Chapter 1

Finite-difference modeling of tensor fields

In this chapter I describe numerical algorithms based primarily on finite-differences that model wave propagation in arbitrary complex media. Spatial complexity is not restricted, and for seismic problems arbitrary stress-strain relationship can be employed. The numerical implementation is modular and written in Ratfor90 (Ratfor + Fortran90 or High Performance Fortran) such that complex methods can be easily composed by using simple building blocks. The scheme used in model experiments described in this thesis allows us , in particular, to model nonlinear source effects and wave propagation in anisotropic media. I use this algorithm for modeling seismic wave propagation, although any problem that is representable as a second order tensorial partial differential equation can be directly modeled. In particular, it allows us to look at effects introduced by the medium without having to switch methods depending on the medium type. In this manner one is sure that those observed effects are not artifacts of different types of algorithms, but originate from the medium itself. I use the popular Marmousi subsurface model to compute seismic wave propagation acoustically and elastically using an algorithm that employs identical building blocks.

Finite-difference approximations to full wave propagation problems have been investigated in the past and are used extensively when information about the complete wave field is required. In principle finite difference methods approximate the fundamental equations of the physics of wave propagation. All effects caused by the governing equations are automatically included, if used properly. One important parameter in finite difference algorithms is the way in which the medium and the wave field are discretized. If discretization is carried out correctly, the finite-difference equation represents a good approximation to the original problem. Dablain (?) shows applications of high-order differencing to the scalar wave equation. Crucial parameters in finding a finite-difference solution to a continuous problem are the coefficients of the derivative operators. Holberg (?) computes optimized coefficients for finite differences in such a way as to approximate the derivative operator by minimizing the least-squares error between the analytical and the approximate derivative. Other optimization criteria are possible, such as average-source spectrum weighted least-squares or an L^p norm minimization. In all those cases one tries to avoid growing error modes, while still retaining the best possible (most accurate and shortest) derivative operator. Virieux (?) models p and sv waves in an isotropic earth model using a velocity stress formulation, that is similar to Mora (?), who in his original inversion scheme used a conjugate gradient optimizer which applied an isotropic finite difference wave equation operator to invert seismic data in a least-squares sense. Etgen (?) and Mikhailenko (?) show some theoretical and numerical details of wave equation computations on grids that are more general than Cartesian grids, including arbitrary boundary conditions. I (?) used finite differences for anisotropic imaging and material property calculation, and Cunha (?) for isotropic reflectivity retrieval using reverse time migration.

As a programming language High Performance Fortran (F90,HPF) is practical and uses a reasonably compact notation. In particular it means that starting from the most general mathematical expression does not necessarily mean starting with the most difficult of all cases. As it turns out, the most general case can be written very simply. Reducing the space dimensionality, or reducing component dimensionality in the seismic case, is merely reflected in different implicit loop limits, while the code itself remains the same. Muir (?) implemented a Lattice-Boltzmann simulation in F90 with great simplicity. Ideally, I would have liked to design this modeling framework in C++, but at the time of developing this numerical tool box, either performance of C++ on parallel computers was too low for practical problems or C++ was not even available on parallel computer architectures.

I was inspired to develop such a general tool box by two things. First, Jon Claerbout's (?)[1] programming method for his books *Processing versus Inversion* and *Applications Of Three Dimensional Filtering*; second, the Stanford Exploration Project's *Programming Geophysics in C++* (?)[1]. They both contain collections of processing subroutines that are used throughout the book as building blocks for larger subroutines and programs, at the same time they were small enough to be included in the text for practical illustration of numerical methods. When using those building blocks I was amazed by the versatility and easy maintainability. Secondly, I had written a several modeling and migration programs for various types of acoustic and elastic media. Those programs were very similar in structure and methodology and my aim was to consolidate those programs into one or more, that all use the same framework and building blocks. Those building blocks serve as a basis for a multitude of conceptually different modeling and migration programs. The blocks themselves can be modified and easily extended. I created a minimal set of the most functional modeling and migration building blocks that can be arranged with ease to create various numerical algorithms, from plain finite differences with exchangeable operators to spectral methods or any combination of those.

[1] available on Internet: http://sepwww.stanford.edu/

1.1 Wave equations

In the following sections I show the main features of the modeling scheme and building blocks by restricting myself to a general elastic wave equation. The modeling implementation is based on a wave equation of the abstract mathematical form

$$\underset{\sim}{\mathbf{a}}(\mathbf{x})\ \nabla\ \underset{\sim}{\mathbf{b}}(\mathbf{x})\ \nabla^T\ \mathbf{u}(\mathbf{x},t)\ -\ \frac{\partial}{\partial t^2}\mathbf{u}(\mathbf{x},t) = \mathbf{f}(\mathbf{x}_0,t) \tag{1.1}$$

$\mathbf{u}$ is an arbitrary wave field (scalar or vector); $\mathbf{f}$ is the force applied at source locations $\mathbf{x}_0$. ∇ is a general gradient operator and ∇^T the divergence operator applied to the wave field components in three dimensions. $\underset{\sim}{\mathbf{a}}$ and $\underset{\sim}{\mathbf{b}}$ are medium property fields such as density, velocity or stiffnesses. The second order equation (1.1) can be rewritten as a sequence of first order partial differential equations. Numerical calculations typically discretize those on a staggered grid for improved accuracy and simplified incorporation of explicit boundary conditions.

Elastic medium

For the anisotropic elastic case the differential operator ∇, defined by

$$\nabla^T = \frac{1}{2}\left(\frac{\partial}{\partial x_l} + \frac{\partial}{\partial x_k}\right) \qquad \text{with} \qquad k,l = 1,2,3 \tag{1.2}$$

operates on the displacement field $\mathbf{u}$ and results in the symmetric strain tensor $\underset{\sim}{\epsilon}$. The indices k and l, and in the following the indices i and j, range over the number of space axes and thus the number of vector components. This gives the following set – in the spatial domain – of first order equations:

$$\epsilon_{kl} = \frac{1}{2}\left(\frac{\partial u_k}{\partial x_l} + \frac{\partial u_l}{\partial x_k}\right) \tag{1.3}$$

$$\sigma_{ij} = b_{ijkl}\epsilon_{kl} \tag{1.4}$$

$$f_i = a_{ij}\frac{1}{2}\left(\frac{\partial}{\partial x_i} + \frac{\partial}{\partial x_j}\right)\sigma_{ij} + \frac{\partial}{\partial t^2}u_j \tag{1.5}$$

where $\underset{\sim}{\mathbf{a}}$ is the density (usually a scalar $a \times$ identity I) and $\underset{\sim}{\mathbf{b}}$ represents the stiffness matrix of elastic moduli. In general the stiffness matrix is a sparse matrix, and can be simplified for different degrees of symmetry. Equation (1.4) represents the stress-strain relationship in the medium and the two second-order tensors $\underset{\sim}{\epsilon}$ and $\underset{\sim}{\sigma}$ are related via the moduli's fourth-order tensor $\underset{\sim}{\mathbf{b}}$. In equation (1.5) the stress field is converted into an acceleration field and a forcing function $\mathbf{f}$ is added. I use this set of first order equations in my implementation to have flexibility in changing parameters and boundary conditions.

Acoustic medium

In the special case of an acoustic medium the stiffness matrix and the stress tensor reduce to scalar values (particle velocity and pressure). One can derive the familiar set of first order equations, relating changes in pressure to changes in velocity:

$$\rho \frac{\partial}{\partial t} u_i = - \frac{\partial}{\partial x_i} p \tag{1.6}$$

$$\frac{\partial}{\partial t} p = K \frac{\partial}{\partial x_k} u_k \tag{1.7}$$

Combining those two equations into a second order partial differential equation and incorporating the source term q gives

$$K(\mathbf{x})\ \nabla\ \rho(\mathbf{x})\ \nabla^T\ p(\mathbf{x}, t)\ -\ \frac{\partial}{\partial t^2} p(\mathbf{x}, t) = q(\mathbf{x}_0, t) \tag{1.8}$$

This corresponds to the general form (1.1), where K is the bulk modulus and ρ is the density. All these quantities are scalar and can be regarded as tensors of rank one.

It is obvious from the previous two examples that the equation structure is identical for different types of media. One can either treat the acoustic case as a completely self-contained problem or as an elastic problem where the material properties happen to be acoustic. Differences in spatial dimensionality or component dimensionality are reflected in the rank of the tensors and thus the bounds of the subscripts. From that I conclude that if the problem structure is identical the same numerical tools can be reused for different modeling purposes. In practical terms it is merely a matter of keeping track of the indices and making sure those indices are efficiently mapped in each case. Thus all the building blocks can reuse the same pieces of code if they are written in a general form.

1.2 Mathematical objects and spaces

The variables in a mathematical formula are usually represented in numerical applications as samples over a discrete field. Such variables are in reality tensorial quantities, which means they have a metric attached to them. On this metric different functions, such as basic addition, subtraction, multiplication, and division are defined. Besides those basic functions other operations such as taking a derivative or a taking an integral are given. Precisely the latter two operations should ideally be defined with the object itself, since then the object is self-contained and can be reused easily in various realizations. If, for example, the wave field is given on a curved general grid, suitable differentiating and integrating functions should be defined with it. This is very easily possible in a C++ implementation, but much harder to do in a Fortran implementation of the numerical algorithm. Thus I restrict myself to a regularly connected sampled grid, that is not completely general. All my differentiating or integrating function implementations assume such a regularly connected grid. The grid layout itself is outlined in 1.1 for the two-dimensional representation. The complete grid consists of several sub grids. In the interior part I calculate the wave field quantities as accurately as possible. Surrounding this portion of the grid are zones with material properties and wave equations that differ from the interior. Those properties can be adjusted in order to minimize artificial reflections from the boundaries of the interior grid.

Following is a list of basic object types that are used throughout the algorithm. The name in capital letters designates the object type, the argument is the variable that will receive those properties.

- `WFIELD(uu)` designates the variable uu as wave field. If it denotes a displacement, it has a certain number of vector components associated with it; if it denotes pressure it is a scalar field.
- `TENSOR(uu)` designates the variable uu as a second rank tensor. Thus it specifies stress or strain in the elastic medium.
- `STIFF(uu)` designates the variable uu as a fourth rank tensor and it denotes the elastic stiffness tensor.
- `BOUND(uu)` designates a masking object that defines the computational domain by specifying grid types, such as interior and exterior orientations.

Practically each of these objects is a macro that is expanded during compilation of the program. If a change to the object type definitons is centrally made, automatically all programs will receive the correct type. The major drawback in F90, unlike in C++, is the absence of functions and subroutines that are tied to those objects. I simulate them by defining all objects on the same computational field, such that operations like derivatives, convolutions or time updates always operate on the same sampled grid type.

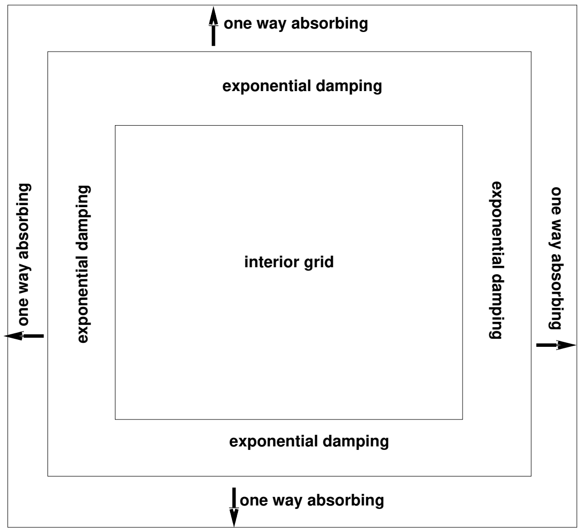

Figure 1.1: Computational grid for the finite-difference modeling calculation. The main computational interior grid is surrounded by two boundary zones, whose function is to suppress unwanted reflection from the boundaries of the interior grid. genmod-computational [NR]

For efficient computation medium types are put into categories: acoustic, isotropic, cubic, transverse isotropic (TI), orthorombic, monoclinic and triclinic. These denote material property symmetries as described in detail in Auld (?). Symmetries not only have a profound influence on seismic wave propagation, but also on the efficiency of the numerical algorithm. The internal representation in the program follows exactly the mathematical representation. Take for example the elastic stiffness matrix $\underset{\sim}{\mathbf{b}}$; its mathematical matrix notation for an isotropic solid is:

$$\begin{pmatrix} b_{11} & b_{12} & b_{12} & & & \\ b_{12} & b_{11} & b_{12} & & & \\ b_{12} & b_{12} & b_{11} & & & \\ & & & b_{33} & & \\ & & & & b_{33} & \\ & & & & & b_{33} \end{pmatrix}, \tag{1.9}$$

where b_{ij} are the elastic moduli. The mathematical form translates to a simple mask function, as can be seen in the code `getstress()` (page 12). For example, the isotropic elastic symmetry mask is as follows:

```
stiffmask(1,1)=1; stiffmask(1,2)=1; stiffmask(1,3)=1;
stiffmask(2,1)=1; stiffmask(2,2)=1; stiffmask(2,3)=1;
stiffmask(3,1)=1; stiffmask(3,2)=1; stiffmask(3,3)=1;
                                   stiffmask(4,4)=1;
                                         stiffmask(5,5)=1;
                                           stiffmask(6,6)=1;
```

Similarly, in the acoustic case, where the material parameter symmetry degenerates to

$$\begin{pmatrix} b_{11} & & \\ & b_{11} & \\ & & b_{11} \end{pmatrix} \tag{1.10}$$

it translates directly to

```
stiffmask(1,1)=1;
    stiffmask(2,2)=1;
          stiffmask(3,3)=1;.
```

The mask is merely a scaled identity matrix. For many realistic media, the stiffness matrix is a very sparse matrix with only a few non-zero elements. Furthermore, symmetries force some of the components' values to coincide with each other. That is fortunate for numerical evaluations, since then not all components need to be actually stored in the computer's memory. In this implementation I store only nonzero elements that are different from each other; other nonzero elements are deduced by symmetry masks.

1.3 Modeling structure

The outline of a modeling algorithm for the general wave equation (1.1) can be written very concisely.

```
loop over experiments {
    [apply displacement boundary conditions]
    [calculate strain]
    [apply strain boundary conditions]
    [calculate stress]
    [apply stress boundary conditions]
    [calculate divergence of stress]
    [apply acceleration boundary conditions]
    [time update the wave field]
    [record wave field]
}
```

Items that are marked with boxes are functional modules that operate on a set of field variables. These boxes can be easily interchanged in a "plug and play" manner, such that a number of different numerical methods can be applied easily. The calculation of those highlighted functions are not tied to a specific implementation, they rather represent conceptual steps in computing field quantities. The code itself reflects closely the mathematical descriptions.

Calculating derivatives

When numerically implementing differential equations, some approximation to the continuous derivative needs to be found. There are several ways in which this discretization can be achieved. In my algorithm I use the well-known method of finite-difference approximation,

$$\frac{\partial}{\partial x} u(\mathbf{x}_k + \frac{1}{2}\Delta h) = \frac{1}{\Delta h} \sum_{i=k-n}^{k+n} b_k \; u_k \tag{1.11}$$

where the grid point k corresponds to the location x_k. The number of coefficients n and the actual values of b_k determine how well such a convolutional operator can approximate the true derivative at the point $x_0 + \frac{1}{2}\Delta h$. Normally, the derivative value is required on the location x_0, but when using a staggered grid, it is needed half-way between grid points. Additionally, first-order derivatives are computed more accurately between original grid locations, as the top plot in Figure 1.2 shows. The top plot shows the centered three point finite-difference stencil $-1, 0, 1$ on the left; on the right, the corresponding amplitude spectrum is depicted. In all the plots the stencils are normalized to unit amplitude in the spatial domain, while in the frequency domain amplitude and frequency axis are normalized to π. A comparison between the desired spectral behavior – a straight line going through the origin and π – and the numerical approximation is then easily possible. The centered stencil shows the worst spectral behavior, since it only approximates the extremely low frequency spectrum. It peaks at half Nyquist frequency and decays down to zero amplitude for all frequencies above. The spectral behavior of the staggered two point stencil is better, since it approximates a longer range of the low frequency spectrum. The next stencil is already 8 grid points in length and is derived from Central Limit Theorem arguments, as described in detail in the appendix to this chapter, such that repeated self-convolution, tends to a stable self-similar operator shape. Its spectrum approaches the theoretically desired curve for a wide range and deviates above 70 percent of Nyquist frequency. The bottom picture in the comparison depicts coefficients computed by using an optimization algorithm that finds unknown stencil coefficients by minimizing the residual differences between the approximated and desired spectrum. The operator is 16 grid points in total length. The optimization assumes that all frequencies in the spectrum are equally important and does not try to weigh the optimization's cost function according to frequency content of the source and the medium. However, such a weighting could readily be done and an even more optimal set of coefficients be derived for a particular source and medium configuration. If one would increase the operator length, one could get better matches between the desired and approximated spectrum. In the limit the stencil is of the size of the total number of grid points and approaches that of a derivative computed by Fourier Transforms (which is the desired straight line in all the comparison plots). In practical terms, there is trade-off between accuracy and operator length and an optimal balance has to be found depending on the particular wave propagation problem to be solved.

The subroutine `conv1dc()` (page 9) implements such a derivative calculation via a general convolution routine that acts on any one dimension. It can be used for convolving over any dimension in space and is the heart of the finite-difference algorithm. Figure 1.3 shows the effect of operator choice on seismic wave propagation. The figure contains a comparison of the three staggered operators in Figure 1.2. The snapshots show that the accuracy of propagating a wave front in a homogeneous medium improves with operator length and an optimum choice of coefficients. The inaccuracy of the derivative operator at high frequencies causes dispersion and introduces an angle-dependent and overall error in first breaks of the snapshots. The solid circle traces the analytically correct position of the wavefront for that time when the snapshot was taken. The spectrum of the wave field is such that for the short operator the inaccuracies dominate, while for the longer operators the spectrum is treated properly. All cases lie within the stability region of the finite-difference propagator and the maximum wavelength generated by the source signal is at the limit of about three grid points per spatial wave length.

```
# conv1dc: convolution 1 dimensional circular boundary condition
# u and w are conjugate to each other
# axis: axis over which to convolve
# stagger:  0 normal conv; 1 shift by one grid point
#%
subroutine conv1dc(conj,u,t,k,w,sidx,  axis, scale,bnd)
implicit none
#include "commons"
integer            conj,  t,k,  sidx
integer                                 axis
real                                          scale
WFIELD(u)
WFIELDLAY(u)
TENSOR(w)
TENSORLAY(w)
BOUND(bnd)
BOUNDLAY(bnd)
integer ishift

if (conj==0) {                           #  the "gradient"
do ishift = nlo-stagger, nhi-stagger {
   w(sidx,:,:,:) = w(sidx,:,:,:) +
        coef(ishift+stagger)*cshift(u(t,k,:,:,:),dim=axis,shift=ishift) * scale
   }
}
else{
do ishift = nlo-stagger, nhi-stagger {  #  the "divergence"
   u(t,k,:,:,:) = u(t,k,:,:,:) +
       coef(ishift+stagger)*cshift(w(sidx,:,:,:),dim=axis,shift=ishift) * scale
   }
}
return
end
```

Strain calculation

Subroutine `del()` (page 11) calculates the strain tensor given in equation (1.3) and represented as a matrix operation in equation (1.13). I deviate here from the mathematical notation (1.3) and use abbreviated indices for speed and efficiency. Symmetries in tensor components allow indices to be abbreviated by mapping a 3×3 double index ij to a single index I (**?**). The abbreviation rule is

$$11 \rightarrow 1\ ;\ 22 \rightarrow 2\ ;\ 33 \rightarrow 3\ ;\ 23 \rightarrow 4\ ;\ 13 \rightarrow 5\ ;\ 11 \rightarrow 6 \tag{1.12}$$

Thus the tensor product can be written as regular 3×6 matrix multiplication:

$$\underset{\sim}{\epsilon} = \mathbf{K}^T\, \mathbf{u} \quad = \quad \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \\ \epsilon_6 \end{pmatrix} = \begin{pmatrix} \frac{\partial}{\partial x_1} & & \\ & \frac{\partial}{\partial x_2} & \\ & & \frac{\partial}{\partial x_3} \\ & \frac{\partial}{\partial x_3} & \frac{\partial}{\partial x_2} \\ \frac{\partial}{\partial x_3} & & \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} & \frac{\partial}{\partial x_1} & \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} \tag{1.13}$$

The strain components are calculated on staggered grid, such that their location coincides with the location of the stiffness component, with which it interacts in the

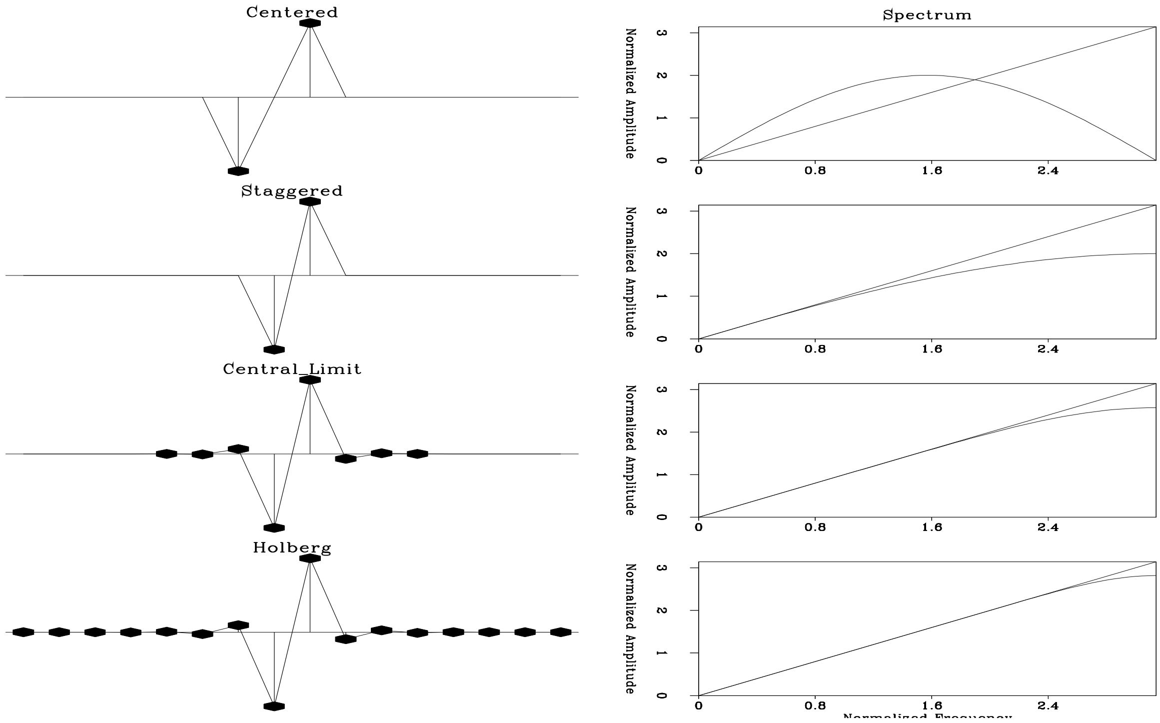

Figure 1.2: Comparison between different convolutional operators. All stencils try to approximate the analytical spectral behavior of a first-order derivative. Spectral and spatial diagrams are normalized; the ideal spectral response is a straight line. genmod-derivatives [ER]

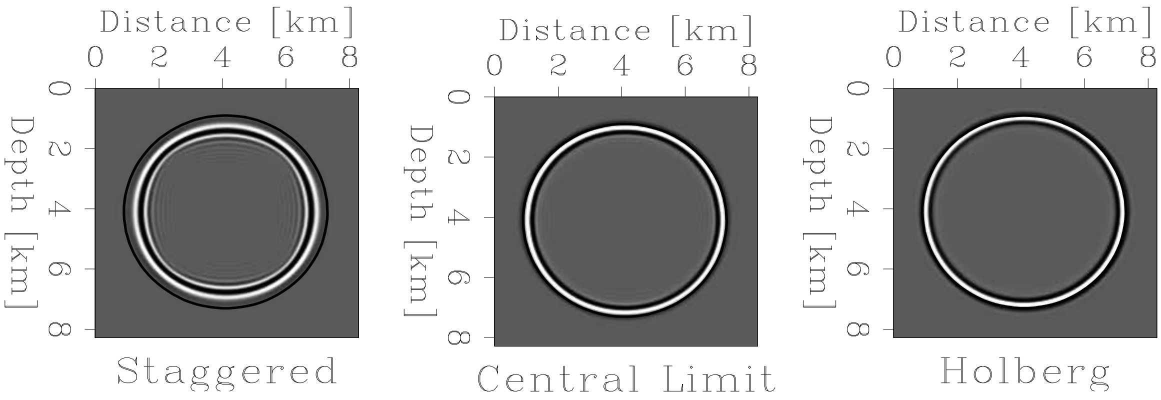

Figure 1.3: Comparison between different snapshots, computed using the stencils given in Figure 1.2. The analytically computed wavefront in this homogeneous medium is overlaid on top of the wave field. The staggered two-point filter shows dispersion as well as reduced propagation velocity since the spectrum of the wave field is not adequately treated by the difference stencil. In contrast, the central limit and the optimized stencil give similar results: the peaks of the wave field lie on top of the theoretical curve. genmod-waves2D [CR]

next calculation step. The parameter `stagger` triggers the shifting of the evaluation point. If `conj==0`, then subroutine `del()` (page 11) will evaluate ∇^T in (1.1), performing an outer product; otherwise it calculates an inner product. Displacement uu and strain ss are in that sense conjugate to each other. The actual derivative calculation is carried out using the previously introduced subroutine `conv1dc()` (page 9), which is a finite-difference approximation to a partial derivative. Instead of using a convolution for derivative computation, I could take a Fourier derivative, then I would perform spatial pseudo-spectral modeling. One could even think of alternating between the two methods depending on which spatial derivative is calculated.

```
# apply symmetric Lagrangian derivative
#
subroutine del (conj,uu,t,ss,bnd)
implicit none
#include "commons"
integer         conj,   t
WFIELD(uu)
WFIELDLAY(uu)
TENSOR(ss)
TENSORLAY(ss)
BOUND(bnd)
BOUNDLAY(bnd)
integer ivec,iten,direction
real scale

if(conj==0) {ss = 0.;        }
else        {uu(t,:,:,:,:)=0.}

do ivec=1,ncomp {
do iten=1,ntcomp{    direction=trudim(deriv(tcomp(iten),icomp(ivec)))
 if (direction>0){   scale = scaling(tcomp(iten),icomp(ivec))
  if (conj==0)  { stagger = stagc0(tcomp(iten),icomp(ivec)) }
  else          { stagger = stagc1(tcomp(iten),icomp(ivec)) }
    call conv1dc(conj,uu,t,ivec,ss,iten,direction,ddx(direction)*scale,bnd)
  }
}}

return
end
```

Stress calculation

Following the general algorithm on page 7, the next step is a stress calculation shown in subroutine `getstress()` (page 12). It performs the double inner product of a stiffness matrix with a strain tensor that results in the stress tensor. Again I use the matrix form of equation (1.4), which can be written as

$$
\begin{pmatrix} \sigma_1 \\ \sigma_2 \\ \sigma_3 \\ \sigma_4 \\ \sigma_5 \\ \sigma_6 \end{pmatrix} = \begin{pmatrix} c_{11} & & \cdots & & c_{16} \\ & & & & \\ \vdots & & & & \\ & & \ddots & & \\ & & & & \\ c_{61} & \cdots & & & c_{66} \end{pmatrix} \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \\ \epsilon_6 \end{pmatrix} \tag{1.14}
$$

Since the stiffness matrix is generally sparse I use and store only nonzero elements, as indicated by the index `k` in the subroutine, for the calculation of stress components. This general routine can be used for many types of calculations as long as the operation can be cast as a tensor inner product.

```
# getstress:    Hooke's Law
# get stresses from strain
#%
subroutine getstress(strain,stiff,stress,bnd)
implicit none
#include "commons"
TENSOR(strain)
TENSORLAY(strain)
TENSOR(stress)
TENSORLAY(stress)
STIFF(stiff)
STIFFLAY(stiff)
BOUND(bnd)
BOUNDLAY(bnd)
integer i,j,k,l

stress=0.
do i=1,ntcomp {
do j=1,ntcomp {               k = ccin(tcomp(i),tcomp(j))
   if (k!=0)  {
      stress(i,:,:,:) = stiff(k,:,:,:) * stiffmask(tcomp(i),tcomp(j))*
                        strain(j,:,:,:) + stress(i,:,:,:)
      }
}}
return
end
```

Acceleration calculation

After the stress calculation, the next step computes acceleration components by applying the tensor divergence operator to the stress field. Since the divergence operation ∇ is merely the conjugate of ∇^T, I am reusing subroutine `del()` (page 11) with `conj==1`. This calculates the acceleration components from the stress tensor by performing an inner product of the ∇ operator with the stress $\underset{\sim}{\sigma}$ and multiplying it with the inverse density $\underset{\sim}{\rho}^{-1}$ to obtain the second time derivative of $\mathbf{u}$.

The wave operator

For convenience, the right-hand side of equation (1.1) is often combined into a single operator L:

$$L = \underset{\sim}{\mathrm{a}}(\mathrm{x}) \ \nabla \ \underset{\sim}{\mathrm{b}}(\mathrm{x}) \ \nabla^T \ . \tag{1.15}$$

```
# apply the simple wave equation operator without src term
#
subroutine wave(u,strain,stiff,stress,rho3,bnd)
implicit none
include "commons"
WFIELD(u)
TENSOR(strain)
```

```
TENSOR(stress)
STIFF(stiff)
MODEL(rho3)
BOUND(bnd)

call del(0,u,3,strain,bnd)

call getstress(strain,stiff,stress,bnd)

call del(1,u,3,stress,bnd)

u(3,:,:,:,:) = u(3,:,:,:,:) * rho3

return
end
```

In the implementation one can compose the wave operator L in a short subroutine `wave()` (page 13) when the boundary conditions are neglected. This newly created operator simplifies calculations of complicated time updates of the equation (1.1) while still allowing a compact notation.

Time update

Once the acceleration at the current time is computed, the displacement at the next time step has to be calculated. This operation is independent of the spatial derivative calculations and one is left with many choices. For efficiency and accuracy I am exclusively employing explicit time update methods; pro and cons are discussed in (**?**), where explicit schemes are recommended for elastic wave propagation. The most common approach is to use a Taylor expansion in the neighborhood of the current time step:

$$\frac{\partial^2}{\partial t^2}\mathbf{u} = \frac{1}{\Delta t}(\mathbf{u}(t+\Delta t) - 2\mathbf{u}(t) + \mathbf{u}(t-\Delta t)) \approx \frac{2}{\Delta t^2}\sum_{k=1}^{n}\frac{\Delta t^{2k}}{(2k)!}\left(\frac{\partial^{2k}}{\partial t^{2k}}\mathbf{u}\right) \quad n = 1,2,3,... \tag{1.16}$$

For the time update one can choose upto what order in time to carry out the update. The straight-forward approach uses a Taylor expansion in the neighborhood of the current time step to calculate wave field values at a future time. The simplest update is a 2nd order update, which is used in the subroutine `update2()` (page 14).

```
# update2
# second order time update
#%
subroutine update2( dt2, u , bnd )
#include "commons"
real                dt2
WFIELD(u)
WFIELDLAY(u)
BOUND(bnd)
BOUNDLAY(bnd)

do ic = 1, ncomp {
  where(bnd==INTERIOR .or. bnd==FREE)
  u(3,ic,:,:,:)=2.*u(2,ic,:,:,:)-u(1,ic,:,:,:)+dt2*u(3,ic,:,:,:)
  u(1,ic,:,:,:)=u(2,ic,:,:,:)
  u(2,ic,:,:,:)=u(3,ic,:,:,:)
```

```
    endwhere
}

return
end
```

If one wants to calculate to higher orders of time one can use the general version of the time update of order n, which applies the wave operator L repeatedly to the wave field, such that successive partial derivatives get added into the time update. The update equation (1.16) can be expressed in terms of spatial derivative rather than temporal derivatives by realizing that $-L = \frac{\partial^2}{\partial t^2}\mathbf{u}$. Alternatively, one can employ a Chebychev time update that similarly calculates $L\mathbf{u}$, and that uses coefficients of a Chebychev Polynomials, instead of Taylor coefficients. Time updates thus can be achieved recursively to high accuracy, in such a manner that the choice of time step becomes less critical and one is free to choose parameters appropiate to the problem, not the particular numerical algorithm.

1.4 Examples

The outlined modeling program lends itself easily to comparisons between different types of modeling, for example acoustic versus elastic, or isotropic versus anisotropic modeling. Since the same pieces of code are used, one can be sure that differences in the modeling results are most likely produced by medium properties themselves and not by artifacts introduced by a difference in the modeling algorithms.

1.4.1 Marmousi velocity model

The Marmousi data has in recent years been used as a benchmark to compare different acoustic imaging and velocity estimation algorithms (**?**). Seismic data together with the velocity and density models are available from the Institute Francais du Petrole. As an example, I use the Marmousi subsurface model, shown in Figure 1.4 with its original pressure wave velocity and density. Since the original subsurface model lacks the specification of a shear wave velocity model for the elastic modeling, I created shear wave velocities assuming a constant Possion's ratio. Thus, P-and S-wave velocities in the following examples are strongly correlated. The elastic and acoustic seismogram modeling results are shown in Figures 1.6 and 1.7. In both cases pressure sources were used and horizontal and vertical displacement components were recorded on the surface. The left side of the figures show the acoustically modeled seismograms, while the right side of the figure shows the difference between elastic and acoustic modeling. Note that the figures are plotted to the same scale. The converted energy in the elastic wave field is nearly as large as the acoustic primary and reflected energy. The elastically converted waves are mainly visible in conversions at sharp material interfaces. The absolute first P-wave arrival time is not changed at all: the amplitude is simply reduced because of energy leaking into converted modes.

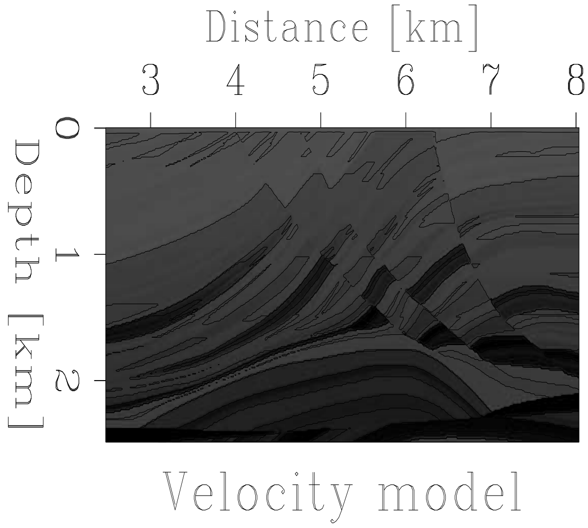

Figure 1.4: A portion of the Marmousi subsurface velocity model that is used to model seismic wave propagation. genmod-marm [CR]

Figure 1.5 compares the divergence of both types of modeling, acoustic and elastic. It shows the fully computed wave field cube in time and space. The front cube is a snapshot of the wave field at 0.810 seconds. The source was close to the surface at horizontal distance of about 3 km. The top face shows the seismogram that would have been recorded if receivers had been placed at the surface of the model. Originating from the source A cone that originates from the source, extends in time away from the source. The delimiting energy corridor is the direct and surface wave energy, while at later times the reflection from the interfaces are recorded up to a maximum recoding time of 3 seconds. The side panel displays the wave field that would have been recorded with receivers in a bore hole traversing the formation at a horizontal distance of 3.6 km, extending from the surface down to the maximum depth of the model. Mode conversions can be seen on all three faces of the elastic wave field data cube.

The divergence is computed from the wave field components using an identical procedure that allows one to compare the P-wave types more easily. The wave field shows that, as expected, pure acoustic modeling cannot account for most of the mode-converted waves. However, a realistic modeling of the earth should include mode-converted waves (p, sv and sh). Since mode conversions are absent in the pure acoustic modeling, the amplitudes of even the primary waves are different. No energy leaks into other modes, but rather remains within the P-wave mode. The most noteworthy event is the strong amplitude of a mode-converted reflection from the top free surface. A non-converted reflection is hardly visible on the acoustic modeling plot, but the free surface boundary conditions clearly cause a strong conversion. Internal mode conversions are less prominent, but are still present.

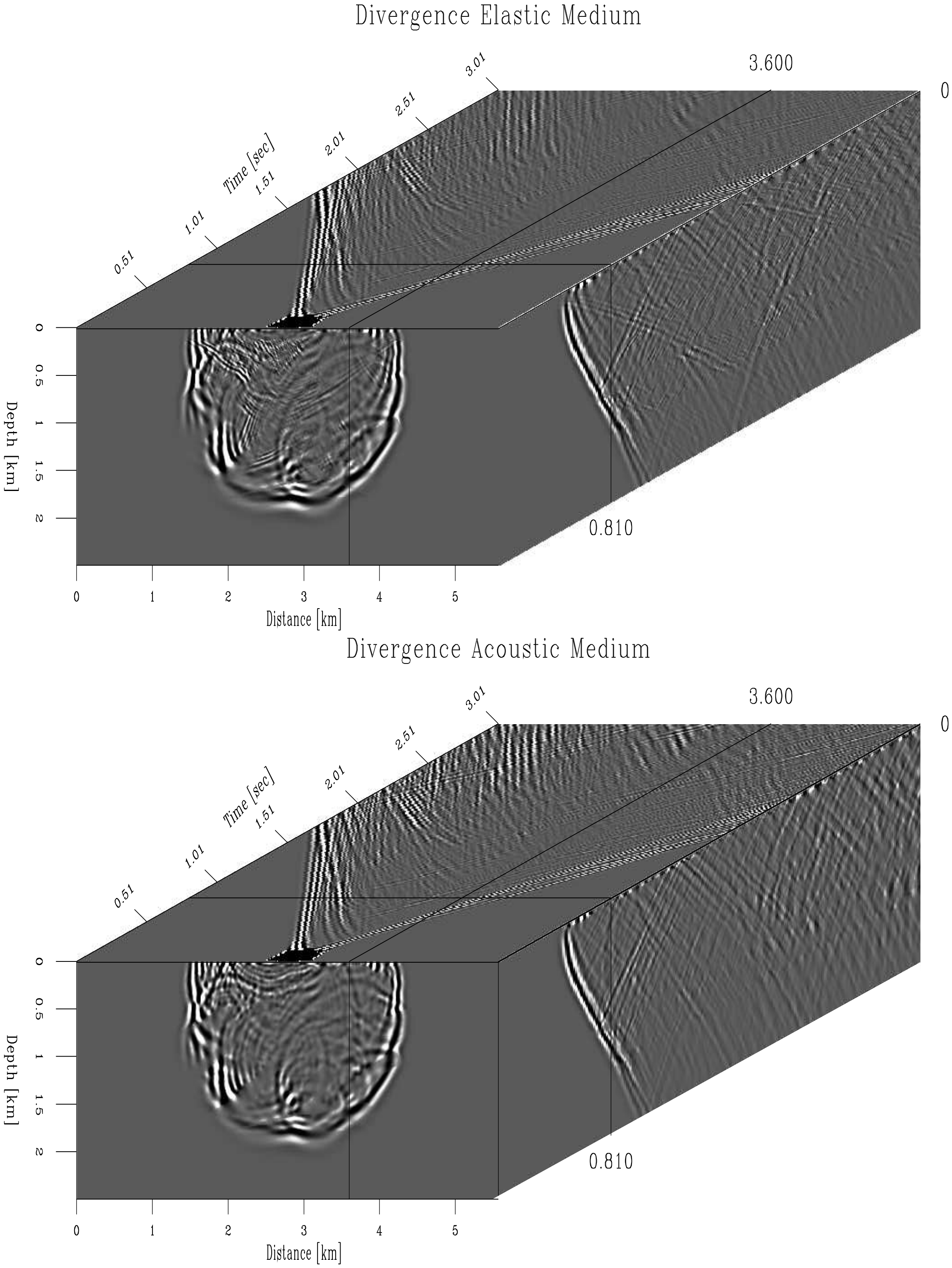

Figure 1.5: Divergence computed from acoustic and elastic modeling schemes. The acoustic modeling does not show mode-converted waves, while the elastic one does show conversions from P to S waves. Energy converts at the free surface resulting in a secondary wavefront propagating downward. But mode conversions are occuring also at sharp interfaces with large contrast. genmod-cubes [CR]

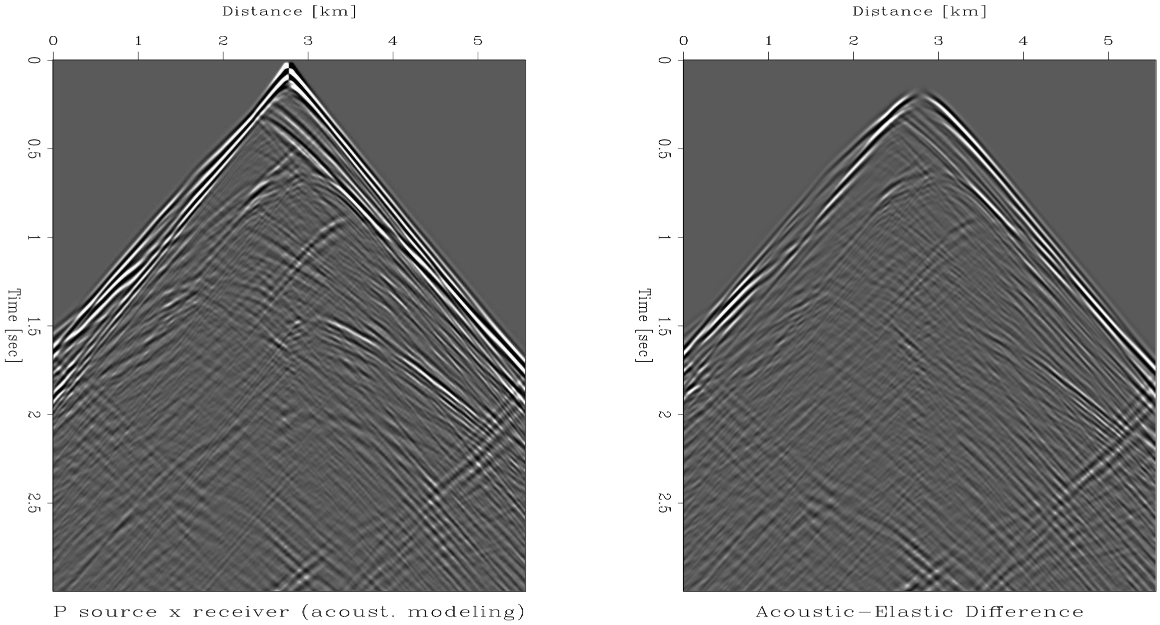

Figure 1.6: The left plot shows an acoustically modeled seismogram (p source into x receiver): the right plot shows the difference between acoustic and elastic modeling clipped to the same scale. genmod-compx [CR]

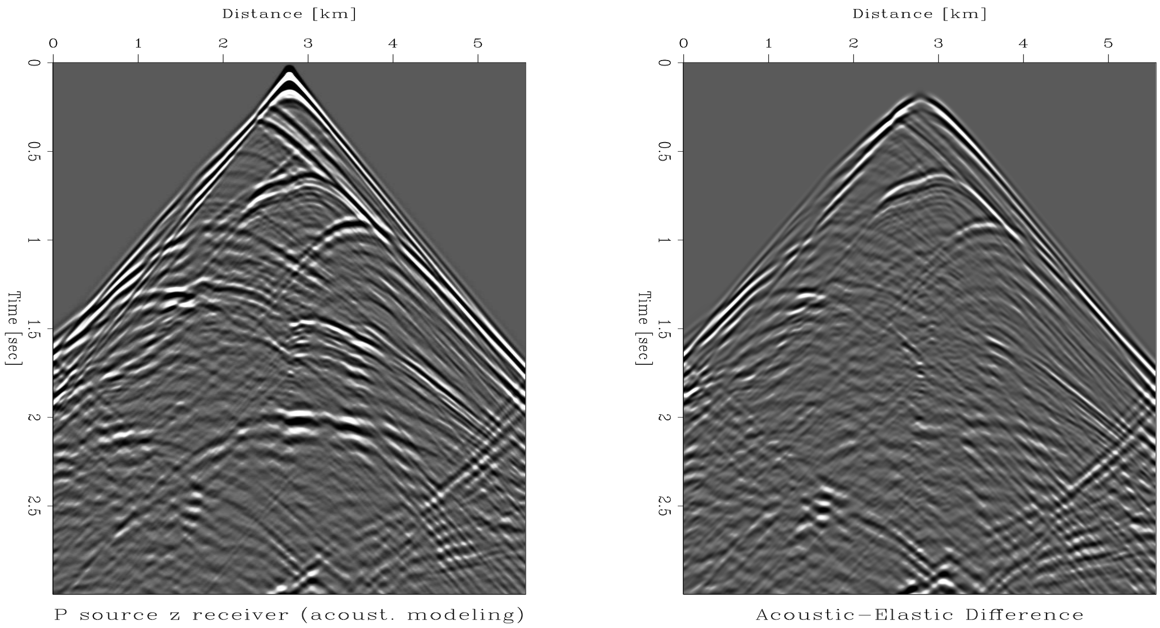

Figure 1.7: The left plot shows an acoustically modeled seismogram (p source into z receiver), the right plot shows the difference between acoustic and elastic modeling clipped to the same scale. genmod-compz [CR]

1.4.2 Summary

By expressing a second order wave equation in terms of mathematical tensor operators, I have accumulated a tool box for acoustic and elastic 2D and 3D modeling and migration algorithms. These building blocks can be put together in many different variations and the existing set can easily be expanded. Only one version of the basic source code has to be maintained. The computational speed is slower than for special purpose programs, but those modules are easy to understand, small enough to be included into scientific papers and, most important, their code is not redundant. I see this as a big improvement over monolithic algorithms and source codes and it helps to demonstrate concepts and their numerical implementation in manageable, comprehensible pieces.