2}\left(-u_5 + 4u_4 + 6u_3 - 20u_2 + 11u_1\right) \tag{6.8b}$$

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{x=x_{10}} = \frac{1}{12\delta x^2}\left(11u_{11} - 20u_{10} + 6u_9 + 4u_8 - u_7\right) \tag{6.8c}$$

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{x=x_{11}} = \frac{1}{12\delta x^2}\left(35u_{11} - 104u_{10} + 114u_9 - 56u_8 + 11u_7\right) \tag{6.8d}$$

In the MOL methodology, all derivatives w.r.t. spatial variables are discretized using either central or biased difference approximations, whereas derivatives w.r.t. the temporal variable are left unchanged. In this way, PDEs are converted into sets of ODEs that can, at least in theory, be solved just like any other ODE models by means of standard ODE solvers.

Next, we need to discuss what is to be done with the boundary conditions. Every PDE has beside from *initial conditions* in time *boundary conditions* in space. For example, the heat equation may have the two boundary conditions:

$$u(x = 0.0, t) = 100.0 \tag{6.9a}$$

$$\frac{\partial u}{\partial x}(x = 1.0, t) = 0.0 \tag{6.9b}$$

The boundary condition of Eq.(6.9a) is called *boundary value condition*. This is the simplest case. All we need to do is to eliminate the differential equation for $u_1(t)$, and replace it by an algebraic equation, in our case:

$$u_1 = 100.0 \tag{6.10}$$

The boundary condition of Eq.(6.9b) is also a special case. It is called a *boundary symmetry condition*. It is handled in the following way. Imagine that there is a mirror at $x = 1.0$. This mirror maps the solution $u(x, t)$ into the range $x \in [1.0, 2.0]$, such that $u(2.0 - x, t) = u(x, t)$. Obviously, the boundary condition at $x = 2.0$ is the same as that at $x = 0.0$. There is then no need at all to specify any boundary condition at $x = 1.0$, since, through symmetry, the desired boundary symmetry condition will be satisfied. Knowing this, we can replace Eqs.(6.8c–d) by:

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{x=x_{10}} = \frac{1}{12\delta x^2}\left(-u_{12} + 16u_{11} - 30u_{10} + 16u_9 - u_8\right) \tag{6.11a}$$

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{x=x_{11}} = \frac{1}{12\delta x^2}\left(-u_{13} + 16u_{12} - 30u_{11} + 16u_{10} - u_9\right) \tag{6.11b}$$

i.e., by central difference approximations. However, since (due to symmetry) $u_{12} = u_{10}$ and $u_{13} = u_9$, we can rewrite Eqs.(6.11a–b) as:

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{x=x_{10}} = \frac{1}{12\delta x^2}\left(16u_{11} - 31u_{10} + 16u_9 - u_8\right) \tag{6.12a}$$

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{x=x_{11}} = \frac{1}{12\delta x^2}\left(-30u_{11} + 32u_{10} - 2u_9\right) \tag{6.12b}$$

and having done this, we can happily forget our virtual mirror again. We don't need to bother to actually compute a solution for the range $x \in [1.0, 2.0]$, since we already know the solution ... it is the mirror image of the solution in the range $x \in [0.0, 1.0]$.

A third type of special boundary conditions is the so–called *temporal boundary condition* of the type:

$$\frac{\partial u}{\partial t}(x = 0.0, t) = f(t) \tag{6.13}$$

In this case, the boundary condition of the PDE is itself described through an ODE. This case is also easy. We simply replace the ODE for u_1 by the boundary ODE:

$$\dot{u}_1 = f(t) \tag{6.14}$$

The more general boundary condition of the type:

$$g\left(u(x = 1.0, t)\right) + h\left(\frac{\partial u}{\partial x}(x = 1.0, t)\right) = f(t) \tag{6.15}$$

where f, g, and h are arbitrary functions, is more tricky. For example, we may have to deal with a boundary condition of the type:

$$\frac{\partial u}{\partial x}(x = 1.0, t) = -k \cdot \left(u(x = 1.0, t) - u_{\text{amb}}(t)\right) \tag{6.16}$$

where $u_{\text{amb}}(t)$ is the ambient temperature. How would we handle such a general boundary condition? The answer is simple. We again replace all spatial derivatives by appropriate Newton–Gregory polynomials, e.g. in the above case:

$$\left.\frac{\partial u}{\partial x}\right|_{x=x_{11}} = \frac{1}{12\delta x}\left(25u_{11} - 48u_{10} + 36u_9 - 16u_8 + 3u_7\right) \tag{6.17}$$

is the fourth–order biased difference approximation polynomial. Plugging Eq.(6.17) into Eq.(6.16), and solving for u_{11}, we find:

$$u_{11} = \frac{12k \cdot \delta x \cdot u_{\text{amb}} + 48u_{10} - 36u_9 + 16u_8 - 3u_7}{12k \cdot \delta x + 25} \tag{6.18}$$

By this process, the *general boundary condition* has been transformed into a *boundary value condition*, and the ODE defining u_{11} can be dropped.

Often we are faced with *nonlinear boundary conditions*, such as the radiation condition:

$$\frac{\partial u}{\partial x}(x = 1.0, t) = -k \cdot \left(u(x = 1.0, t)^4 - u_{\text{amb}}(t)^4\right) \tag{6.19}$$

which leads to:

$$\begin{aligned}\mathcal{F}(u_{11}) =& 12k \cdot \delta x \cdot u_{11}^4 + 25u_{11} - 12k \cdot \delta x \cdot u_{\text{amb}}^4 - 48u_{10} + 36u_9 \\ &- 16u_8 + 3u_7 = 0.0\end{aligned} \tag{6.20}$$

i.e., an *implicit boundary value condition* that can be solved by Newton iteration. Convergence should be fast since we can always use the value of $u_{11}(t_k - h)$ as the starting value of the iteration.

Finally, let us consider diffusion of heat through a wall. Assume that the wall has two layers consisting of two different materials, one of 1 m thickness, the other of 10 cm thickness. In that case, the diffusion coefficient, σ, assumes a different value in the two materials. We can formulate this problem as follows:

$$\frac{\partial u}{\partial t} = \sigma_u \cdot \frac{\partial^2 u}{\partial x^2} \tag{6.21a}$$

$$\frac{\partial v}{\partial t} = \sigma_v \cdot \frac{\partial^2 v}{\partial x^2} \tag{6.21b}$$

where the PDE for $u(x, t)$ is valid in the region $x \in [0.0, 1.0]$, and the PDE for $v(x, t)$ is valid in the region $x \in [1.0, 1.1]$, with boundary conditions at the boundary between the two layers:

$$\frac{\partial u}{\partial x}(x = 1.0, t) = -k_u \cdot (u(x = 1.0, t) - v(x = 1.0, t)) \tag{6.22a}$$

$$\frac{\partial v}{\partial x}(x = 1.0, t) = -k_v \cdot (v(x = 1.0, t) - u(x = 1.0, t)) \tag{6.22b}$$

which leads to the following two equations:

$$(12k_u \cdot \delta x_u + 25)u_{11} - 12k_u \cdot \delta x_u \cdot v_1 = 48u_{10} - 36u_9 + 16u_8 - 3u_7 \tag{6.23a}$$

$$-12k_v \cdot \delta x_v \cdot u_{11} + (12k_v \cdot \delta x_v + 3)v_1 = 16v_2 - 36v_3 + 48u_4 - 25v_5 \tag{6.23b}$$

Eqs.(6.23a–b) constitute a *linear algebraic loop* in the unknown variables u_{11} and v_1 that can be solved either symbolically or numerically.

6.3 Parabolic PDEs

Some very simple types of PDEs are so common that they were given special names. Let us consider the following PDE in two variables x and y:

$$a\frac{\partial^2 u}{\partial x^2} + b\frac{\partial^2 u}{\partial x \partial y} + c\frac{\partial^2 u}{\partial y^2} = d \tag{6.24}$$

which is characteristic of many field problems in physics. x and y can be either spatial or temporal variables, and a, b, c, and d can be arbitrary functions of x, y, u, $\partial u/\partial x$, and $\partial u/\partial y$. Such a PDE is called *quasi–linear*, since it is linear in the highest derivatives.

Depending on the numerical relationship between a, b, and c, Eq.(6.24) is classified as either being *parabolic*, *hyperbolic*, or *elliptic*. The classification is as follows:

$$b^2 - 4ac > 0 \Longrightarrow \text{PDE is hyperbolic} \tag{6.25a}$$

$$b^2 - 4ac = 0 \Longrightarrow \text{PDE is parabolic} \tag{6.25b}$$

$$b^2 - 4ac < 0 \Longrightarrow \text{PDE is elliptic} \tag{6.25c}$$

This classification makes sense, since the numerical methods most suitable for these three types of PDEs are vastly different. In this section, we shall deal with PDEs of the parabolic type exclusively.

Parabolic PDEs are very common. For example, all thermal field problems are of that nature. The simplest example of a parabolic PDE is the one–dimensional heat diffusion problem of Eq.(6.1). A complete example of such a problem is specified once more below.

$$\frac{\partial u}{\partial t} = \frac{1}{10\pi^2} \cdot \frac{\partial^2 u}{\partial x^2} \quad ; \quad x \in [0,1] \quad ; \quad t \in [0,\infty) \tag{6.26a}$$

$$u(x, t=0) = \cos(\pi \cdot x) \tag{6.26b}$$

$$u(x=0, t) = \exp(-t/10) \tag{6.26c}$$

$$\frac{\partial u}{\partial x}(x=1, t) = 0 \tag{6.26d}$$

Equation (6.26a) is the one–dimensional heat equation, Eq.(6.26b) constitutes its single initial condition, and Eqs.(6.26c–d) describe its two boundary conditions.

Let us discretize this problem using the MOL approach. We split the spatial axis into n segments of length $\delta x = 1/n$. We shall apply the third–order accurate central difference formula of Eq.(6.7) for the approximation of the spatial derivatives. We furthermore use the symmetry boundary condition approach at the right end of the interval. This leads to the following set of ODEs:

$$u_1 = \exp(-t/10) \tag{6.27a}$$

$$\dot{u}_2 = \frac{n^2}{10\pi^2} \cdot (u_3 - 2u_2 + u_1) \tag{6.27b}$$

$$\dot{u}_3 = \frac{n^2}{10\pi^2} \cdot (u_4 - 2u_3 + u_2) \tag{6.27c}$$

etc.

$$\dot{u}_n = \frac{n^2}{10\pi^2} \cdot (u_{n+1} - 2u_n + u_{n-1}) \tag{6.27d}$$

$$\dot{u}_{n+1} = \frac{n^2}{5\pi^2} \cdot (-u_{n+1} + u_n) \tag{6.27e}$$

with initial conditions:

$$u_2(0) = \cos\left(\frac{\pi}{n}\right) \tag{6.28a}$$

$$u_3(0) = \cos\left(\frac{2\pi}{n}\right) \tag{6.28b}$$

$$u_4(0) = \cos\left(\frac{3\pi}{n}\right) \tag{6.28c}$$

etc.

$$u_n(0) = \cos\left(\frac{(n-1)\pi}{n}\right) \tag{6.28d}$$

$$u_{n+1}(0) = \cos(\pi) \tag{6.28e}$$

This is a linear, time–invariant, inhomogeneous, n^{th}–order, single–input system of the type:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} + \mathbf{b} \cdot u \tag{6.29}$$

where:

$$\mathbf{A} = \frac{n^2}{10\pi^2} \cdot \begin{pmatrix} -2 & 1 & 0 & 0 & \dots & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & \dots & 0 & 2 & -2 \end{pmatrix} \tag{6.30}$$

A is a band–structured matrix of dimensions $n \times n$. Let us calculate its eigenvalues. They are tabulated in Table 6.1.

$n = 3$	$n = 4$	$n = 5$	$n = 6$	$n = 7$
-0.0244	-0.0247	-0.0248	-0.0249	-0.0249
-0.1824	-0.2002	-0.2088	-0.2137	-0.2166
-0.3403	-0.4483	-0.5066	-0.5407	-0.5621
	-0.6238	-0.9884	-0.9183	-0.9929
		-0.8044	-1.2454	-1.4238
			-1.4342	-1.7693
				-1.9610

TABLE 6.1. Eigenvalue distribution for diffusion model.

All eigenvalues are strictly negative and real. This is characteristic of all thermal field problems and all parabolic PDEs converted to sets of ODEs by the MOL technique.

We notice at once that, whereas the damping properties of the system (determined by the location of the dominant pole) don't change significantly with the number of segments, the *stiffness ratio*, i.e., the ratio between the absolute largest real part and the absolute smallest real part of any eigenvalue depends heavily on the number of segments. Figure 6.1 shows the square root of the stiffness ratio plotted over the number of segments chosen.

It turns out that, for all practical purposes, the stiffness ratio grows quadratically with the number of segments chosen in the spatial discretization process. The more accurate we wish to solve the diffusion equation, the stiffer the corresponding ODE problem will become. Since diffusion problems are usually quite smooth, the BDF algorithms are optimally suited to simulate the resulting set of ODEs.

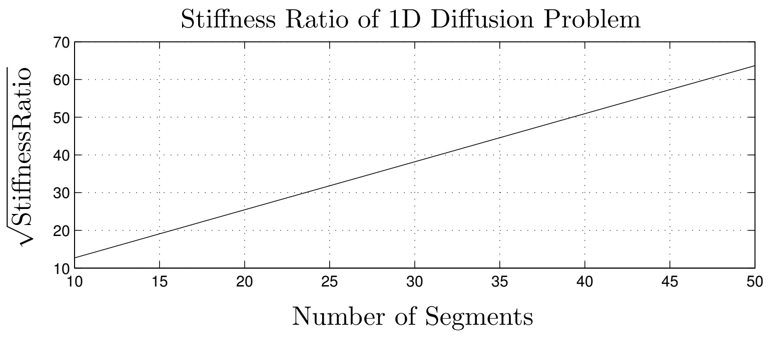


FIGURE 6.1. Dependence of stiffness ratio on discretization.

We chose a PDE problem, the analytical solution of which is known. It happens to be:

$$u_c(x,t) = \exp(-t/10) \cdot \cos(\pi \cdot x) \tag{6.31}$$

Hence we can compare the analytical solution of the original PDE problem with the equally analytical solution of the discretized ODE problem after applying the MOL discretization.

The analytical solution of the discretized ODE problem is a little harder to come by. We can create a system description of the continuous–time problem:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} + \mathbf{b} \cdot u \tag{6.32a}$$
$$\mathbf{y} = \mathbf{C} \cdot \mathbf{x} + \mathbf{d} \cdot u \tag{6.32b}$$

where **C** is an identity matrix of suitable dimensions, and **d** is a zero vector using MATLAB's *control system toolbox*:

$$\mathbf{Sc} = \text{ss}(\mathbf{A}, \mathbf{b}, \mathbf{C}, \mathbf{d}) \tag{6.33}$$

This continuous–time system can then be converted to an equivalent discrete–time system:

$$\mathbf{x_{k+1}} = \mathbf{F} \cdot \mathbf{x_k} + \mathbf{g} \cdot u_k \tag{6.34a}$$
$$\mathbf{y_k} = \mathbf{H} \cdot \mathbf{x_k} + \mathbf{i} \cdot u_k \tag{6.34b}$$

using the statement:

$$\mathbf{Sd} = \text{c2d}(\mathbf{Sc}, h) \tag{6.35}$$

from which the **F**–matrix and **g**–vector of the discrete state equations can be extracted using the statement:

$$[\mathbf{F}, \mathbf{g}] = \text{ssdata}(\mathbf{Sd}) \tag{6.36}$$

The discrete–time system can now be "simulated" by means of iteration of the discrete state equations. The solution of the discrete difference equation (ΔE) system is identical with that of the continuous ODE problem at the sampling points $k \cdot h$, where h is the step size (sampling rate) of the discrete problem, except for the discretization of the input function. The discrete system assumes that the input function $u(t)$ is kept constant in between sampling points.

Consequently, the step size, h, must be chosen small enough for the effect of the discretization of the input function to be negligible.

Let us look at the results of the experiment. The top left graph of Fig.6.2 shows the solution of the PDE problem, u_c, as a function of space and time, whereas the top right graph shows the solution of the discretized ODE problem, u_d, simulated using the approach discussed above. The two graphs look identical by visual inspection. The bottom left graph of Fig.6.2 displays the difference between the two functions, i.e.:

$$err = u_c - u_d \tag{6.37}$$

and the bottom right graph of Fig.6.2 presents the maximum error, er_{max}, as a function of the number of segments used in the discretization. The maximum error was computed using the MATLAB statement:

$$er_{max} = \max(\max(\text{abs}(err))); \tag{6.38}$$

The step size, h, was chosen small enough so that a further reduction of h would not visibly change the bottom right graph of Fig.6.2 any longer. In the given example, a step size of $h = 0.001$ had to be chosen to accomplish this goal.

We have just come across a new type of error. The *consistency error* describes the difference between the original PDE problem that we wish to solve, and the discretized ODE problem that we are actually solving.

Evidently, the consistency error cannot be overcome by either step–size or order control of the underlying ODE solver. Even the best ODE solver can only approximate the analytical solution, u_d, of the discretized ODE problem, but never the true analytical solution, u_c, of the original PDE problem.

Is the consistency error a *modeling error* or a *simulation error*? The answer to this question depends on the point of view. If we use a modeling environment that allows us to describe the PDE problem directly, we are inclined to call this a simulation error. However, it is an error that is incurred during the symbolic formulae manipulations that accompany the compilation of the model, rather than at run time. On the other hand, if we use a lower–level modeling environment that forces us to convert the

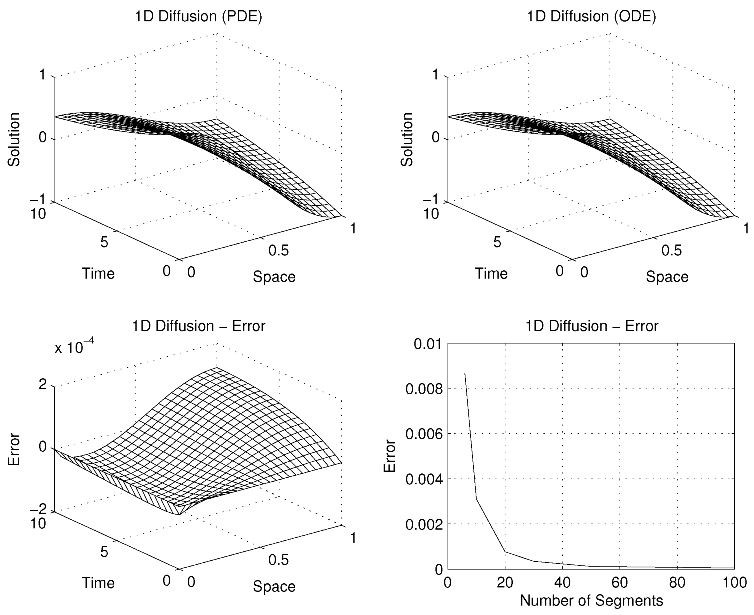


FIGURE 6.2. Solution of the 1D heat diffusion problem.

PDE manually into a set of ODEs, we would be more inclined to call this a modeling error.

Can the consistency error be overcome by choosing a more accurate scheme for the computation of the spacial derivatives? Let us use a 5^{th}–order accurate central difference scheme together with an equally 5^{th}–order accurate biased difference scheme for the discretization points near the two boundaries, hence:

$$u_1 = \exp(-t/10) \tag{6.39a}$$

$$\dot{u}_2 = \frac{n^2}{120\pi^2} \cdot (u_6 - 6u_5 + 14u_4 - 4u_3 - 15u_2 + 10u_1) \tag{6.39b}$$

$$\dot{u}_3 = \frac{n^2}{120\pi^2} \cdot (-u_5 + 16u_4 - 30u_3 + 16u_2 - u_1) \tag{6.39c}$$

$$\dot{u}_4 = \frac{n^2}{120\pi^2} \cdot (-u_6 + 16u_5 - 30u_4 + 16u_3 - u_2) \tag{6.39d}$$

$$\text{etc.}\textit{nonumber} \tag{6.39e}$$

$$\dot{u}_{n-1} = \frac{n^2}{120\pi^2} \cdot (-u_{n+1} + 16u_n - 30u_{n-1} + 16u_{n-2} - u_{n-3}) \tag{6.39f}$$

$$\dot{u}_n = \frac{n^2}{120\pi^2} \cdot (16u_{n+1} - 31u_n + 16u_{n-1} - u_{n-2}) \tag{6.39g}$$

$$\dot{u}_{n+1} = \frac{n^2}{60\pi^2} \cdot (-15u_{n+1} + 16u_n - u_{n-1}) \tag{6.39h}$$

The bulk of the equations are formulated using 5th–order accurate central differences. Equation (6.39b) is specified using the 5th–order accurate biased difference formula, whereas Eqs.(6.39g) and (6.39h) are derived by making use of the symmetry boundary condition.

Hence the resulting **A**–matrix takes the form:

$$\mathbf{A} = \frac{n^2}{120\pi^2} \cdot \begin{pmatrix} -15 & -4 & 14 & -6 & 1 & \dots & 0 & 0 & 0 & 0 \\ 16 & -30 & 16 & -1 & 0 & \dots & 0 & 0 & 0 & 0 \\ -1 & 16 & -30 & 16 & -1 & \dots & 0 & 0 & 0 & 0 \\ 0 & -1 & 16 & -30 & 16 & \dots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \dots & 16 & -30 & 16 & -1 \\ 0 & 0 & 0 & 0 & 0 & \dots & -1 & 16 & -31 & 16 \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & -2 & 32 & -30 \end{pmatrix} \tag{6.40}$$

The **A**–matrix is again band–structured. However, the bandwidth is now wider. Its eigenvalues are tabulated in Table 6.2.

$n = 5$	$n = 6$	$n = 7$	$n = 8$	$n = 9$
-0.0250	-0.0250	-0.0250	-0.0250	-0.0250
-0.2288	-0.2262	-0.2253	-0.2251	-0.2250
-0.5910	-0.6414	-0.6355	-0.6302	-0.6273
-0.7654	-0.9332	-1.1584	-1.2335	-1.2368
-1.2606	-1.3529	-1.4116	-1.6471	-1.9150
	-1.8671	-2.0761	-2.1507	-2.2614
		-2.5770	-2.9084	-3.0571
			-3.3925	-3.8460
				-4.3147

TABLE 6.2. Eigenvalue distribution for diffusion model.

The eigenvalue distribution has changed very little. In particular, all of them are still negative and real. Using this discretization scheme, the smallest number of segments is now five.

Figure 6.3 shows the square root of the stiffness ratio plotted as a function of the number of segments chosen. The corresponding stiffness ratio plot for the previously used **A**–matrix is presented also for comparison.

For the same number of segments, the stiffness ratio of the 5th–order scheme is slightly higher than that of the 3rd–order scheme. As the correct solution of the PDE problem corresponds to a discretization with infinitely many segments, i.e., an ODE problem with infinite stiffness, we may expect that the solution produced by the 5th–order scheme is indeed more accurate than that of the 3rd–order scheme.

Let us now perform the same experiment as before, this time using the 5th–order scheme. Figure 6.4 shows the consistency error as a function of

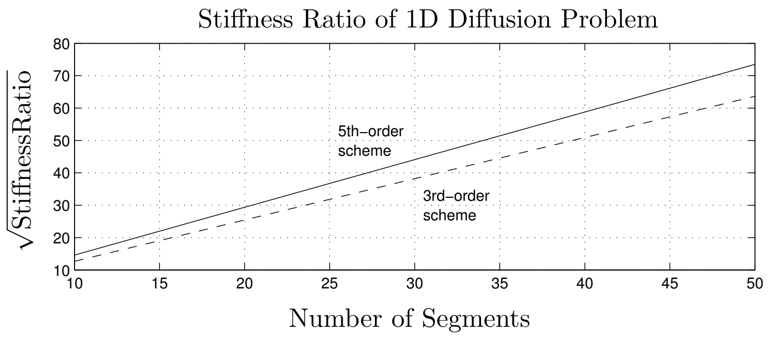


FIGURE 6.3. Dependence of stiffness ratio on discretization.

the number of segments used in the discretization scheme. The results of using the 3^{rd}–order accurate discretization scheme and those using the 5^{th}–order accurate discretization scheme are superposed on the same graph.

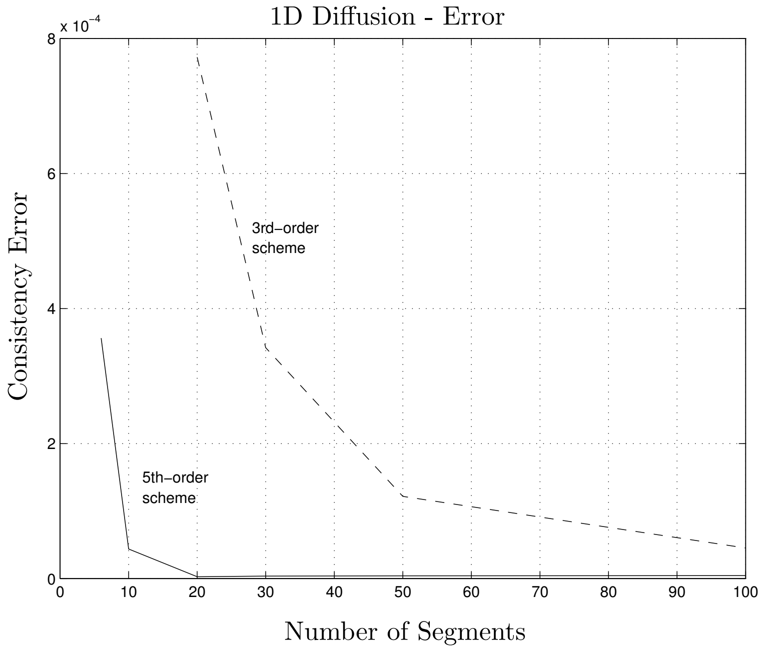


FIGURE 6.4. Consistency error of the 1D heat diffusion problem.

The improvement achieved by the more accurate discretization scheme is quite dramatic. Yet, the "simulation" of the discretized problem is much more expensive in this case. We had to choose a smaller step size of $h = 0.0001$ before the consistency error would no longer decrease by further reducing the step size.

This observation is not overly surprising. Since the stiffness ratio for the same number of segments has grown, yet the slowest eigenvalues have not moved, the fastest eigenvalues are now much further to the left in the complex λ–plane. Hence we need to choose a smaller step size, h, in order to operate within the accuracy region of the complex $\lambda \cdot h$–plane of the numerical simulation scheme.

This, unfortunately, is the biggest crux in the numerical solution of parabolic PDE problems. If we double the number of segments, the number of ODEs to be simulated doubles as well. However, since the stiffness ratio grows quadratically in the number of segments, the step size needs to decrease inverse quadratically in order to keep the accuracy the same in the complex $\lambda \cdot h$–plane. Hence doubling the number of segments forces us to quadruple the number of time steps. Hence the *simulation effort* grows cubically in the number of segments.

Let us try another approach. You certainly remember the *Richardson extrapolation* technique that we talked about in Chapter 3 of this text. Let us ascertain whether Richardson extrapolation may provide us with better answers to our approximation problem.

We can find four different third–order accurate approximations of $\partial^2 u/\partial x^2$:

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{x=x_i}^{P_1} (\delta x^2) = \frac{u_{i+1} - u_i + u_{i-1}}{\delta x^2} \tag{6.41a}$$

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{x=x_i}^{P_2} (4\delta x^2) = \frac{u_{i+2} - u_i + u_{i-2}}{4\delta x^2} \tag{6.41b}$$

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{x=x_i}^{P_3} (9\delta x^2) = \frac{u_{i+3} - u_i + u_{i-3}}{9\delta x^2} \tag{6.41c}$$

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{x=x_i}^{P_4} (16\delta x^2) = \frac{u_{i+4} - u_i + u_{i-4}}{16\delta x^2} \tag{6.41d}$$

These approximations differ only in the grid width δx used to obtain them. We can write:

$$\frac{\partial^2 u}{\partial x^2}(\eta) = \frac{\partial^2 u}{\partial x^2} + e_1 \cdot \eta + e_2 \cdot \frac{\eta^2}{2!} + e_3 \cdot \frac{\eta^3}{3!} + \dots \tag{6.42}$$

where $\partial^2 u/\partial x^2$ is the true (yet unknown) value of the second spatial derivative of u, whereas $\partial^2 u(\eta)/\partial x^2$ is the numerical value that we find when we approximate the second spatial derivative using a grid width of η. Obviously, this value contains an error. Equation (6.42) is a Taylor–Series in η around the (unknown) correct value. The e_i variables are errors of the approximation.

We truncate the Taylor Series after the cubic term, and write Eq.(6.42) down for the same values of the grid width that had been used in Eqs.(6.41a–d). We find:

$$\begin{aligned}
\frac{\partial^2 u}{\partial x^2}^{P_1}(\delta x^2) &\approx \frac{\partial^2 u}{\partial x^2} + e_1 \cdot \delta x^2 + \frac{e_2}{2!} \cdot \delta x^4 + \frac{e_3}{3!} \cdot \delta x^6 \\
\frac{\partial^2 u}{\partial x^2}^{P_2}(4\delta x^2) &\approx \frac{\partial^2 u}{\partial x^2} + e_1 \cdot (4\delta x^2) + \frac{e_2}{2!} \cdot (4\delta x^2)^2 + \frac{e_3}{3!} \cdot (4\delta x^2)^3 \\
\frac{\partial^2 u}{\partial x^2}^{P_3}(9\delta x^2) &\approx \frac{\partial^2 u}{\partial x^2} + e_1 \cdot (9\delta x^2) + \frac{e_2}{2!} \cdot (9\delta x^2)^2 + \frac{e_3}{3!} \cdot (9\delta x^2)^3 \\
\frac{\partial^2 u}{\partial x^2}^{P_4}(16\delta x^2) &\approx \frac{\partial^2 u}{\partial x^2} + e_1 \cdot (16\delta x^2) + \frac{e_2}{2!} \cdot (16\delta x^2)^2 + \frac{e_3}{3!} \cdot (16\delta x^2)^3
\end{aligned} \tag{6.43}$$

or in a matrix notation:

$$\begin{pmatrix} \frac{\partial^2 u}{\partial x^2}^{P_1} \\ \frac{\partial^2 u}{\partial x^2}^{P_2} \\ \frac{\partial^2 u}{\partial x^2}^{P_3} \\ \frac{\partial^2 u}{\partial x^2}^{P_4} \end{pmatrix} \approx \begin{pmatrix} (\delta x^2)^0 & (\delta x^2)^1 & (\delta x^2)^2 & (\delta x^2)^3 \\ (4\delta x^2)^0 & (4\delta x^2)^1 & (4\delta x^2)^2 & (4\delta x^2)^3 \\ (9\delta x^2)^0 & (9\delta x^2)^1 & (9\delta x^2)^2 & (9\delta x^2)^3 \\ (16\delta x^2)^0 & (16\delta x^2)^1 & (16\delta x^2)^2 & (16\delta x^2)^3 \end{pmatrix} \cdot \begin{pmatrix} \frac{\partial^2 u}{\partial x^2} \\ e_1 \\ e_2/2 \\ e_3/6 \end{pmatrix} \tag{6.44}$$

By inverting the Van–der–Monde matrix, we can solve for the unknown $\partial^2 u/\partial x^2$ and the three error variables. Since we aren't interested in the errors, we only look at the first row of the inverted Van–der–Monde matrix. It turns out that the values in this row don't depend at all on the grid width δx. We find:

$$\frac{\partial^2 u}{\partial x^2} \approx \begin{pmatrix} \frac{56}{35} & -\frac{28}{35} & \frac{8}{35} & -\frac{1}{35} \end{pmatrix} \cdot \begin{pmatrix} \frac{\partial^2 u}{\partial x^2}^{P_1} \\ \frac{\partial^2 u}{\partial x^2}^{P_2} \\ \frac{\partial^2 u}{\partial x^2}^{P_3} \\ \frac{\partial^2 u}{\partial x^2}^{P_4} \end{pmatrix} \tag{6.45}$$

We can plug Eqs.(6.41) into Eq.(6.45), and find:

$$\begin{aligned}
\left.\frac{\partial^2 u}{\partial x^2}\right|_{x=x_i} \approx &\frac{1}{5040\delta x^2}(-9u_{i+4} + 128u_{i+3} - 1008u_{i+2} + 8064u_{i+1} \\
&- 14350u_i + 8064u_{i-1} - 1008u_{i-2} + 128u_{i-3} - 9u_{i-4})
\end{aligned} \tag{6.46}$$

which is exactly the central difference formula of order 9. Once again, the *Richardson extrapolation* has raised the approximation accuracy to the highest possible order.

Let us now look at a slightly different problem:

$$\frac{\partial u}{\partial t} = 4\frac{\partial^2 u}{\partial x^2} \quad ; \quad x \in [0,1] \quad ; \quad t \in [0,\infty) \tag{6.47a}$$

$$u(x, t=0) = 20\sin\left(\frac{\pi}{2}x\right) + 300 \tag{6.47b}$$

$$u(x=0, t) = 20\sin\left(\frac{\pi}{12}t\right) + 300 \tag{6.47c}$$

$$\frac{\partial u}{\partial x}(x=1, t) = 0 \tag{6.47d}$$

We again solve a one–dimensional heat equation, but with a different time constant, and different initial and boundary conditions.

This time around, we don't know the analytical solution, hence we cannot compute the consistency error explicitly. What do we do? Similarly to the step–size control algorithms discussed in the previous chapters, we need an estimator of the spatial discretization error.

All numerical algorithms should have a second algorithm built in to them that reasons about the sanity of the first algorithm and starts screaming if it thinks that something is going awry. Without such a sanity check, numerical algorithms are never safe. It is precisely the availability of such alarm systems that constitutes one of the major distinctions between *production codes* and *experimental codes*.

We propose to compute all spatial derivatives twice, once with the grid size δx, and once with the grid size $2\delta x$ using central differences.

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{x=x_i}^{P_1} (\delta x^2) = \frac{u_{i+1} - u_i + u_{i-1}}{\delta x^2} \tag{6.48a}$$

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{x=x_i}^{P_2} (4\delta x^2) = \frac{u_{i+2} - u_i + u_{i-2}}{4\delta x^2} \tag{6.48b}$$

(6.48c)

The two approximations form two separate partial derivative vectors, $\mathbf{u_{xx}^{P_1}}$ and $\mathbf{u_{xx}^{P_2}}$. Using these approximations, we can formulate a spatial error estimate:

$$\varepsilon_{\text{rel}} = \frac{|\mathbf{u_{xx}^{P_1}} - \mathbf{u_{xx}^{P_2}}|}{\max(|\mathbf{u_{xx}^{P_1}}|, |\mathbf{u_{xx}^{P_2}}|, \delta)} \tag{6.49}$$

where δ is a fudge factor, e.g., $\delta = 10^{-10}$.

If the estimated spatial discretization error is too big, we must either choose a more narrow grid, or alternatively, we must increase the approximation order of the spatial derivatives.

Is it wasteful to compute the entire vector of spatial derivatives twice? This question must clearly be answered in the negative. The two predictors can be used in a *Richardson corrector* step:

$$\mathbf{u_{xx}^C} = \frac{4}{3} \cdot \mathbf{u_{xx}^{P_1}} - \frac{1}{3} \cdot \mathbf{u_{xx}^{P_2}} \tag{6.50}$$

This is equivalent to having raised the approximation order of the spatial derivatives from three to five. However, by writing the 5^{th}–order accurate spatial derivative formula in this way, we get an error estimator essentially for free.

Since the problem is stiff, a BDF formula may be appropriate for its integration. As we wish to obtain a global accuracy of 1%, we decided to simulate the system using BDF3. We chose $n_{seg} = 50$ in order to receive sufficiently many output points in space, and simulated across 10 seconds in time. The simulation results are shown in Fig.6.5.

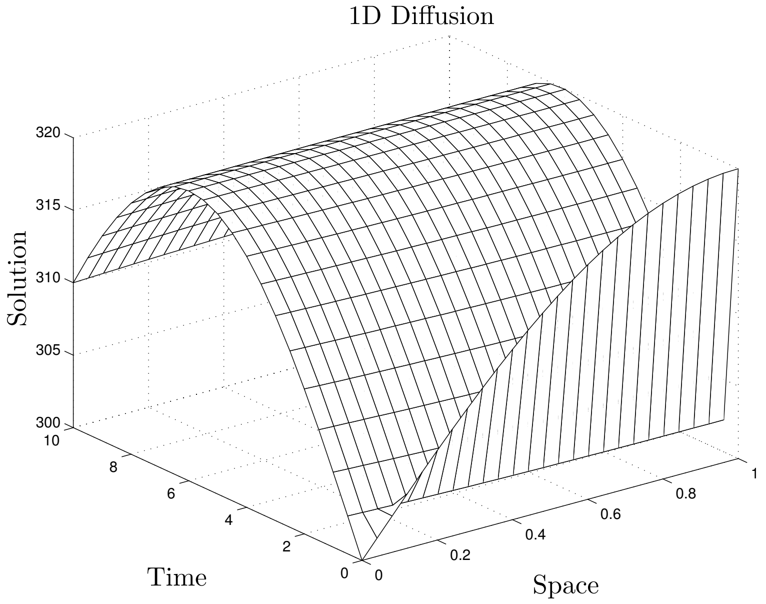


FIGURE 6.5. Solution of heat diffusion problem.

Figure 6.6 shows a slice through the solution at $x = 1.0$.

Unfortunately, the solution exhibits a fast transient precisely during the start–up period. The problem isn't truly stiff until the fast transients have died out. Initially, the solution is heavily controlled by accuracy requirements beside from the numerical stability constraints.

Assuming a fixed step size to be used throughout the solution, we repeated the simulation thrice, once using order buildup, i.e., a BDF starter, once using an RK3 starter, and once using an IEX3 starter. Figure 6.7 shows the step size required to achieve a desired level of accuracy using these three start–up algorithms.

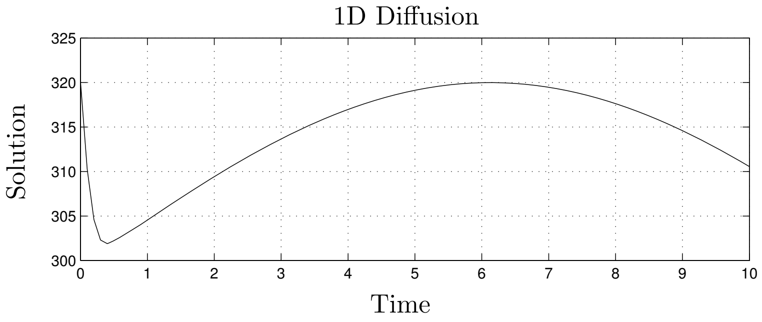


FIGURE 6.6. Solution of heat diffusion problem.

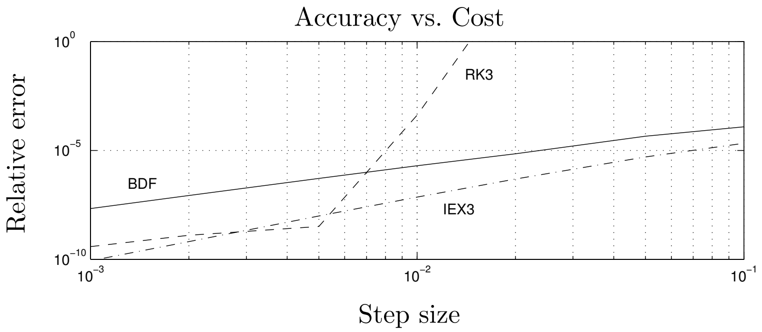


FIGURE 6.7. Accuracy *vs.* cost for different start–up algorithms.

Overall, the accuracy of the simulation seems to be quite a bit better than the 3^{rd}–order algorithm would have made us believe. In addition, the effect of the start–up algorithm on the simulation accuracy is quite dramatic. For small step sizes, the RK3 starter seems to work much better than the BDF starter. However, at $h = 0.005$, the numerical stability is lost, and the overall accuracy of the simulation degrades rapidly, in spite of the fact that the RK3 algorithm is only being used during the first two steps of the simulation. Of course, an RK starter implemented in a production code would be expected to proceed with a smaller step size than during the remainder of the simulation, but we did not want to make use of any type of step–size control in this experiment, as this would make an interpretation of the obtained results much more difficult.

The IEX3 starter, implemented using BDF1 steps internally, performs similarly to the RK3 starter for small step sizes, but without being plagued by the numerical stability problems of the RK3 starter for larger step sizes.

We also tried a $\text{BI4/5}_{0.45}$ starter. It didn't work well at all in this application. The reason is the following. The backward RK semi–step is numerically highly unstable. It is only stabilized by the Newton iteration. In the given application, we ran into roundoff error problems. The unstable semi–step produced numbers so big that the Newton iteration could not stabilize them any longer due to roundoff.

Parabolic PDE problems discretized using the MOL approach always turn into very stiff ODE systems. The more accurate we wish to simulate, the stiffer the problem becomes. Yet, decent stiff system solvers, such as DASSL [6.1], are usually quite capable of dealing with such problems effectively and efficiently.

6.4 Hyperbolic PDEs

Let us now analyze the second class of PDE problems, the hyperbolic PDEs. The simplest specimen of this class of problems is the *wave equation* or *linear conservation law*:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \cdot \frac{\partial^2 u}{\partial x^2} \tag{6.51}$$

We can easily transform this second–order PDE in time into two first order PDEs in time:

$$\frac{\partial u}{\partial t} = v \tag{6.52a}$$

$$\frac{\partial v}{\partial t} = c^2 \cdot \frac{\partial^2 u}{\partial x^2} \tag{6.52b}$$

At this point, we can replace the spatial derivatives again by finite difference approximations, and we seem to be in business.

Equations (6.53a–e) constitute a complete specification of such a model.

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} \quad ; \quad x \in [0,1] \quad ; \quad t \in [0,\infty) \tag{6.53a}$$

$$u(x, t=0) = \sin\left(\frac{\pi}{2}x\right) \tag{6.53b}$$

$$\frac{\partial u}{\partial t}(x, t=0) = 0.0 \tag{6.53c}$$

$$u(x=0, t) = 0.0 \tag{6.53d}$$

$$\frac{\partial u}{\partial x}(x=1, t) = 0.0 \tag{6.53e}$$

Equation (6.53a) is the one–dimensional wave equation, Eqs.(6.53b–c) constitute its two initial conditions, and Eqs.(6.53d–e) describe its two boundary conditions.

Let us simulate this problem using the MOL approach. We decide to split the spatial axis into n segments of width $\delta x = 1/n$. If we work with the central difference formula of Eq.(6.7), and using the symmetry boundary condition approach at the right end of the interval, we obtain the following set of ODEs:

$$u_1 = 0.0 \tag{6.54a}$$
$$\dot{u}_2 = v_2 \tag{6.54b}$$
etc.
$$\dot{u}_{n+1} = v_{n+1} \tag{6.54c}$$
$$v_1 = 0.0 \tag{6.54d}$$
$$\dot{v}_2 = n^2\left(u_3 - 2u_2 + u_1\right) \tag{6.54e}$$
$$\dot{v}_3 = n^2\left(u_4 - 2u_3 + u_2\right) \tag{6.54f}$$
etc.
$$\dot{v}_n = n^2\left(u_{n+1} - 2u_n + u_{n-1}\right) \tag{6.54g}$$
$$\dot{v}_{n+1} = 2n^2\left(u_n - u_{n+1}\right) \tag{6.54h}$$

with the initial conditions:

$$u_2(0) = \sin\left(\frac{\pi}{2n}\right) \tag{6.55a}$$
$$u_3(0) = \sin\left(\frac{\pi}{n}\right) \tag{6.55b}$$
$$u_4(0) = \sin\left(\frac{3\pi}{2n}\right) \tag{6.55c}$$
etc.
$$u_n(0) = \sin\left(\frac{(n-1)\pi}{2n}\right) \tag{6.55d}$$
$$u_{n+1}(0) = \sin\left(\frac{\pi}{2}\right) \tag{6.55e}$$
$$v_2(0) = 0.0 \tag{6.55f}$$
etc.
$$v_{n+1}(0) = 0.0 \tag{6.55g}$$

This is a linear, time–invariant, inhomogeneous, $(2n)^{\text{th}}$–order, single–input system of the type specified in Eq.(6.29), where:

$$\mathbf{A} = \begin{pmatrix} \mathbf{0^{(n)}} & \mathbf{I^{(n)}} \\ \mathbf{A_{21}} & \mathbf{0^{(n)}} \end{pmatrix} \tag{6.56}$$

with:

$$\mathbf{A_{21}} = n^2 \begin{pmatrix} -2 & 1 & 0 & 0 & \ldots & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & \ldots & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & \ldots & 0 & 2 & -2 \end{pmatrix} \tag{6.57}$$

A is a band–structured matrix of dimensions $2n \times 2n$ with two separate non–zero bands. Let us calculate its eigenvalues. They are tabulated in Table 6.3.

$n = 3$	$n = 4$	$n = 5$	$n = 6$
$\pm 1.5529j$	$\pm 1.5607j$	$\pm 1.5643j$	$\pm 1.5663j$
$\pm 4.2426j$	$\pm 4.4446j$	$\pm 4.5399j$	$\pm 4.5922j$
$\pm 5.7956j$	$\pm 6.6518j$	$\pm 7.0711j$	$\pm 7.3051j$
	$\pm 7.8463j$	$\pm 8.9101j$	$\pm 9.5202j$
		$\pm 9.8769j$	$\pm 11.0866j$
			$\pm 11.8973j$

TABLE 6.3. Eigenvalue distribution of linear conservation law.

All eigenvalues are strictly imaginary. All hyperbolic PDEs converted to sets of ODEs using the MOL technique show complex eigenvalues. Many of them have their eigenvalues spread up and down fairly close to the imaginary axis. The linear conservation law has all its eigenvalues exactly on the imaginary axis.

Figure 6.8 shows the *frequency ratio*, i.e., the ratio between the absolute largest and the absolute smallest imaginary parts of any eigenvalues plotted over the number of segments used in the discretization.

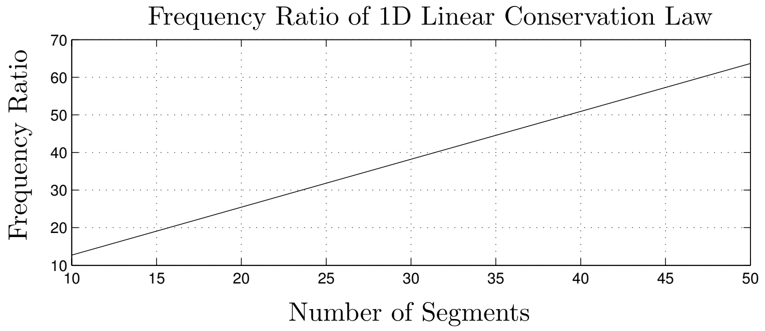


FIGURE 6.8. Frequency ratio of the 1D linear conservation law.

Evidently, the frequency ratio of the 1D linear conservation law grows linearly with the number of segments used in the discretization.

The numerical challenges are quite different from those in the parabolic case. The conservation law does not lead to a stiff set of ODEs. No "fast transients" appear that die out after some time, and consequently, the step size in the numerical integration must be kept small to account for all the eigenvalues of the discretized problem. The more narrow the grid width is chosen, the smaller the time steps will have to be in order to keep all eigenvalues within the asymptotic region of the numerical integration algorithm. Luckily, the spreading of the eigenvalues grows only linearly with the number of segments chosen.

We have seen that PDEs pose a new kind of challenge. In the case of ODE solutions, we only worried about stability and accuracy. In the case of PDE solution, we must concern ourselves with *stability*, *accuracy*, and *consistency*.

> *Definition:* "A discretization scheme is called *consistent* if the analytical solution of the discretized problem smoothly approaches the analytical solution of the original continuous problem as the grid width is being reduced to smaller and smaller values."

The *consistency error* is thus the deviation of the analytical solution of the discretized problem from the analytical solution of the continuous problem, whereas the *accuracy error* is the deviation of the numerical solution of the discretized problem from the analytical solution of the discretized problem.[1]

The example of Eqs.(6.53a–e) is so simple that an analytical solution of the continuous (field) problem can be given. It is:

$$u(x,t) = \frac{1}{2}\sin\left(\frac{\pi}{2}(x-t)\right) + \frac{1}{2}\sin\left(\frac{\pi}{2}(x+t)\right) \tag{6.58}$$

Since the discretized problem is linear with constant input, we can use the method described in Hw.[H4.8] to derive its analytical solution. Thus, we can go after the consistency error directly.

Figure 6.9 shows in its top left graph the analytical solution of the original PDE problem, in its top right graph the analytical solution of the discretized ODE problem. The two solutions look identical when compared by the naked eye. The bottom left curve shows the difference between the top two curves.

Since the input function is zero, the solution of the discretized ODE problem is independent of the chosen step size, h, in time. The discretization in

[1]Traditionally, the numerical PDE literature talks about the three facets: *stability*, *consistency*, and *convergence*. It is then customary to prove that any two of the three imply the third one, i.e., it is sufficient to look at any selection of two of the three [6.13]. However, that way of reasoning is more conducive to fully discretized (finite difference or finite element) schemes, where the step size in time, h, is locked in a fixed relationship with the grid width in space, δx. Consequently, h and δx approach zero simultaneously. In the context of the MOL methodology, our approach may be more appealing.

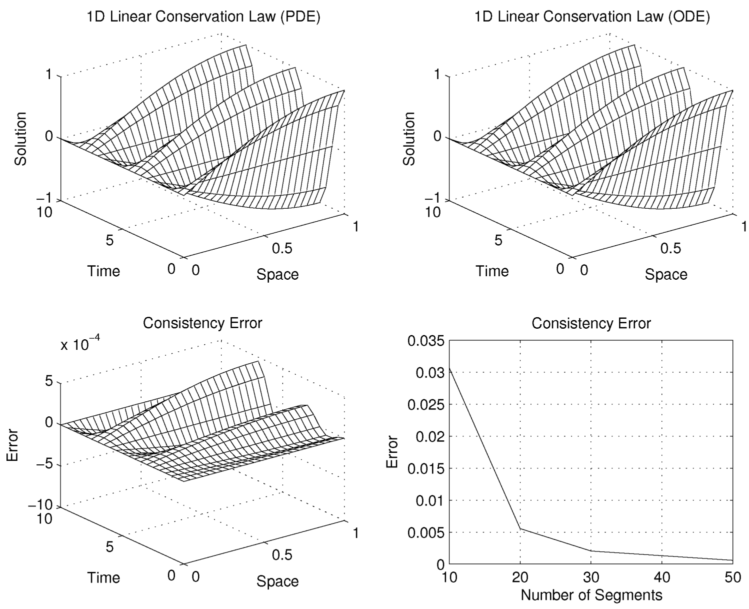


FIGURE 6.9. Analytical solutions of the 1D wave equation.

time serves here only for the purpose of generating sufficiently many output points. Hence the curve shown in the bottom left graph is the true *consistency error*. The only potential sources of numerical pollution could be due to roundoff and accumulation, but these are insignificant in magnitude in comparison with the analytical consistency error.

The bottom right graph shows the consistency error plotted against the number of segments chosen for the spatial discretization. The consistency error is here much larger than in the previous parabolic PDE examples. If we wish to obtain simulation results with a numerical accuracy of 1%, the consistency error itself ought to be at least one order of magnitude smaller. This means we should choose at least 40 segments for this simulation.

Just like in the case of the parabolic PDE problems, let us discuss what happens when we choose a higher–order discretization in space. Let us try first with 5^{th}–order central differences.

Figure 6.10 shows the frequency ratio plotted against the number of segments chosen in the spatial discretization scheme. The frequency ratio of the 3^{rd}–order scheme is plotted on the same graph for comparison.

The frequency ratio of the more accurate 5^{th}–order scheme is consistently higher than that of the less accurate 3^{rd}–order scheme for the same number of segments. Since the true PDE solution, corresponding to the solution with infinitely many infinitely dense discretization lines, has a frequency ratio that is infinitely large, we suspect that choosing a higher–order discretization scheme may indeed help with the reduction of the consistency

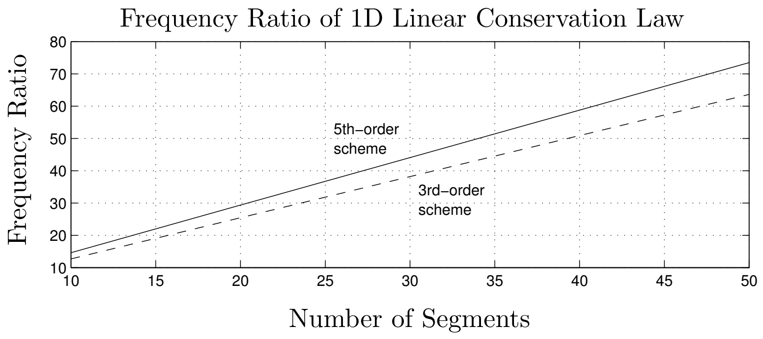


FIGURE 6.10. Frequency ratio of the 1D wave equation.

error.

Figure 6.11 shows the consistency error plotted over the number of segments used in the discretization. The improvement is quite dramatic. The consistency error has been reduced by at least two orders of magnitude.

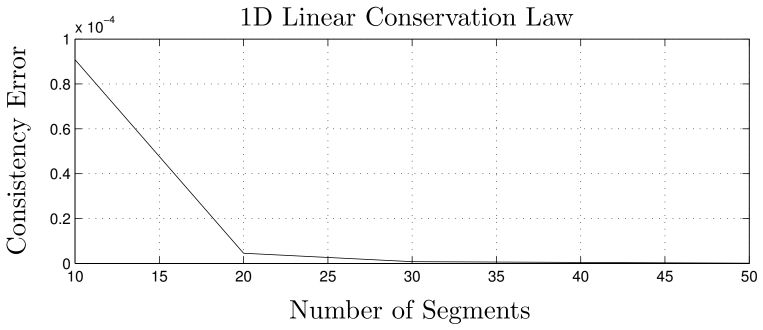


FIGURE 6.11. Consistency error of the 1D wave equation.

In a true simulation experiment, the 5^{th}–order spatial discretization scheme should be implemented using the *Richardson predictor–corrector technique* presented earlier in this chapter.

Let us compute the cost–versus–accuracy plot for the above problem, comparing the various third–order algorithms to each other that we meanwhile know. We shall use 50 segments for the spatial discretization together with 5^{th}–order central differences, in order to keep the consistency error sufficiently small, so that it won't affect the simulation results.

We computed the global accuracy of seven algorithms for simulating the discretized wave equation across 10 seconds of simulated time using a fixed step size of h, namely: RK3, IEX3, BI3, AB3, ABM3, AM3, and BDF3. We chose the step sizes: $h = 0.1,\ h = 0.05,\ h = 0.02,\ h = 0.01,\ h = 0.005,\ h = 0.002,$ and $h = 0.001$ corresponding to 100, 200, 500, 1000, 2000, 5000, and 10000 steps, respectively. The results are tabulated in Table 6.4.

h	RK3	IEX3	BI3
0.1	unstable	0.6782e-4	0.4947e-6
0.05	unstable	0.8668e-5	0.2895e-7
0.02	unstable	0.5611e-6	0.1324e-8
0.01	0.7034e-7	0.7029e-7	0.2070e-8
0.005	0.8954e-8	0.8791e-8	0.2116e-8
0.002	0.2219e-8	0.2145e-8	0.2120e-8
0.001	0.2127e-8	0.2119e-8	0.2120e-8

h	AB3	ABM3	AM3	BDF3
0.1	unstable	unstable	unstable	garbage
0.05	unstable	unstable	unstable	garbage
0.02	unstable	unstable	unstable	garbage
0.01	unstable	0.6996e-7	unstable	garbage
0.005	0.7906e-7	0.8772e-8	0.8783e-8	0.9469e-2
0.002	0.5427e-8	0.2156e-8	0.2149e-8	0.1742e-6
0.001	0.2239e-8	0.2120e-8	0.2120e-8	0.4363e-7

TABLE 6.4. Comparison of accuracy of integration algorithms.

Using a step size of $h = 0.001$, all seven integration algorithms simulate the problem successfully. In fact, all of them with the exception of BDF3 are down to the level of the consistency error.

As the step size becomes smaller, the higher–order terms in the Taylor–series expansion become less and less important. For sufficiently small step sizes, all integration algorithms behave either like forward or backward Euler.

BDF3 performs a little poorer than the other algorithms, because its *error coefficient* is considerably larger than those of its competitors. BDF algorithms perform generally somewhat poor in terms of accuracy in comparison with their peers of equal order. The BDF algorithms had been known before they were made popular by Bill Gear in the early seventies [6.9]. However, they were considered "garbage algorithms" due to their poor accuracy properties.

It turns out that the problem is kind of "stiff," although it does not meet most of John Lambert's definitions of stiffness [6.11]. The problem is "stiff" in the sense that all the algorithms with stability domains looping into the left–half plane are unable to produce solutions with the desired accuracy of 1.0%, since they are numerically unstable when a step size is used that would produce the desired accuracy otherwise. BDF3 doesn't suffer the same fate, but it eventually succumbs to error accumulation problems. As the step sizes grow too big, the computations become so inaccurate that the simulation error exceeds the simulation output in magnitude. Hence

BDF3 starts accumulating numerical garbage.

Only IEX3 and BI3 are capable of solving the problem successfully for large step sizes. Between the two, BI3 seems to work a little better, which is no big surprise. Being an *F–stable algorithm*, BI3 is earmarked for these types of applications.

Figure 6.12 presents the same results graphically in a cost *vs.* accuracy plot.

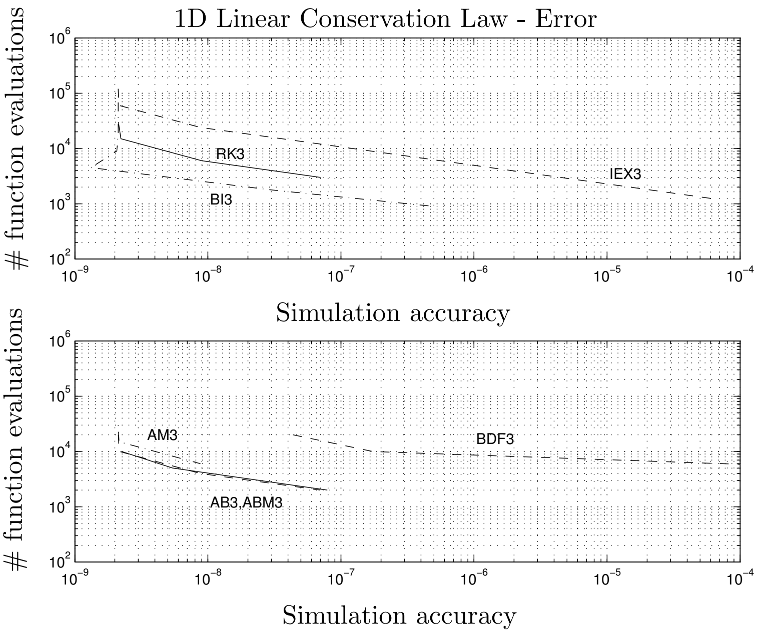


FIGURE 6.12. Cost *vs.* accuracy of the 1D wave equation.

These results are somewhat deceiving, since they do not take into account the effort spent in computing inverse Hessians. This decision was taken on purpose, since the number of function evaluations is the only objective measure available that depends on the algorithm alone, rather than on implementational details of the production code, as different codes vary a lot in how often and how accurately they compute inverse Hessians.

Of course, since the given problem is linear and since we don't vary the step size ever, it would suffice to compute one inverse Hessian at the beginning of the simulation. Yet, this fact is peculiar to the specific problem at hand. For nonlinear problems, the explicit algorithms, i.e., RK3, AB3, and ABM3, may be at least as attractive as BI3.

We would still argue in favor of the BI algorithms for these types of applications, not because of their superior cost–per–accuracy properties, but because of their better *robustness* characteristics. Using BI3, we can obtain a decent answer using any step size that we may try without having the algorithm blow up on us, and we get a meaningful accuracy in each and every case.

We could have included also GE3 in the comparison of this section. Since the problem to be solved is a linear conservation law, the stand–alone versions of the explicit Godunov schemes would have been excellently suited for the task at hand. However, we decided against doing so, because the comparison would have been quite unfair. All of the techniques compared against each other in this section are *general–purpose* numerical ODE solvers, whereas the stand–alone versions of the GE algorithms are limited to dealing with linear conservation laws only.

6.5 Shock Waves

Let us now study a more involved hyperbolic PDE problem. A thin tube of length 1 m is initially pressurized at $p_B = 1.1$ atm. The tube is located at sea level, i.e., the surrounding atmosphere has a pressure of $p_0 = 1.0$ atm $=$ 760.0 Torr $= 1.0132 \cdot 10^5$ N m^{-2}. The current temperature is $T = 300.0$ K. At time zero, the tube is opened at one of its two ends. We wish to determine the pressure at various places inside the tube as functions of time.[1]

As the tube is opened, air rushes out of the tube, and a *rarefaction wave* enters the pipe. Had the initial pressure inside the pipe been smaller than the outside pressure, air would have rushed in, and a *compression wave* would have formed.

The problem can be mathematically described by a set of first–order hyperbolic PDEs:

$$\frac{\partial \rho}{\partial t} = -v \cdot \frac{\partial \rho}{\partial x} - \rho \cdot \frac{\partial v}{\partial x} \tag{6.59a}$$

$$\frac{\partial v}{\partial t} = -v \cdot \frac{\partial v}{\partial x} - \frac{a}{\rho} \tag{6.59b}$$

$$\frac{\partial p}{\partial t} = -v \cdot a - \gamma \cdot p \cdot \frac{\partial v}{\partial x} \tag{6.59c}$$

$$a = \frac{\partial p}{\partial x} + \frac{\partial q}{\partial x} + f \tag{6.59d}$$

$$q = \begin{cases} \beta \cdot \delta x^2 \cdot \rho \cdot \left(\frac{\partial v}{\partial x}\right)^2 \quad ; \quad \frac{\partial v}{\partial x} < 0.0 \\ 0.0 \quad ; \quad \frac{\partial v}{\partial x} \geq 0.0 \end{cases} \tag{6.59e}$$

$$f = \frac{\alpha \cdot \rho \cdot v \cdot |v|}{\delta x} \tag{6.59f}$$

[1]The problem can be found in a slightly modified form in the FORSIM–VI manual [6.4]. It is being reused here with the explicit permission by the author.

where $\rho(x,t)$ denotes the *gas density* inside the tube at position x and time t, $v(x,t)$ denotes the *gas velocity*, and $p(x,t)$ denotes the *gas pressure*. The quantity a was pulled out into a separate algebraic equation, since the same quantity is used in two places within the model. The two quantities computed in Eqs.(6.59e–f) are artificial, as their dependence on δx shows. Clearly, δx is not a physical quantity, but is introduced only in the process of converting the (small) set of PDEs into a (large) set of ODEs. q denotes the *pseudo viscous pressure*, and f denotes the *frictional resistance*. They were introduced by Richtmyer and Morton [6.14] in order to smoothen out numerical problems with the solution. We shall discuss this issue in due course. γ is the ratio of specific heat constants, a non–dimensional constant with a value of $\gamma = c_p/c_v = 1.4$. α and β are non–dimensional numerical fudge factors. We shall initially assign the following values to them: $\alpha = \beta = 0.1$. The "ideal" (i.e., undamped) problem has $\alpha = \beta = 0.0$.

Introduction of the two dissipative terms is not a bad idea, since the "ideal" solution does not represent a physical phenomenon in any true sense. Phenomena without any sort of dissipation belong allegedly in the world that we may enter after we die. They certainly don't form any part of *this* universe.

The initial conditions are:

$$\rho(x, t = 0.0) = \rho_B \tag{6.60a}$$
$$v(x, t = 0.0) = 0.0 \tag{6.60b}$$
$$p(x, t = 0.0) = p_B \tag{6.60c}$$

where ρ_B is determined by the *equation of state* for ideal gases (cf. Chapter 9 of the companion book *Continuous System Modeling* [6.5]):

$$\rho_B = \frac{p_B \cdot M_{\text{air}}}{R \cdot T} \tag{6.61}$$

where $T = 300.0$ is the absolute temperature (measured in Kelvin), $R = 8.314$ J K^{-1} mole^{-1} is the gas constant, and $M_{\text{air}} = 28.96$ g mole^{-1} is the average molar mass of air.[1] The boundary conditions are:

$$v(x = 0.0, t) = 0.0 \tag{6.62a}$$
$$\rho(x = 1.0, t) = \rho_0 \tag{6.62b}$$
$$\begin{cases} v(x = 1.0, t) = -\sqrt{\frac{2(p_0 - p(x=1.0,t)}{\rho(x=1.0,t)}} \;\; ; \;\; v(x = 1.0, t) < 0.0 \\ p(x = 1.0, t) = p_0 \;\; ; \;\; v(x = 1.0, t) \geq 0.0 \end{cases} \tag{6.62c}$$

[1]Air consists roughly to 78% of nitrogen (N_2) with a molar mass of 28 g mole^{-1}, to 21% of oxygen (O_2) with a molar mass of 32 g mole^{-1}, and to 1% of argon (Ar) with a molar mass of 40 g mole^{-1}.

As proposed in [6.4], we converted all spatial derivatives by means of second–order accurate central differences using the formula:

$$\left.\frac{\partial u}{\partial x}\right|_{x=x_i} \approx \frac{1}{2\delta x} \cdot (u_{i+1} - u_{i-1}) \tag{6.63}$$

except near the boundaries, where we used second–order accurate biased formulae:

$$\frac{\partial u}{\partial x}(x = x_1, t) \approx \frac{1}{2\delta x} \cdot (-u_3 + 4u_2 - 3u_1) \tag{6.64a}$$

$$\frac{\partial u}{\partial x}(x = x_{n+1}, t) \approx \frac{1}{2\delta x} \cdot (3u_{n+1} - 4u_n + u_{n-1}) \tag{6.64b}$$

where u can stand for either ρ, v, p, or q.

In order to keep the consistency error small, we chose 50 segments for each of the three PDEs. We created a MATLAB function:

$$\mathbf{u_x} = \text{partial}(\mathbf{u}, \delta x, bc, bctype) \tag{6.65}$$

which implements the above set of formulae with correction terms in the case of a *symmetry boundary condition.* The variable bc indicates whether the boundary condition is applied at the left end, $bc = -1$, or at the right end, $bc = +1$. The variable $bctype$ specifies the type of boundary condition. $bctype = 0$ indicates a symmetry boundary condition. $bctype = 1$ denotes a function value condition.

In the case of a symmetry boundary condition, the central formulae are used all the way to the boundary while folding the values that are outside the domain back into the domain, as explained earlier.

The correction formulae are:

$$\frac{\partial u}{\partial x}(x = x_1, t) \approx 0.0 \tag{6.66}$$

for a symmetry boundary condition at the left end, and:

$$\frac{\partial u}{\partial x}(x = x_{n+1}, t) \approx 0.0 \tag{6.67}$$

for a symmetry boundary condition at the right end.

The state–space model itself has been encoded in another MATLAB function:

```
function [xdot] = st_eq(x, t)
   %
   % State − space model of shock − tube problem
   %
   n = round(length(x)/3);
   n1 = n + 1;
   δx = 1/n;
```

```
%
% $Constants$
%
$R\ =\ 8.314;$
%
% $Physical\ parameters$
%
$Temp\ =\ 300;$
$Mair\ =\ 0.02896;$
$p_0\ =\ 1.0132e5;$
$\rho_0\ =\ p_0 * Mair/(R * Temp);$
$\gamma\ =\ 1.4;$
%
% $Fudge\ factors$
%
global $\alpha\ \beta$
%
% $Unpack\ individual\ state\ vectors\ from\ total\ state\ vector$
%
$\rho\ =\ [\ x(1:n)\ ;\ \rho_0\ ];$
$v\ =\ [\ 0\ ;\ x(n_1 : 2*n)\ ];$
$p\ =\ x(n_1 + n : n_1 + 2*n);$
%
% $Calculate\ nonlinear\ boundary\ condition$
%
if $v(n1)\ <\ 0,$
   $v(n_1)\ =\ -\mathrm{sqrt}(\max([2*(p_0\ -\ p(n_1))/\rho(n_1), 0]));$
else
   $p(n_1)\ =\ p_0;$
end
%
% $Calculate\ spatial\ derivatives$
%
$\rho_x\ =\ \mathrm{partial}(\rho, \delta x, +1, +1);$
$v_x\ =\ \mathrm{partial}(v, \delta x, -1, +1);$
$p_x\ =\ \mathrm{partial}(p, \delta x, +1, +1);$
%
% $Calculate\ algebraic\ quantities$
%
$f\ =\ \alpha * (\rho\ .*\ v\ .*\ \mathrm{abs}(v))/\delta x;$
$q\ =\ \mathrm{zeros}(n1, 1);$
for $i = 1 : n_1,$
   if $v_x(i)\ <\ 0,$
      $q(i)\ =\ \beta * (\delta x^2) * \rho(i) * (v_x(i)^2);$
   end,
end
$q_x\ =\ \mathrm{partial}(q, \delta x, -1, +1);$
$a\ =\ p_x\ +\ q_x\ +\ f;$
%
% $Calculate\ temporal\ derivatives$
%
$\rho_t\ =\ -(v\ .*\ \rho_x)\ -\ (\rho\ .*\ v_x);$
$v_t\ =\ -(v\ .*\ v_x)\ -\ (a\ ./\ \rho);$
```

```
    $p_t \;=\; -(v\ .*\ a)\ -\gamma*(p\ .*\ v_x);$
    %
    % Pack individual state derivatives into total state derivative vector
    %
    $xdot \;=\; [\ \rho_t(1:n)\ ;\ v_t(2:n1)\ ;\ p_t\ ];$
return
```

The resulting set of 151 nonlinear ODEs was simulated across 0.01 sec using the RKF4/5 algorithm, as we learnt that RK algorithms are expected to perform decently when faced with nonlinear hyperbolic PDE problems converted to sets of ODEs by the MOL approach.

This time around, we used all the bells and whistles and included step–size control in time. The results of this simulation are shown in Fig.6.13.

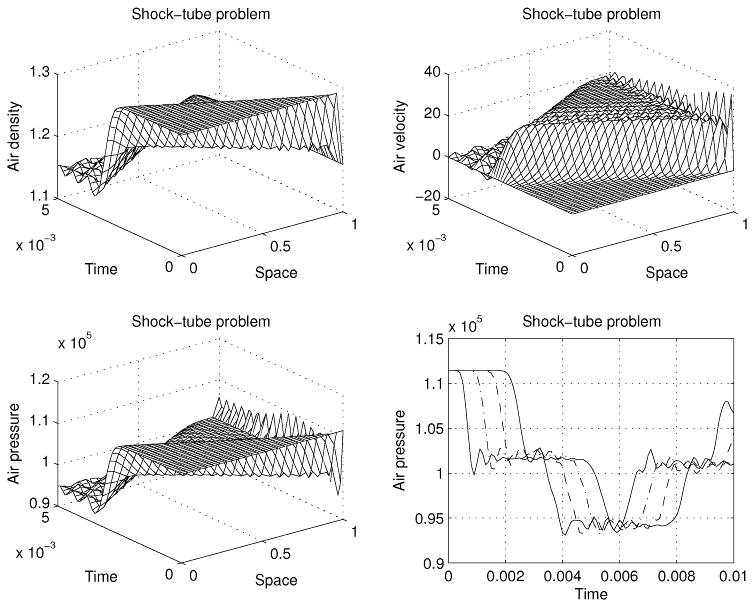


FIGURE 6.13. Shock tube simulation.

The first three graphs depict $\rho(x,t)$, $v(x,t)$, and $p(x,t)$. The solutions look like the water falls of the Iguazu looked at from the Argentinean side of the river. The bottom left parts of all three functions look dangerously irregular in shape. Are the simulation results inaccurate?

The bottom right curve shows the air pressure as a function of time. The solid curve depicts the pressure 20 cm away from the closed end, the dashed line shows the pressure 40 cm away, the dot–dashed line 60 cm away, and the dotted line 80 cm away.

As the end of tube opens, the point closest to the opening experiences the rarefaction wave first. The points further into the tube experience the wave later. From Fig.6.13, it can be concluded that the wave travels through the tube with a constant wave–front velocity of roughly 35 cm per 0.001 sec, or 350.0 m $\sec^{-1}$. This is the correct value of the velocity of sound at sea level and at a temperature of $T = 300$ K. Thus, our simulation seems to be working fine. (There is nothing more healthy in simulation of physical systems than a little reality check once in a while!)

As the rarefaction wave reaches the closed end of the tube, the inertia of the flowing air creates a vacuum. The air flows further, but cannot be replaced by more air from the left. Consequently, the air pressure now sinks below that of the outside air.

As the vacuum reaches the open end of the tube, a new wave is created, this time a *compression wave*, that races back into the tube.

We ended the simulation at $t = 0.01$ sec, since shortly thereafter, the Runge–Kutta algorithm would finally give up on us, and die with an error message.

How accurate are these simulation results? To answer this question, we repeated the simulation with 100 segments. The simulated air pressure at the center of the tube, $x = 50$ cm, is shown in Fig.6.14. For comparison, the results of the 50–segment simulation are superposed on the same graph. As the model itself depends explicitly on the grid width, we set $\alpha = \beta = 0.0$ for this experiment. In this way, the explicit (artificial) dependence of the model on the grid width is eliminated.

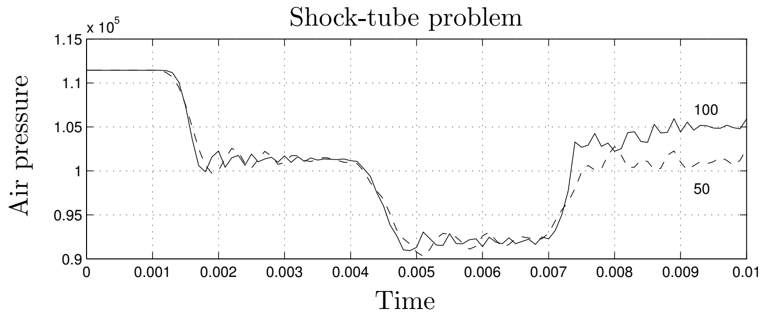


FIGURE 6.14. Consistency error for shock tube simulation.

The simulation results are visibly different. Moreover, the differences seem to grow over time. Is this a consistency error, or simply the result of an inaccurate simulation?

To answer this question, we repeated the same experiment, this time using a different integration algorithm. The F–stable Backinterpolation technique is supposed to work at least as well as the RK algorithm.

The simulation results are indistinguishable by naked eye. Whereas the largest relative distance between the air pressure with 50 and 100 segments:

$$err = \frac{\max(\max(\text{abs}(p_{100} - p_{50})))}{\max([\|p_{100}\|, \|p_{50}\|])} \tag{6.68}$$

is $err = 7.5726e - 4$, the largest relative distance between the air pressure with 50 segments comparing the two different integration algorithms is $err = 1.2374e - 7$, and with 100 segments, it is $err = 6.3448e - 7$.

Hence the *simulation error* is smaller than the *consistency error* by three orders of magnitude. Evidently, we are not faced with a *simulation problem* at all, but rather with a *modeling problem.* The simulation is as accurate as can be expected.

The BI4 algorithm is considerably less efficient than the RKF4/5 algorithm in simulating this problem. Its inefficiency is not caused by the step size. In fact, the step–size controlled BI4 algorithm can make use of step sizes that are quite a bit larger than those used by RKF4/5. The inefficiency is caused by the computation of the Jacobians and of the inverse Hessians.

Since the problem is nonlinear, the Jacobians need to be numerically estimated, using an algorithm such as:

```
**function** $[J] = \text{jacobian}(x, t)$
  %
  % *Jacobian of shock − tube problem*
  %
  $n = \text{length}(x);$
  $J = \text{zeros}(n, n);$
  $xd_{ref} = \text{st\_eq}(x, t);$
  **for** $i = 1 : n,$
    $x_{new} = x;$
    **if** $\text{abs}(x(i)) < 1.0e - 6,$
      $x_{new}(i) = 0.05;$
    **else**
      $x_{new}(i) = 1.05 * x(i);$
    **end**,
    $xd_{new} = \text{st\_eq}(x, t);$
    $J(:, i) = (xd_{new} - xd_{ref})/(x_{new}(i) - x(i));$
  **end**
**return**
```

Thus, every single Jacobian, which is being computed once per integration step, requires 152 additional function evaluations in the case of a 50–segment simulation, and 302 additional function evaluations in the case of a 100–segment simulation. No wonder that production codes of implicit ODE solvers are frugal in the frequency of Jacobian evaluations.

The Hessian is of the same size as the Jacobian:

$$\mathbf{H} = \mathbf{I}^{(\mathbf{n})} + \mathbf{J} \cdot \bar{h} + \frac{1}{2!} \cdot (\mathbf{J} \cdot \bar{h})^2 + \frac{1}{3!} \cdot (\mathbf{J} \cdot \bar{h})^3 + \frac{1}{4!} \cdot (\mathbf{J} \cdot \bar{h})^4 \tag{6.69}$$

where $\bar{h} = -h/2$ is the step size of the right half–step of the BI4 algorithm.

The Hessian is used in a Gauss elimination step once per iteration step:

```
**while** $err_2 > 0.1 * tol$,
    $[x_{right4}, x_{right5}] = \text{rkf45\_step}(x_{new}, t_{new}, -h/2)$;
    $n_{fct} = n_{fct} + 6$;
    $x_{new} = x_{new} - H\backslash(x_{right4} - x_{left4})$;
    $err_2 = \text{norm}(x_{right4} - x_{left4}, '\text{inf}')/\max([\text{norm}(x_{left4}), \text{norm}(x_{right4}), tol])$;
**end**
```

The computational burden of these algorithms is atrocious. We shall have to do something about the size of these matrices. This problem shall be tackled in the next chapter of this book.

What can we do to reduce the consistency error? From our previous observation, we know the answer to this question. If we increase the approximation order of the spatial derivatives by two, the consistency error is expected to decrease by two orders of magnitude.

We modified the *partial* function to use fourth–order accurate central differences instead of the previously used second–order accurate central differences. To this end, the following formulae were now coded into the *partial* function:

$$\left.\frac{\partial u}{\partial x}\right|_{x=x_i} \approx \frac{1}{12\delta x} \cdot (-u_{i+2} + 8u_{i+1} - 8u_{i-1} + u_{i-2}) \tag{6.70}$$

except near the boundaries, where we used fourth–order accurate biased formulae:

$$\frac{\partial u}{\partial x}(x = x_1, t) \approx \frac{1}{12\delta x} \cdot (-3u_5 + 16u_4 - 36u_3 + 48u_2 - 25u_1) \tag{6.71a}$$

$$\frac{\partial u}{\partial x}(x = x_2, t) \approx \frac{1}{12\delta x} \cdot (u_5 - 6u_4 + 18u_3 - 10u_2 - 3u_1) \tag{6.71b}$$

$$\frac{\partial u}{\partial x}(x = x_n, t) \approx \frac{1}{12\delta x} \cdot (3u_{n+1} + 10u_n - 18u_{n-1} + 6u_{n-2} - u_{n-3}) \tag{6.71c}$$

$$\frac{\partial u}{\partial x}(x = x_{n+1}, t) \approx \frac{1}{12\delta x} \cdot (25u_{n+1} - 48 * u_n + 36u_{n-1} - 16u_{n-2} + 3u_{n-3}) \tag{6.71d}$$

In the case of a symmetry boundary condition, the central formulae are used all the way to the boundary while folding the values that are outside the domain back into the domain, as explained earlier.

The correction formulae are:

$$\frac{\partial u}{\partial x}(x = x_1, t) \approx 0.0 \tag{6.72a}$$

$$\frac{\partial u}{\partial x}(x = x_2, t) \approx \frac{1}{12\delta x} \cdot (-u_4 + 8u_3 + u_2 - 8u_1) \tag{6.72b}$$

$$\tag{6.72c}$$

for a symmetry boundary condition at the left end, and:

$$\frac{\partial u}{\partial x}(x = x_n, t) \approx \frac{1}{12\delta x} \cdot (8u_{n+1} - u_n - 8u_{n-1} + u_{n-2}) \tag{6.73a}$$

$$\frac{\partial u}{\partial x}(x = x_{n+1}, t) \approx 0.0 \tag{6.73b}$$

for a symmetry boundary condition at the right end.

We then simulated the system using RKF4/5. Unfortunately, the experiment failed miserably. The integration step size had to be reduced by three orders of magnitude to values around $h = 10^{-8}$, in order to obtain a numerically stable solution, and the results are still incorrect.

What happened? In the previous experiment, the global relative simulation error had been around $err = 10^{-7}$, which is small in comparison with the consistency error, but is still quite large, taking into account that MATLAB computes everything in double precision. With step sizes in the order of $h = 10^{-5}$, we had already sacrificed roughly nine digits to *shiftout.*

In the new experiment with step sizes smaller by three orders of magnitude, we lose at least another three digits to shiftout, i.e., the simulation error is now of the same order of magnitude as the former consistency error. Hence we have not gained anything.

In reality, the problem is even worse. With step sizes that small, the higher order terms of the Taylor–series expansion become irrelevant, and RKF4/5 behaves just like forward Euler. Consequently, also the stability domain of the method shrinks to that of forward Euler, which is totally useless with eigenvalues of the Jacobian spreading up and down along the imaginary axis of the complex $\lambda \cdot h$–plane.

How did BI4 fare in this endeavor? Unfortunately, its destiny is not much better than that of RKF4/5. Remember that BI4 consists of two semi–steps of RKF4/5. With larger step sizes, the left forward RKF4/5 semi–step produces highly unstable x_{left4} values, which the right backward RKF4/5 semi–step needs to stabilize in its Newton iteration.

Unfortunately, it cannot do so, because in the statement:

$$x_{new} = x_{new} - H \backslash (x_{right4} - x_{left4}); \tag{6.74}$$

we subtract a potentially very large number, x_{left4}, from another equally large number, x_{right4}, which again leads to an extreme case of *roundoff.*

With smaller step sizes, the BI4 algorithm degenerates to a forward Euler semi–step followed by a backward Euler semi–step, i.e., to an inefficient implementation of the trapezoidal rule. This is clearly superior to forward Euler alone, since also BI2 is still F–stable, but unfortunately, the semi–steps themselves still suffer from the shiftout problems of the RKF4/5 algorithm, i.e., the simulation error is still of the same order of magnitude as the former consistency error.

Why did all simulation attempts fail after a little more than 0.01 seconds of simulated time? In flow simulations (and in real flow phenomena), it can happen that the top of the wave travels faster than the bottom of the wave. When this happens, the wave will eventually topple over, and at this moment, the wave front becomes infinitely steep. The flow is no longer *laminar*, it has now become *turbulent*.

This is what happens in our shock–tube problem as subsequent versions of rarefaction and compression waves chase after each other back and forth through the tube at ever shorter time intervals. No wonder that the bottom of the three–dimensional plots of the shock–tube simulation look like the bottom of a water fall.

The MOL approach doesn't work for simulating turbulent flows. There exist other simulation techniques (such as the vortex methods [6.12]) that work well for very high Reynolds numbers (above 100 or 1000), and that don't work at all for laminar flows. Reynolds numbers between 1.0 (transition from laminar to turbulent flow) and 100, is where the real research in numerical solution of hyperbolic PDE problems is to be found. Until this day, we don't have any decent simulation methods that can deal appropriately with turbulent flows at low Reynolds numbers.

6.6 Upwind Discretization

In the previous section, we have recognized that hyperbolic PDEs, when converted to sets of ODEs using the MOL approach, lead to systems that share into some of the properties associated with stiff systems, although they do not meet most of the definitions of stiff systems. Yet, the step size had to be often reduced in order to obtain stable solutions when using explicit integration algorithms. In the case of the shock–tube example, the step size reduction was detrimental in that it led to a bad shiftout problem, before the consistency error could be reduced to an insignificantly small value.

How can we stabilize the RK algorithms when dealing with hyperbolic PDEs? One successful idea that was first proposed by Carver and Hinds is to bias the spatial discretization formulae of moving waves in the direction of the provenance of the wave [6.3].

Many wave propagation problems can be formulated in the following way:

$$\frac{\partial u}{\partial t} + v \cdot \frac{\partial u}{\partial x} = 0.0 \tag{6.75}$$

The velocity v determines the direction of flow of the wave. If $v > 0$, the wave moves from left to right. If $v < 0$, it moves from right to left.

The upwind discretization scheme can thus be implemented e.g. as follows:

$$\frac{\partial u}{\partial x}(x = x_i, t) \approx \begin{cases} (3u_i - 4u_{i-1} + u_{i-2})/(2\delta x) & , \quad v \gg 0 \\ (u_{i+1} - u_{i-1})/(2\delta x) & , \quad v \approx 0 \\ (-u_{i+2} + 4u_{i+1} - 3u_i)/(2\delta x) & , \quad v \ll 0 \end{cases} \tag{6.76}$$

if second–order accurate spatial differences are to be used.

Looking once more at the shock–tube problem with $\alpha = \beta = 0.0$:

$$\frac{\partial \rho}{\partial t} = -v \cdot \frac{\partial \rho}{\partial x} - \rho \cdot \frac{\partial v}{\partial x} \tag{6.77a}$$

$$\frac{\partial v}{\partial t} = -v \cdot \frac{\partial v}{\partial x} - \frac{1}{\rho} \cdot \frac{\partial p}{\partial x} \tag{6.77b}$$

$$\frac{\partial p}{\partial t} = -v \cdot \frac{\partial p}{\partial x} - \gamma \cdot p \cdot \frac{\partial v}{\partial x} \tag{6.77c}$$

we notice that all three of these PDEs look like Eq.(6.75), each with a correction term.

We thus encoded the fourth–order accurate upwind formulae in the function:

$$\mathbf{u_x} = \text{upwindv}(\mathbf{u}, \delta x, bc, bctype, \mathbf{fdirv}) \tag{6.78}$$

where **fdirv** is a vector of flow directions, and replaced each occurrence of *partial* in the state equations by *upwindv*, setting the argument *fdirv* as the velocity vector, **v**.

Unfortunately, it didn't work. The shock–tube model discretized using any fourth–order accurate spatial discretization scheme seems to be unstable beyond redemption.

Upwind discretization schemes have become quite fashionable in recent years and come in many different variations. They can be quite effective at times. We still like the original scheme [6.3] best for its simplicity. Yet, there doesn't seem to exist a clean recipe for when and how to use upwind discretization. Sometimes, it helps to only discretize one of several PDEs using an upwind scheme, while discretizing the remaining PDEs using a central difference scheme. What works best can often only be determined by trial and error.

6.7 Grid–width Control

How can we make the solution more accurate without paying too much for it? We already know that it is generally a bad idea to reduce the consistency error by decreasing the grid width. It is much more effective to increase the approximation order of the spatial discretization scheme, whenever possible. Yet, the shock–tube problem has demonstrated that this approach may not always work.

A more narrow grid may be needed in order to accurately compute a wave front. It seems intuitively evident that a more narrow grid width should be used where the absolute spatial gradient is large, thus:

$$\delta x_i(t) \propto \left| \frac{\partial u}{\partial x}(x = x_i, t) \right|^{-1} \tag{6.79}$$

When applied to hyperbolic PDEs, Eq.(6.79) unfortunately suggests use of an *adaptively moving grid*, since the narrowly spaced regions of the grid should follow the wave fronts through space and time.

As we mentioned earlier, naïvely implemented grid–width control is problematic, to say the least. However when implemented carefully, grid–width control can provide an answer to containing the consistency error without leading to either numerical stability problems or at least unacceptably expensive simulation runs. Mack Hyman published some very interesting results on this topic [6.10]. The general gist of his algorithms is the following. We basically operate on a *fixed* grid as before. However, we want to make sure that:

$$\delta x_i(t) \cdot \left| \frac{\partial u}{\partial x}(x = x_i, t) \right| \leq k_{\max} \tag{6.80}$$

at all times. If the absolute spatial gradient grows at some point in space and time, we must reduce the local grid size in order to keep Eq.(6.80) satisfied. We do this by inserting a new auxiliary grid point in the middle between two existing points. We should do this *before* the consistency error grows too large. It thus makes sense to look at the quantity:

$$\frac{1}{h}\left(\left| \frac{\partial u}{\partial x}(x = x_i, t = t_k) \right| - \left| \frac{\partial u}{\partial x}(x = x_i, t = t_{k-1}) \right| \right) \approx \frac{d}{dt}\left(\left| \frac{\partial u}{\partial x}(x = x_i, t) \right| \right) \tag{6.81}$$

If Eq.(6.80) is in danger of not being satisfied any longer and if the temporal gradient of the absolute spatial gradient is positive, we insert a new grid point. On the other hand, if Eq.(6.80) shows a sufficiently small value and if furthermore the temporal gradient is negative, neighboring auxiliary grid points can be thrown out again.

The new grid point solutions are computed using spatial interpolation. These solutions are then used as initial conditions for the subsequent integration of the newly activated differential equations over time. When a grid point is thrown out again, so is the differential equation that accompanies it.

The entire process is completely transparent to the user. Only those solution points are reported for which a solution had been requested. The actually used basic grid width (determined using true grid–width control at time zero) and the auxiliary grid points that are introduced and removed during the simulation run are internal to the algorithm, and the casual user doesn't need to be made aware of their existence. This corresponds to the concept of *communication points* and a *communication interval* discussed in Chapter 4 of this book.

6.8 PDEs in Multiple Space Dimensions

In principle, the MOL methodology can be extended without modification to the case of PDEs in multiple space dimensions. For example, the two–dimensional heat flow problem:

$$\frac{\partial u}{\partial t} = \sigma \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \tag{6.82}$$

discretized using third–order accurate finite difference formulae for both the discretization in the x– and in the y–directions leads to the following ODE at point $x = x_i$ and $y = y_j$:

$$\frac{du_{i,j}}{dt} \approx \sigma \left(\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\delta y^2} \right) \tag{6.83}$$

but the problems are formidable. The first, and most frightening, problem is concerned with the sheer numbers of resulting ODEs. Everything that we wrote about the consistency error still applies. Except for toy problems, we shall certainly need in the order of 50 segments in each space direction, in order to obtain sufficiently smooth output curves. In two space dimensions, this leads to $50 \times 50 = 2500$ ODEs. In the case of three space dimensions, we obtain $50 \times 50 \times 50 = 125,000$ ODEs. Let us assume the differential equation is linear, and we decided to write it in matrix form. The **A**–matrix of the three-dimensional problem consists of $125,000 \times 125,000 = 15,625,000,000$ elements. If you are interested in solving such problems, you better get yourself a *fast* computer and powerful *sparse matrix solvers*. This is the kind of problems for which supercomputers were invented.

The second problem has to do with the distribution of the non–zero elements in the **A**–matrix. Until now, it always happened that the **A**–matrix of a single linear PDE converted by use of finite differences was *band–structured* with a narrow band width. There exist special matrix routines for very efficient handling of band–structured matrices. Unfortunately, the same technique no longer applies to two– and three–dimensional PDEs. Figure 6.15 shows the distribution of non–zero elements in the two–dimensional and three–dimensional heat equations converted to ODEs by means of third–order accurate finite differences using 10 segments in each space dimension. The differential equations were numbered from left to right, from top to bottom, and from front to back, i.e., starting with the last of the three indices. We assumed function value boundary conditions along all edges of the solution cube.

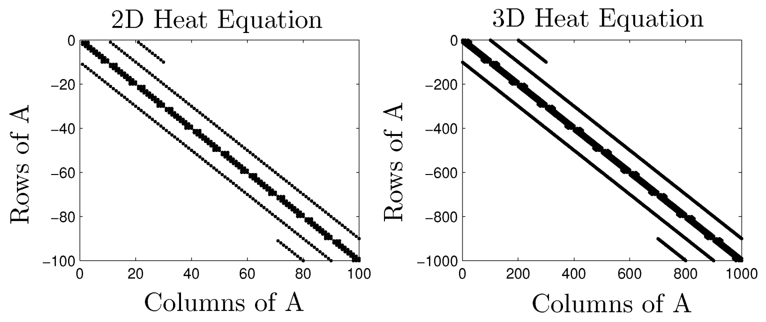


FIGURE 6.15. Distribution of non–zero elements in 2D and 3D heat equations.

Whereas the band width was five in the one–dimensional case, it is $4n+1$ in the two–dimensional case, and $4n^2+1$ in the three–dimensional case. Of course, the precise structure of the **A**–matrix is application dependent. Unfortunately, this means that, when efficiency becomes truly an issue, we may no longer be able to apply the highly efficient algorithms for handling band–structured matrices. General sparse matrix techniques will still work, but they are considerably less efficient than the band–structured algorithms.

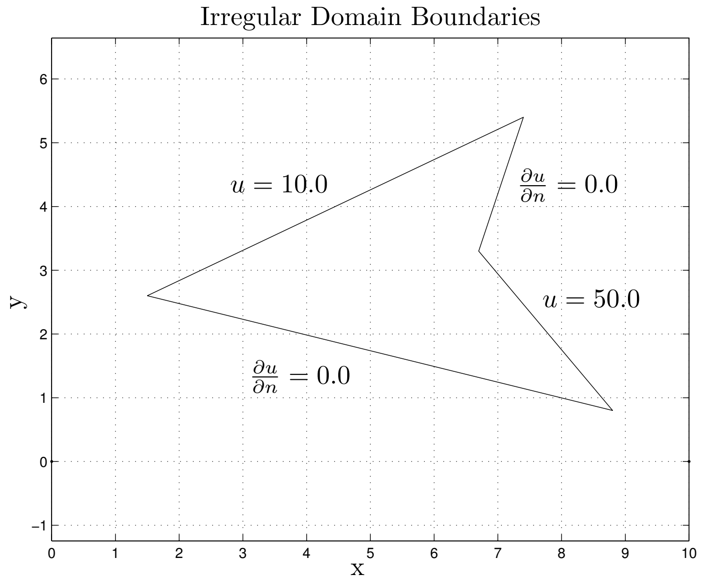
Special algorithms have been designed for renumbering a set of linear equations in such a manner as to minimize the band width of the resulting **A**–matrix. For example the *red–black algorithm* often works well. These algorithms have been described in [6.15].

Unfortunately, we are not at the end of our misery yet. The next problem is illustrated in Fig.6.16.

Figure 6.16 shows a PDE that is defined on an irregularly shaped domain. Until now, we were always able to make the boundary condition coincide with one of the grid points. As Fig.6.16 shows, this may no longer be true in the multidimensional case.

Let us assume that four neighboring values on grid points in x–direction for $y = y_j$ are $u_{1,j}$, $u_{2,j}$, $u_{3,j}$, and $u_{4,j}$. Let us assume further that the boundary value is known at $x = x_{1.35}$ located between x_1 and x_2.

If we know the four solution values $u_{1,j}$, $u_{2,j}$, $u_{3,j}$, and $u_{4,j}$, we can use



Irregular Domain Boundaries
$u = 10.0$
$\frac{\partial u}{\partial n} = 0.0$
$u = 50.0$
$\frac{\partial u}{\partial n} = 0.0$
y
x

FIGURE 6.16. Irregular domain boundaries.

the Nordsieck vector approach presented in Chapter 4 to compute $u_{1.35,j}$. $u_{1.35,j}$ can be expressed as a weighted sum of $u_{1,j}$, $u_{2,j}$, $u_{3,j}$, and $u_{4,j}$. In reality, however, we know $u_{1.35,j}$ (boundary value), and $u_{2,j}$, $u_{3,j}$, and $u_{4,j}$ (through numerical integration). What is unknown is $u_{1,j}$. Thus, we need to solve the previously determined equation for the unknown $u_{1,j}$ instead for the known $u_{1.35,j}$.

To summarize this section: PDEs in one space dimension were still lots of fun. PDEs in multiple space dimensions are painful, to say the least. A large number of applied mathematicians devote their entire academic careers to nothing but solving these types of challenging numerical PDE problems. The purpose of the utterly brief description presented in this section is certainly not to add these specialists to the force of unemployed people, since you, by now, are able to solve all these problems on your own. The purpose of this section is to show you that there are still plenty of *very* challenging research topics around, and to possibly and hopefully wake your appetite for delving more deeply into one or the other of those areas.

6.9 Elliptic PDEs and Invariant Embedding

Equations (6.24) and (6.25) specified what elliptic PDEs are. However, this way of looking at the nature of PDEs is synthetic. People usually don't solve PDEs just for fun. They solve PDEs because they represent physical problems that they are interested in. Physically meaningful parabolic PDEs represent predominantly heat diffusion or chemical reaction problems, and physically meaningful hyperbolic PDEs describe field problems in either hydrodynamics, electromagnetism, optics, general relativity theory, etc. Elliptic PDEs, on the other hand, are used to model stress and strain problems in mechanical structural analysis.

The simplest elliptic PDE is the Laplace equation, e.g. in two space dimensions:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0.0 \tag{6.84}$$

Let us assume the Laplace equation is defined in a circular domain of radius $r = 1.0$ around the origin. Since the domain is circular, it is much more appropriate to formulate the problem using *polar coordinates*.

$$x = r \cdot \cos\varphi \tag{6.85a}$$
$$y = r \cdot \sin\varphi \tag{6.85b}$$

or:

$$r = \sqrt{x^2 + y^2} \tag{6.86a}$$
$$\varphi = \arctan\left(\frac{y}{x}\right) \tag{6.86b}$$

We can express $u(x, y)$ as $\tilde{u}(r(x, y), \varphi(x, y))$. Thus,

$$\frac{\partial u}{\partial x} = \frac{\partial \tilde{u}}{\partial r} \cdot \frac{\partial r}{\partial x} + \frac{\partial \tilde{u}}{\partial \varphi} \cdot \frac{\partial \varphi}{\partial x} \tag{6.87}$$

or, in short–hand notation:

$$u_x = \tilde{u}_r \cdot r_x + \tilde{u}_\varphi \cdot \varphi_x \tag{6.88}$$

Using the chain rule and the multiplication rule, we find:

$$\begin{aligned} u_{xx} + u_{yy} = {} & \left(r_x^2 + r_y^2\right)\tilde{u}_{rr} + 2\left(r_x\varphi_x + r_y\varphi_y\right)\tilde{u}_{r\varphi} + \left(\varphi_x^2 + \varphi_y^2\right)\tilde{u}_{\varphi\varphi} \\ & + \left(r_{xx} + r_{yy}\right)\tilde{u}_r + \left(\varphi_{xx} + \varphi_{yy}\right)\tilde{u}_\varphi \end{aligned} \tag{6.89}$$

or finally:

$$\frac{\partial^2 \tilde{u}}{\partial r^2} + \frac{1}{r} \cdot \frac{\partial \tilde{u}}{\partial r} + \frac{1}{r^2} \cdot \frac{\partial^2 \tilde{u}}{\partial \varphi^2} = 0.0 \tag{6.90}$$

The boundary condition could be something like:

$$\frac{\partial \tilde{u}}{\partial r} = f(\varphi, t) \tag{6.91}$$

Notice that there is no need for any initial condition, since the PDE doesn't depend on time at all (except possibly through the boundary condition as in the above example). No numerical integration across time will take place at all. We are thus in trouble with our MOL methodology.

In some cases, we might still be able to apply the MOL approach by either differentiating along r and integrating along φ, or alternatively, by differentiating along φ and integrating along r. In both cases, however, we would be lacking one initial condition, and would instead have one final condition too many. This is therefore not an *initial value problem*, but rather a *boundary value problem*. We haven't discussed yet how those can be solved.

Does this mean that we have to give up for the time being, or is there a chance that we may turn this problem into one of our known initial value problems after all?

Let us simplify Eq.(6.91) a bit by assuming that the boundary condition does not depend on time. In this case, the problem is totally *static* in nature, i.e., the solution is not time–dependent at all. The solution consists simply of a set of u–values at the grid points.

We can now embed this problem within another problem as follows:

$$\frac{\partial \tilde{u}}{\partial t} = \frac{\partial^2 \tilde{u}}{\partial r^2} + \frac{1}{r} \cdot \frac{\partial \tilde{u}}{\partial r} + \frac{1}{r^2} \cdot \frac{\partial^2 \tilde{u}}{\partial \varphi^2} \tag{6.92}$$

with the boundary condition:

$$\frac{\partial \tilde{u}}{\partial r} = f(\varphi) \tag{6.93}$$

and with arbitrary initial conditions.

This is now clearly a parabolic initial value problem, which we already know how to solve. Since the PDE is analytically stable, and since the boundary condition is not a function of time, the solution will eventually settle into a *steady state*. However, once the steady state has been reached, the solution no longer changes with time, thus:

$$\frac{\partial \tilde{u}}{\partial t} = 0.0 \tag{6.94}$$

Therefore we conclude that the steady–state solution of the parabolic PDE is identical with the solution of the original elliptic PDE. This method of solving elliptic PDEs is called *invariant embedding*.[1] Of course, the price that we have to pay for this comfort is formidable. We were able to convert

[1] A majority of the references spell "imbedding" with an "i" rather than with an "e," probably because the inventor of the method didn't have a dictionary handy when he

a boundary value problem into an initial value problem at the expense of increasing the number of dimensions by one.

6.10 Finite Element Approximations

Those of you who read the companion book *Continuous System Modeling* [6.5] know our reservations against writing down mathematical formulae deprived of their physical meaning. Mathematics is no end in itself. *Mathematics is simply the language of physics.* Voltages and currents in an electronic circuit don't change their values as functions of time, because they observe some differential equations. They change their values in order to bring the system to a state of minimal energy. A differential equation is not the cause that makes physics tick, it is only one way of describing, in mathematical terms and after the fact, what happens in the process of energy exchange taking place in the physical system.

You may remember also that there are two ways of looking at energy conservation laws:

1. We can look at the energy itself. In the most general case, we write down a *Hamiltonian* or possibly a *Hamiltonian field* of the system (at least if the system is conservative), and from there, we can then derive a set of differential equations if we so choose.

2. Rather than looking at the stored energy itself, we can look at incremental energies, i.e., at *power flows*. This leads directly to the *bond graph approach* to modeling that was advocated in Chapters 7–9 of the companion book.

We strongly advocated the latter approach since power flow is a *local property* of the system, whereas energy is a *global property* of the system. Thus, power flow considerations lend themselves directly to an object–oriented approach to modeling.

In *distributed parameter system simulation*, the situation is a little different. As explained earlier, the PDE models that we are dealing with today are still structurally so simple that object orientation is of little concern. Also, especially if we are solving a boundary value problem anyway, as in the case of the elliptic PDEs, we need to solve a global optimization problem over the entire definition domain of the PDE, thus, the advantages of a local model description are gone.

Looking at the solution of the previously discussed Laplace equation, we know that the solution will minimize the amount of energy stored in

wrote his first paper about the method ... but we cannot bring ourselves to follow the trend — it looks so ugly (!)

the system. Consequently, we can write an energy function parameterized in the (unknown) solution values, and solve a minimization problem over the set of unknown parameters. This leads to a set of algebraic equations, possibly nonlinear, in the unknown solution vector.

Approaches that follow this line of reasoning are called *finite element methods*. They come in many shades and colors. The technique was originally developed by civil engineers trying to determine the static stress in bridges and other building structures. However, the method has a much broader range of possible applications. For all practical purposes, it can be viewed as an alternative to the finite difference approaches. Thus, it can conceptually also be used for other than elliptic PDEs.

The two approaches have their own particular advantages and disadvantages. Finite elements usually are less infected by problems with consistency errors than finite difference methods. Consequently, we can get by with a larger (and irregular) mesh, and thus, with a smaller number of equations. On the other hand, finite difference approximations always lead to sparse matrices. Finite element approximations do not share this property. As a consequence, although the number of equations is smaller in the finite element case, we may not be able to use sparse matrix techniques, and it is therefore not evident that the smaller system size truly leads to a more economical algorithm. Also, a finite difference formulation is usually easier to derive and harder to solve than a finite element formulation. However, it is easier to incorporate irregular and even non–convex domain boundaries into a finite element description.

Meanwhile, finite element methods have also been extended to the solution of non–stationary problems by means of a Galerkin formulation [6.17]. Thus, finite elements have suddenly become a contender to finite differences even in the context of the MOL methodology. However, more research in this area is still needed.

6.11 Summary

In this chapter, we have first and primarily discussed the numerical solution of PDEs in one space dimension. The method–of–lines approach lets us reduce such PDEs to large ODE systems that we can solve using regular ODE software.

Parabolic PDEs lead to sets of (artificially) stiff ODEs that can be treated appropriately using stiff system solvers such as the BDF algorithms. Since all of today's continuous–system modeling and simulation environments, such as Dymola [6.7, 6.8], offer stiff system solvers as part of their simulation run–time library, it became clear that they are perfectly capable of dealing with parabolic PDEs in one space dimension. The most cumbersome part in the conversion process was the derivation of the coefficients for the spatial finite difference approximations using Newton–Gregory polynomials, but this process can be easily automated.

Hyperbolic PDEs lead to large sets of marginally stable ODEs that can best be solved by F–stable integration algorithms, such as the backinterpolation techniques. However, explicit algorithms, such as AB3 or RKF4/5 may sometimes work just as well, as they avoid the need of computing expensive Jacobians and inverse Hessians. Hyperbolic PDEs are numerically more demanding than their parabolic cousins due to the occurrence of traveling shock waves. Adaptive moving mesh algorithms can provide a solution to this problem, but then call for special–purpose software, since these algorithms are non–trivial in their implementation. It would be too much of a burden to ask the user to implement such algorithms manually. Yet, powerful modeling environments can make also this process transparent to the modeler.

Elliptic PDEs in one space dimension are no PDEs at all. They are one class of boundary value ODEs, and we shall discuss later in this book how these can be tackled in general. However, one method was already provided here, namely the method of invariant embedding, a method that converts the boundary value ODE into a parabolic PDE in one space dimension, with which we can then proceed as elaborated above.

Multidimensional PDEs were discussed next. Although they can, in principle, be treated in exactly the same manner as their one–dimensional counterparts, the numerical problems are formidable, and efficiency considerations become here an issue of utmost importance.

Does there exist general–purpose PDE software? We had already mentioned the FORSIM–VI software [6.4]. FORSIM–VI is just a Fortran program. No preprocessor is involved at all. The user simply provides a Fortran subroutine describing his or her model. This makes FORSIM inappropriate for use in more complex ODE situations, since not even an equation sorter is offered, lest an object–oriented modeling facility. What makes FORSIM different from any other (simple–minded) ODE simulation system is that FORSIM provides built–in subroutines for converting spatial derivatives into finite difference approximations. These routines know how to compute the necessary coefficients, and consequently, the user doesn't need to worry about Newton–Gregory polynomials. FORSIM works with both equidistantly and non–equidistantly spaced grids. FORSIM also offers built–in routines for converting general and even nonlinear boundary conditions into boundary value conditions. Thus, FORSIM helps the user tremendously with the encoding of his or her PDEs. Routines are available for converting PDEs in one to three space dimensions, however, the two– and three–dimensional routines are not general since they work only on rectangular domains. FORSIM is strictly MOL–oriented. Spatial derivatives are discretized by means of finite difference approximations, whereas temporal derivatives are kept in the program for numerical integration across time. FORSIM offers a Gear (BDF) algorithm for the solution of parabolic problems, and an RKF4/5 algorithm for hyperbolic ones.

A fairly similar software system is DSS/2 [6.16]. The two systems, FORSIM–VI and DSS/2 are in fact so similar that a further discussion of DSS/2 can be skipped.

Other systems, such as PDEL [6.2], went another route. For the benefit of a more finely tuned numerical solution, they sacrificed generality for efficiency. These software systems allow the user to choose between a set of standard frequently occurring PDEs, and then employ different types of (not necessarily MOL) algorithms to solve the problem.

It may be noticed that all of these systems are fairly old. In the early seventies, it was hoped that PDE problems could be solved by general–purpose PDE software just as ODE problems are solved by general–purpose ODE software. This turned out to be an illusion. The ODE situation is *much* simpler. All we need to do is to provide a tool that allows to choose between a set of different numerical integration algorithms, and we are in business. Moreover, it often doesn't matter too much what algorithm we choose. One algorithm may be 30% faster or 20% slower than another, but who cares. Modern PCs have become so powerful that they can effectively and efficiently deal with the simulation of a large majority of lumped parameter models. In contrast, there exist many different techniques to solve PDE problems. Even if we limit our discussion to MOL–solutions, we must choose:

1. a numerical integration algorithm for integration across time,
2. a grid for discretization in space,
3. a numerical discretization scheme for differentiation across space,
4. an algorithm to translate boundary conditions specified at an arbitrary point in space to boundary conditions specified at the nearest grid point
5. an algorithm for converting general boundary conditions to boundary value conditions,

and this is only one among many approaches for numerically solving PDEs. Furthermore, the sensitivity of the solution to the selection of just the right combination of algorithms is much greater in the PDE case than in the ODE case. Selecting one method may mean that we have to wait for 50 hours until we obtain a (hopefully correct) answer, whereas the same problem may be solved by the best possible combination of algorithms in just a few seconds.

For these reasons, general–purpose PDE software hasn't lived up to its promise. The "casual" user of PDE software cannot be protected from having to understand the intricacies of the underlying numerical algorithms, and the numerical solution to all but toy PDE problems is so expensive that it is well worth spending some time on understanding what is going on before starting to crunch numbers. Getting coefficients out of Newton–Gregory polynomials may be but the least of our problems.

The situation is somewhat different in the case of elliptic PDEs. Elliptic PDEs are the simplest and most benign of all PDE problems. An extensive effort was undertaken by John Rice and his colleagues with large amounts of funding through the national agencies to solve that problem once and for all. They designed the ELLPACK software [6.15]. ELLPACK started out as a collection of useful algorithms to solve general–purpose elliptic PDEs in two and three space dimensions.

It turned out that the situation became soon too messy. Casual users no longer could learn to use these algorithms without help from the professional. To remedy the situation, a simple language was designed, and a compiler was written that would translate programs written in that language into a Fortran program that would then invoke the previously discussed algorithms that now form part of the run–time library. Thus, by this time, we are in the same situation as with the continuous–system simulation languages.

It turned out that it didn't work. The approach was too simple–minded. As a new algorithm became available, new keywords had to be added to the language in order to make this new algorithm accessible, and consequently, the compiler had to be updated frequently. This became too much of a hassle to the software designers. So they decided to parameterize the compiler. The compiler was generated out of a data template file that described both syntax and semantics of the ELLPACK language by means of a *compiler–compiler*. So, from now on, new features needed only to be incorporated into the data template file, and a new compiler for the so modified language could be generated at once.

Well, you may already have guessed ... it didn't work. The researchers found the manual generation of the data template file much too cumbersome after all. That problem was taken care of easily. The precise details of the data template file were generated by a *data template compiler* out of a more abstract description of the data template file. Of course, also the data template compiler wasn't hand–coded. Why should it? Instead, the data template compiler was generated out of an abstract description of its duties by the same compiler–compiler that also generates the ELLPACK language compiler. This allows us to also update the data template compiler easily and readily.

At this point in time, only one question remains: Who wrote the compiler–compiler? We assume most of you read the story of Münchhausen who pulls himself out of the swamp by pulling on his own hair ... the compiler–compiler wrote itself. A first (bootstrap) version of the compiler–compiler was hand–coded. This version was already able to read a language description in terms of its syntax and semantics. Well, the first language description it got to read was its own. So, by running the bootstrap compiler–

compiler through a description of itself, a second and much cleaner version of the compiler–compiler was obtained that could subsequently be used to generate new versions of the ELLPACK language compiler, the data template compiler, and –why not– itself.

Was it worth it? As an intellectual stimulus, most certainly. As an experimental toolbox for solving new kinds of elliptic PDEs, probably. As a general–purpose production tool for solving specific PDE problems posed in industry, not likely. We acquired the tool some years ago when we held a contract from the microelectronic industry to design a device simulator that could predict the breakdown behavior of bipolar power transistors (effectively, of any kind of reverse–biased p–n junction). The results that we obtained using ELLPACK were documented in [6.18]. ELLPACK allowed us to fairly quickly and easily go through a number of different algorithms and gain a feeling for which combination of algorithms might work decently well. However, the simulations obtained in this manner were painfully slow. A simple p–n junction milled for an hour or two on a VAX 11/780. More complex devices could not be handled at all within reasonable time limits. Therefore, we then designed our own special–purpose device simulator, ASEPS [6.19]. This program was able to simulate simple p–n junctions in a few seconds of CPU time on the same machine. ASEPS then enabled us to also study more complex device structures such as special geometric configurations of device termination structures for radiation–hardened power MOSFETs [6.6]. These simulation runs took a few minutes each, and optimization studies could be performed in batch mode over night.

Good special–purpose finite element software for structural analysis, such as NASTRAN, has been around for some time. This software doesn't attempt to solve general–purpose elliptic PDEs. Only one type of problem is solved, but the program is very flexible with respect to the specification of the domain on which the problem is to be solved and with respect to the selection of grid points (finite element programs aren't limited to using rectangular grids). Special–purpose numerical PDE solvers exist also for several other classes of applications, such as fluid dynamics.

It is disappointing to a generalist that the general–purpose approach to numerical PDE solution didn't work out. Unfortunately, we don't see any cure yet. Consequently, special–purpose solutions for specific PDE problems will be around for years to come.

6.12 References

[6.1] Kathryn E. Brenan, Stephen L. Campbell, and Linda R. Petzold. *Numerical Solution of Initial–Value Problems in Differential–Algebraic Equations.* North–Holland, New York, 1989. 256p.

[6.2] Alfonso F. Cárdenas and Walter J. Karplus. PDEL — A Language for Partial Differential Equations. *Comm. ACM*, 13:184–191, 1970.

[6.3] Michael B. Carver and H.W. Hinds. The Method of Lines and the Advective Equation. *Simulation*, 31:59–69, 1978.

[6.4] Michael B. Carver, D.G. Stewart, J.M. Blair, and W.M. Selander. The FORSIM VI Simulation Package for the Automated Solution of Arbitrarily Defined Partial and/or Ordinary Differential Equation Systems. Technical Report AECL–5821, Chalk River Nuclear Laboratories, Atomic Energy of Canada Limited, Chalk River, Ontario, Canada., 1978.

[6.5] François E. Cellier. *Continuous System Modeling.* Springer Verlag, New York, 1991. 755p.

[6.6] Kenneth R. Davis, Ronald D. Schrimpf, Kenneth F. Galloway, and François E. Cellier. The Effects of Ionizing Radiation on Power–MOSFET Termination Structures. *IEEE Trans. Nuclear Sci.*, 36(6):2104–2109, 1989.

[6.7] Hilding Elmqvist. *A Structured Model Language for Large Continuous Systems.* PhD thesis, Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1978.

[6.8] Hilding Elmqvist. *Dymola — Dynamic Modeling Language, User's Manual, Version 5.3.* DynaSim AB, Research Park Ideon, Lund, Sweden, 2004.

[6.9] C. William Gear. *Numerical Initial Value Problems in Ordinary Differential Equations.* Series in Automatic Computation. Prentice–Hall, Englewood Cliffs, N.J., 1971. 253p.

[6.10] J. Mack Hyman. Moving Mesh Methods for Partial Differential Equations. In Jerome A. Goldstein, Steven Rosencrans, and Gary A. Sod, editors, *Mathematics Applied to Science: In Memoriam Edward D. Conway*, pages 129–153. Academic Press, Boston, Mass., 1988.

[6.11] John D. Lambert. *Numerical Methods for Ordinary Differential Systems: The Initial Value Problem.* John Wiley, New York, 1991. 304p.

[6.12] R. Ivan Lewis. *Vortex Element Methods for Fluid Dynamic Analysis of Engineering Systems.* Cambridge University Press, New York, 1991. 588p.

[6.13] Anthony Ralston and Herbert S. Wilf. *Mathematical Methods for Digital Computers.* John Wiley & Sons, New York, 1960. 287p.

[6.14] John R. Rice and Ronald F. Boisvert. *Solving Elliptic Problems Using Ellpack.* Springer–Verlag, New York, 1985. 497p.

[6.15] Robert D. Richtmyer and K. William Morton. *Difference Methods for Initial Value Problems.* Wiley Interscience, New York, 1967. 405p.

[6.16] William E. Schiesser. *The Numerical Method of Lines: Integration of Partial Differential Equations.* Academic Press, San Diego, Calif., 1991. 326p.

[6.17] V. Rao Vemuri and Walter J. Karplus. *Digital Computer Treatment of Partial Differential Equations.* Prentice–Hall, Englewood Cliffs, N.J., 1981. 449p.

[6.18] Qiming Wu and François E. Cellier. Simulation of High–Voltage Bipolar Devices in the Neighborhood of Breakdown. *Mathematics and Computers in Simulation*, 28:271–284, 1986.

[6.19] Qiming Wu, Chimin Yen, and François E. Cellier. Analysis of Breakdown Phenomena in High–Voltage Bipolar Devices. *Transactions of SCS*, 6(1):43–60, 1989.

6.13 Bibliography

[B6.1] Myron B. Allen, Ismael Herrera, and George F. Pinder. *Numerical Modeling in Science and Engineering.* John Wiley & Sons, New York, 1988. 418p.

[B6.2] William F. Ames. *Numerical Methods for Partial Differential Equations.* Academic Press, New York, 3^{rd} edition, 1992. 433p.

[B6.3] T. J. Chung. *Computational Fluid Dynamics.* Cambridge University Press, Cambridge, United Kingdom, 2002. 800p.

[B6.4] Peter S. Huyakorn and George F. Pinder. *Computational Methods in Subsurface Flow.* Academic Press, New York, 1983. 473p.

[B6.5] Leon Lapidus and George F. Pinder. *Numerical Solution of Partial Differential Equations in Science and Engineering.* John Wiley & Sons, New York, 1999. 677p.

[B6.6] Robert Vichnevetsky and John B. Bowles. *Fourier Analysis of Numerical Approximations of Hyperbolic Equations.* SIAM Publishing, Philadelphia, Penn., 1982. 140p.

[B6.7] John Keith Wright. *Shock Tubes.* John Wiley & Sons, New York, 1961. 164p.

6.14 Homework Problems

[H6.1] Heat Diffusion in the Soil

Agricultural engineers are interested in knowing the temperature distribution in the soil as a function of the surface air temperature. As shown in Fig.H6.1a, we want to assume that we have a soil layer of 50 cm. Underneath the soil, there is a layer that acts as an ideal heat insulator.

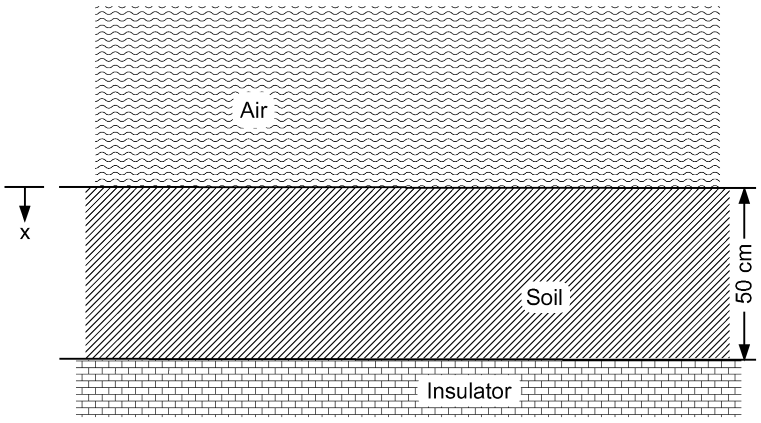


FIGURE H6.1a. Soil topology.

The heat flow problem can be written as:

$$\frac{\partial u}{\partial t} = \frac{\lambda}{\rho \cdot c} \cdot \frac{\partial^2 u}{\partial x^2} \tag{H6.1a}$$

where $\lambda = 0.004$ cal cm^{-1} sec^{-1} K^{-1} is the specific thermal conductance of soil, $\rho = 1.335$ g cm^{-3} is the density of soil, and $c = 0.2$ cal g^{-1} K^{-1} is the specific thermal capacitance of soil.

The surface air temperature has been recorded as a function of time. It is tabulated in Table H6.1a.

We want to assume that the surface soil temperature is identical with the surface air temperature at all times. We want to furthermore assume that the initial soil temperature is equal to the initial surface temperature everywhere.

Specify this problem using hours as units of time, and centimeters as units of space. Discretize the problem using third–order accurate finite differences everywhere. Simulate the resulting linear ODE system using MATLAB. Plot on one graph the soil temperature at the surface and at the insulator as functions of time. Generate also a three–dimensional plot showing the temperature distribution in the soil as a function of time and space.

t [hours]	u °C
0	6
6	16
12	28
18	21
24	18
36	34
48	18
60	25
66	15
72	4

TABLE H6.1a. Surface air temperature.

[H6.2] Electrically Heated Rod

We start out with a simple parabolic partial differential equation describing the temperature distribution in an electrically heated copper rod. This phenomenon can be modeled by the following equation:

$$\frac{\partial T}{\partial t} = \sigma \left(\frac{\partial^2 T}{\partial r^2} + \frac{1}{r} \cdot \frac{\partial T}{\partial r} + \frac{P_{\text{electr}}}{\lambda \cdot V} \right) \tag{H6.2a}$$

The first two terms represent the standard diffusion equation in polar coordinates as described previously in Eq.(6.92), and the constant term describes the electrically generated heat. It can be derived from Fig.8.13 of the companion book on *Continuous System Modeling* [6.5].

$$\sigma = \frac{\lambda}{\rho \cdot c} \tag{H6.2b}$$

is the diffusion coefficient, where $\lambda = 401.0$ J m^{-1} sec^{-1} K^{-1} is the specific thermal conductance of copper, $\rho = 8960.0$ kg m^{-3} is its density, and $c = 386.0$ J kg^{-1} K^{-1} is the specific thermal capacitance.

$$P_{\text{electr}} = u \cdot i \tag{H6.2c}$$

is the dissipated electrical power, and

$$V = \pi \cdot R^2 \cdot \ell \tag{H6.2d}$$

is the volume of the rod with the length $\ell = 1$ m and the radius $R = 0.01$ m. The rod is originally in an equilibrium state at room temperature $T_{\text{room}} = 298.0$ K.

The boundary conditions are:

$$\left.\frac{\partial T}{\partial r}\right|_{r=0.0} = 0.0 \tag{H6.2e}$$

$$\left.\frac{\partial T}{\partial r}\right|_{r=R} = -k_1\left(T(R)^4 - T_{\text{room}}^4\right) - k_2\left(T(R) - T_{\text{room}}\right) \tag{H6.2f}$$

where the quartic term models the heat radiation, whereas the linear term models convective heat flow away from the rod.

We want to simulate this system using the MOL approach with 20 spatial segments (in radial direction), and using second–order accurate finite difference approximations for the first–order spatial derivatives, and third–order accurate finite differences for the second–order spatial derivatives. We are going to treat the boundary condition at the center as a general boundary condition rather than as a symmetry boundary condition in order to circumvent the difficulties with computing the term $(\partial T/\partial r)/r$, which evaluates to $0/0$ at $r = 0.0$.

For internal segments, we obtain thus differential equations of the type:

$$\frac{dT_i}{dt} \approx \sigma\left(\frac{T_{i+1} - 2T_i + T_{i-1}}{\delta r^2} + \frac{1}{r}\cdot\frac{T_{i+1} - T_{i-1}}{2\delta r} + \frac{P_{\text{electr}}}{\lambda \cdot V}\right) \tag{H6.2g}$$

which are straightforward to implement. For the left–most segment, we have the condition:

$$\left.\frac{\partial T}{\partial r}\right|_{r=0.0} = 0.0 \approx \frac{1}{2\delta r}\left(-T_3 + 4T_2 - 3T_1\right) \tag{H6.2h}$$

and therefore:

$$T_1 \approx \frac{4}{3}T_2 - \frac{1}{3}T_3 \tag{H6.2i}$$

Consequently, we don't need to solve a differential equation at $r = 0.0$, and thereby, we skip the $0/0$ division.

At the right–most segment, we obtain:

$$\left.\frac{\partial T}{\partial r}\right|_{r=R} = -k_1\left(T_{21}^4 - T_{\text{room}}^4\right) - k_2\left(T_{21} - T_{\text{room}}\right) \approx \frac{1}{2\delta r}\left(-3T_{21} + 4T_{20} - T_{19}\right) \tag{H6.2j}$$

Thus, we obtain a nonlinear equation in the unknown T_{21}:

$$\begin{aligned}\mathcal{F}(T_{21}) &= k_1\left(T_{21}^4 - T_{\text{room}}^4\right) + k_2\left(T_{21} - T_{\text{room}}\right) - \frac{1}{2\delta r}\left(-3T_{21} + 4T_{20} - T_{19}\right) \\ &\approx 0.0\end{aligned} \tag{H6.2k}$$

which can be solved by Newton iteration:

$$\begin{aligned} T_{21}^0(t) &= T_{21}(t-h) && \text{(H6.2l)} \\ T_{21}^1(t) &= T_{21}^0(t) - \frac{\mathcal{F}(T_{21}^0)}{\mathcal{H}(T_{21}^0)} && \text{(H6.2m)} \\ T_{21}^2(t) &= T_{21}^1(t) - \frac{\mathcal{F}(T_{21}^1)}{\mathcal{H}(T_{21}^1)} && \text{(H6.2n)} \\ & \text{until convergence} \end{aligned}$$

where:

$$\mathcal{H}(T_{21}) = \frac{\partial \mathcal{F}}{\partial T_{21}} = 4k_1 T_{21}^3 + k_2 - \frac{3}{2\delta r} \qquad \text{(H6.2o)}$$

[H6.3] Wave Equation

The wave equation has been written as:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \cdot \frac{\partial^2 u}{\partial x^2} \qquad \text{(H6.3a)}$$

Let us rewrite $u(x,t)$ as $\tilde{u}(v,w)$, where:

$$v = t + x \qquad \text{(H6.3b)}$$

$$w = t - x \qquad \text{(H6.3c)}$$

What happens?

[H6.4] Shock Tube Simulation

We wish to analyze the influence of α and β on the accuracy of the simulation. Repeat the same 50 segment simulation with different values for α and β. What do you conclude about the relative influence of α and β in comparison with the consistency error. Assuming a small value of α, which is the largest value of β acceptable before the relative error exceeds 1%. Similarly, assuming a small value of β, which is the largest value of α acceptable before the relative error exceeds 1%.

Use one half the maximum values of α and β found above, and simulate across a longer period of time. Can you reach steady–state? Interpret the results.

[H6.5] River Bed Simulation

Hydrologists are interested in determining the movement of river beds with time. The dynamics of this system can be described through the PDE:

$$\frac{\partial v}{\partial t} + v \cdot \frac{\partial v}{\partial x} + g \cdot \frac{\partial h}{\partial x} + g \cdot \frac{\partial z}{\partial x} = w(v) \qquad \text{(H6.5a)}$$

where $v(x,t)$ is the absolute value of the flow velocity of the water, $h(x,t)$ is the water depth, and $z(x,t)$ is the altitude of the river bed relative to an arbitrary constant level. $g = 9.81$ m sec^{-2} is the gravitational constant.

$w(v)$ is the friction of water at the river bed:

$$w(v) = -\frac{g \cdot v^2}{s_k^2 \cdot h^{4/3}} \tag{H6.5b}$$

where $s_k = 32.0$ m$^{3/4}$ sec^{-1} is the Strickler constant.

The continuity equation for the water can be written as:

$$\frac{\partial h}{\partial t} + v \cdot \frac{\partial h}{\partial x} + h \cdot \frac{\partial v}{\partial x} = 0.0 \tag{H6.5c}$$

and the continuity equation for the river bed can be expressed as:

$$\frac{\partial z}{\partial t} + \frac{df(v)}{dv} \cdot \frac{\partial v}{\partial x} = 0.0 \tag{H6.5d}$$

where $f(v)$ is the transport equation of Meyer–Peter simplified by means of regression analysis:

$$f(v) = f_0 + c_1(v - v_0)^{c_2} \tag{H6.5e}$$

where $c_1 = 1.272 \cdot 10^{-4}$ m, and $c_2 = 3.5$.

We want to study the Rhine river above Basel over a distance of 6.3 km. We want to simulate this system across 20 days of simulated time. The initial conditions are tabulated in Table H6.5a.

x [m]	v [m/s]	h [m]	z [m]
0.0	2.3630	3.039	59.82
630.0	2.3360	3.073	59.06
1260.0	2.2570	3.181	58.29
1890.0	1.6480	4.357	56.75
2520.0	1.1330	6.337	54.74
3150.0	1.1190	6.416	54.60
3780.0	1.1030	6.509	54.45
4410.0	1.0620	7.001	53.91
5040.0	0.8412	8.536	52.36
5670.0	0.7515	9.554	51.33
6300.0	0.8131	8.830	52.02

TABLE H6.5a. Initial data for river bed simulation.

We need three boundary conditions. We want to assume that the amount of water $q = h \cdot v$ entering the simulated river stretch is constant. For simplicity, we shall assume both h and v constant. At the lower end, there is a weir. Therefore, we can assume that the sum of z and h is constant at the lower end of the simulated stretch of river. Since the water moves much faster than the river bed, it doesn't make too much sense to apply boundary conditions to the river bed.

This system is pretty awful. The time constants of the water are measured in seconds, whereas those of the ground are measured in days. We are interested in the slow time constant, yet it is the fast time constant that dictates the integration step size. We can think of the first two PDEs as a nonlinear function generator for the third PDE. Let us therefore modify Eq.(H6.5d) as follows:

$$\frac{\partial z}{\partial t} + \beta \cdot \frac{df(v)}{dv} \cdot \frac{\partial v}{\partial x} = 0.0 \tag{H6.5f}$$

The larger we choose the tuning parameter, the faster will the river bed move. Select a value somewhere around $\beta = 100$ or even $\beta = 1000$. Later, we must analyze the damage that we did to the PDE system by introducing this tuning parameter. Maybe, we can extrapolate to the correct system behavior at $\beta = 1.0$.

The third boundary condition is analytically correct, but numerically not very effective since it is specified at the wrong end of the system. Since water always flows downhill, a boundary condition at the bottom is about as effective as commanding my dog to solve this homework problem. Let us therefore introduce yet another boundary condition at the top end:

$$\frac{\partial z}{\partial x} = constant \tag{H6.5g}$$

However, since we cannot specify a derivative boundary condition for a first–order equation, we reformulate Eq.(H6.5g) as:

$$z_1 = z_2 + constant \tag{H6.5h}$$

Plot the river bed altitude $z(x)$ measured at the end of every five day period superposed onto one graph.

Rerun the simulation for different values of β. Is it possible to extrapolate what the solution would look like for $\beta = 1.0$?

[H6.6] Boundary Value Conversion

A PDE in one space dimension is specified in the range [0.0,1.0] with $\delta x = 0.1$. Unfortunately, one of the boundary values if given as: $u(x = 0.98, t) = f(t)$.

We want to translate this boundary value to an equivalent boundary value at $u(x = 1.0, t)$. Use the Nordsieck vector approach to come up with a third–order accurate equation for $u(x = 1.0, t)$ as a function of $u(x = 0.98, t)$, $u(x = 0.9, t)$, $u(x = 0.8, t)$, and $u(x = 0.7, t)$.

[H6.7] Coordinate Transformation

Verify that Eq.(6.92) is indeed correct.

[H6.8] Coordinate Transformation

We wish to solve the Laplace equation for diffusion along the surface of a globe, assuming that no diffusion takes place in radial direction. To this end, we start out with the three–dimensional Laplace equation:

$$\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}+\frac{\partial^2}{\partial z^2}\right)u(x,y,z)=0.0 \tag{H6.8a}$$

We want to rewrite this Laplace equation as a function of three different coordinates:

$$u(x,y,z)=\tilde{u}(\rho,\xi,\eta) \tag{H6.8b}$$

where ρ is the radius of the globe, ξ is the longitude, and η is the latitude. We obtain a modified Laplace equation in these coordinates. We then specify that:

$$\frac{\partial^2\tilde{u}}{\partial\rho^2}=\frac{\partial\tilde{u}}{\partial\rho}=0.0 \tag{H6.8c}$$

It is easy to make mistakes in such transformations. We therefore want to check whether the result is at least potentially correct. To this end, we let $\rho\to\infty$. Obviously, this must give us the original Laplace equation back, now expressed in ξ and η instead of x and y.

[H6.9] Poiseuille Flow Through a Pipe

The following equations describe the stationary flow of an incompressible fluid through a pipe:

$$\frac{d\hat{v}}{d\rho}=\frac{-\sqrt{2\Gamma}}{(\tau_M+1)^2}\cdot\rho\cdot\tau^2 \tag{H6.9a}$$

$$\frac{d}{d\rho}\left(\frac{\rho}{T}\cdot\frac{d\tau}{d\rho}\right)=\frac{-\Gamma}{(\tau_M+1)^3}\cdot\rho^3\cdot\tau^2 \tag{H6.9b}$$

where:

$$\rho=\frac{r}{R} \tag{H6.9c}$$

$$\tau=\frac{T(r)}{T_W} \tag{H6.9d}$$

are two normalized coordinates. r is the distance from the center of the pipe, and R is the radius of the pipe. $T(r)$ is the temperature of the fluid at a distance r from the center, and T_W is the temperature of the pipe wall. T_W is assumed constant. $\hat{v}=k_1 * v$ is the normalized flow velocity, where

k_1 is a constant that depends on the viscosity, the thermal conductivity, and the average temperature of the fluid.

The boundary conditions are:

$$\frac{d\hat{v}}{d\rho}(\rho = 0.0) = 0.0 \tag{H6.9e}$$

$$\frac{d\tau}{d\rho}(\rho = 0.0) = 0.0 \tag{H6.9f}$$

$$\hat{v}(\rho = 1.0) = 0.0 \tag{H6.9g}$$

$$\tau(\rho = 1.0) = 1.0 \tag{H6.9h}$$

Thus, this is a boundary value problem. We could integrate this problem across ρ in the range $\rho = [0.0, 1.0]$ with unknown initial conditions $\hat{v}(\rho = 0.0) = \hat{v}_M$ and $\tau(\rho = 0.0) = \tau_M$.

However, in the light of what we learnt in this chapter, we shall try another approach. We embed this boundary value problem into a parabolic PDE, which we solve with arbitrary initial conditions until we reach steady–state.

Notice that the equations contain two yet unknown parameters. Γ is a constant that depends on the fluid. Let us assume that $\Gamma = 10.0$. τ_M is the value of the normalized temperature at the center of the pipe. We simply introduce the momentary value of that temperature into the equation, and modify that value as the simulation proceeds.

6.15 Projects

[P6.1] Grid–Width Control

Implement a moving grid algorithm for the shock tube problem using the ideas that were outlined in this chapter.

6.16 Research

[R6.1] Grid–Width Control

Generalize the idea of a moving grid algorithm to hyperbolic PDEs in two space dimensions.

Develop a general theory for assessment of the consistency error, and derive a grid–width control algorithm that contains the consistency error in a reliable and systematic fashion.